UNIVERZA V LJUBLJANI

EKONOMSKA FAKULTETA

MAGISTRSKO DELO

ZAGOTAVLJANJE KAKOVOSTI RAZVOJA PROGRAMSKE OPREME V IZBRANEM PODJETJU

Ljubljana, januar 2018 PIA GRKOVIČ

IZJAVA O AVTORSTVU

Podpisana Pia Grkovič, študentka Ekonomske fakultete Univerze v Ljubljani, avtorica predloženega dela z naslovom Zagotavljanje kakovosti razvoja programske opreme v izbranem podjetju, pripravljenega v sodelovanju s svetovalcem prof. dr. Tomažem Turkom.

I Z J A V L J A M

1. da sem predloženo delo pripravila samostojno;

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

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

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

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

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

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

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

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

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

V Ljubljani, dne 17.1.2018 Podpis študentke: _____

KAZALO

KAZALO SLIK

SEZNAM KRATIC

QA	(ang. *Quality Assurance*); zagotavljanje kakovosti
FLS	(ang. *First Line Support)*;
SDLC	(ang. *Software development lifecycle*)
DSD	(ang. *Distributed software development*); Razpršen razvoj programske opreme

UVOD

V današnjem zelo tekmovalnem poslovnem okolju razvoja programske opreme so podjetja podvržena velikim pritiskom, da so čimbolj učinkovita in uspešna v razvoju programske opreme. V kolikor podjetju ne uspe najti strategij za zmanjšanje stroškov razvoja, bo to uspelo konkurenci. Znižanje cen, hitrejši razvoj in dokončanje produktov so ključni za prevzem strank in pridobivanje konkurenčne prednosti (Watkins, 2009, str. 1).

Podjetja, ki so pri projektih uporabljala slapovni model, so se več let trudila dovolj vnaprej planirati, analizirati in oblikovati, da bi bil projektni urnik čimbolj natančen in imel dovolj manevrskega prostora za soočanje z nepredvidljivimi situacijami. V realnosti se je velikokrat zgodilo, da so bili projekti prekinjeni ali pa so bili zaključeni s prekoračenimi roki ali proračunom ter z okrnjenimi funkcionalnostmi, kot so bile sprva planirane (Schiel, 2010, str.11).

Kot odgovor na formalne, težke, neučinkovite in birokratske razvoje programske opreme se je razvil agilni razvoj programske opreme (Kong, 2007, str. 12).

Leta 2001 je 17 avtorjev spisalo Manifest agilnega razvoja programske opreme, ki ga lahko najdemo na spletni strani agilnega manifesta (Agile Manifesto, 2001): “Odkrivamo boljše načine razvoja programske opreme, tako, da jo razvijamo, in pri tem pomagamo tudi drugim. Naše vrednote so ob tem postale:

- **Posamezniki in interakcije** pred procesi in orodji,
- **Delujoča programska oprema** pred vseobsežno dokumentacijo,
- **Sodelovanje s stranko** pred pogodbenimi pogajanji ter
- **Odziv na spremembe** pred togim sledenjem načrtom

Z drugimi besedami, četudi cenimo dejavnike na desni, vseeno bolj cenimo tiste na levi.”
Razvojno skupino sestavljajo: programerji, testerji, oblikovalci, produktni vodje in ključni nosilci interesov. Vsi igrajo vlogo pri kakovosti končnega izdelka. Odgovornost za kakovost nosijo vsi v razvojni skupini in ne le skupina za zagotavljanje kakovosti (ang. *Quality Assurance*). Vsaka vrstica kode, ki nastane pod okriljem razvojne skupine, naredi uporabniško izkušnjo ali dobro ali slabo (Atlassian, 2016).

Buenen in Teje (2014, str. 7) v poročilu ugotavljata, da v zadnjih letih narašča pomen skupine za zagotavljanje kakovosti (v nadaljevanju QA) in testiranja, prav tako podjetja vedno več sredstev namenjajo temu področju.

Testiranje programske opreme je pomembna aktivnost, še posebej v poslovno kritični programski opremi. Končni uporabniki pričakujejo 100-odstotno delujoč program (Scrum Alliance, 2016).

Testiranje je pomembno zaradi različnih razlogov, in sicer, da se odkrije napake v programu, preveri kakovost kode in preveri uporabnost kode (Crispin & Gregory, 2009, str. 97).

Luo (b.l.) pravi, da je testiranje vpeto v vsako stopnjo življenjskega cikla razvoja programske opreme, vendar pa je narava testiranja v vsaki stopnji različna in ima različne cilje:

- testiranje enot (ang. *Unit testing*) se izvaja na najbolj osnovni stopnji;
- integracijsko testiranje (ang. *Integration testing*) se izvaja, ko sta dve ali več testiranih enot povezani v večjo strukturo;
- sistemsko testiranje (ang. *System testing*) je namenjeno celovitemu testiranju. Testiranje večinoma temelji na funkcionalnih specifikacijah/zahtevah sistema;
- testiranje ustreznosti (ang. *Acceptance testing*) se izvaja, ko dokončan sistem preide od razvijalca do stranke, oziroma uporabnika. Namen testiranja ustreznosti je potrditev delovanja sistema in ne iskanje napak.

Crispin in Gregory (2009, str. 22) sta mnenja, da testiranje poganja agilne projekte, posledično povratna informacija igra veliko vlogo v kateremkoli agilnem timu. Eden najpomembnejših doprinosov testerja je pomoč lastniku produkta ali stranki prepoznati zahteve za vsako zgodbo v obliki primerov in testov.

Podjetje, ki ga bom preučila v magistrski nalogi, razvija lasten produkt, ki ga uspešno trži tako v Sloveniji kot tudi tujini. V podjetju so bili zaradi povečanega obsega dela, števila in raznolikosti strank primorani spremeniti način dela in razvoja produkta za doseganje večje učinkovitosti in uspešnosti. Prav tako so spoznali, da je testiranje zelo pomemben del razvoja programske opreme, predvsem v agilnem načinu razvoja, ki se ga poslužujejo. Zato je podjetje pred letom dni vpeljalo posebno skupino za zagotavljanje kvalitete, imenovano First Line Support skupina (v nadaljevanju FLS), ki skrbi za testiranje produkta (sistemsko testiranje). Produkta se poslužuje veliko število strank pri katerih so projekti v različnih razvojnih fazah. Ob testiranju v skupini skrbijo tudi za podporo končnim uporabnikom. Torej za stranke, ki produkt že uporabljajo v produkcijskem okolju. S pomočjo znanja, pridobljenega v teoretičnem delu, bom analizirala delovanje podjetja in skupine. V zadnjem delu naloge bom podala lastne predloge za izboljšavo delovanja skupine podkrepljene s teoretičnim znanjem in analizo.

V magistrskem delu bom analizirala proces vpeljave FLS skupine in potek njenega dela. Skupina skrbi za sodelovanje in testiranje na projektih, ki so še v fazi razvoja, ter za sodelovanje na projektih, ki so že vpeljani v produkcijo, kjer pa je ob testiranju poseben izziv skupine podpora končnim uporabnikom. V nalogi bom preučila stanje skupine po letu dni obstoja in uspešnost vpeljave skupine v podjetje. Prav tako bom preučila možnosti za izboljšavo delovanja in načina dela skupine.

V magistrski nalogi bom poiskala odgovor na raziskovalno vprašanje ali je bila vpeljava in način organizacije FLS skupine v podjetju uspešna.

Namen magistrske naloge je preučitev uvajanja in organiziranosti skupine za zagotavljanje kakovosti v podjetju ter analiza delovanja skupine. Tako bom lahko na podlagi ugotovljenih dejstev in preučene literature predlagala boljšo organizacijo skupine ter izboljšati in formalizirati postopke njenega delovanja.

V magistrski nalogi sem si zastavila več ciljev, s pomočjo katerih bom prišla do odgovora na raziskovalno vprašanje in dosegla namen magistrske naloge.

Prvi cilj magistrske naloge je preučiti zgodovino teorije razvoja, vzdrževanja ter zagotavljanja kakovosti programske opreme, kakor tudi novejše agilne pristope, ki jih podjetja uporabljajo pri razvoju sedaj. Tudi opisano podjetje je v preteklosti uporabljalo starejše modele pri razvoju (npr. slapovni model), sedaj pa sledi najnovejšim smernicam razvoja programske opreme.

Zagotavljanje kakovosti programske opreme ter kvalitetna podpora uporabnikom sta lahko za podjetje velika konkurenčna prednost. Glede na to, da se večina podjetij poslužuje agilnih pristopov, je drugi cilj magistrske naloge natančneje preučiti agilne pristope in procese pri zagotavljanju kvalitete programske opreme.

Tretji cilj magistrske naloge je na podlagi preučene literature in preteklih izkušenj v podjetju predlagati izboljšano organizacijo skupine za zagotavljanje kvalitete ter formalizirati postopke pri podpori uporabnikov, kakor tudi pri testiranju v različnih stopnjah razvoja programske opreme (analiza, razvoj, predaja stranki, vzdrževanje).

Magistrsko delo bo sestavljeno iz teoretičnega in praktičnega dela. V teoretičnem delu se bom osredotočila na teorijo s področja razvoja programske opreme, testiranja programske opreme in življenjskega cikla razvoja programske opreme.

Naredila bom kratek pregled zgodovine načinov razvoja in testiranja programske opreme ter kako se je področje razvijalo do današnjega dne. Preučila bom trenutno najbolj aktualen način razvoja, agilni razvoj in z njim povezano testiranje.

Na praktičnem primeru slovenskega podjetja za razvoj programske opreme bom prikazala njihovo dobro prakso vpeljave agilnega razvoja, reorganizacije celotne skupine zaposlenih, ki razvija lasten produkt. Pomemben del pri reorganizaciji je bila uvedba skupine imenovane FLS, ki skrbi za kakovost produkta in podporo končnim strankam. Preučila bom delovanje skupine, njene delovne procese in organiziranost testiranja produkta v vseh projektih.

Na koncu bom na podlagi preučene literature in preučitve stanja v podjetju predstavila možne predloge za izboljšavo.

1 ZGODOVINA RAZVOJA IN TESTIRANJA

Obstoječe metodologije razvoja programske opreme lahko razdelimo v glavnem na dve večji skupini, in sicer na metodologije, ki so nastale pred letom 2000 in tiste, ki so nastale po letu 2000. Vsaka metodologija vsebuje tudi testiranje, ki pa se prav tako razlikuje med metodologijami.

Vsako od spodaj opisanih metodologij sestavlja tudi testiranje. Zavedanje o pomembnosti testiranja sega v vse metodologije, ne glede na to ali gre za novejše ali že zastarele metodologije.

Myers (2004) je že leta 1979 opazil trend, da podjetja polovico časa trajanja projekta in polovico stroškov namenijo testiranju. Enak trend opazuje tudi dandanes.

V nadaljevanju poglavja so opisane najbolj prepoznane metodologije do leta 2000, v naslednjem poglavju pa sledi opis metodologij po letu 2000.

1.1 Opis metodologij pred letom 2000

Razvoj programske opreme se že od samega začetka dalje spreminja, izboljšuje in prilagaja tako poslovnim kot tudi tehnološkim zahtevam in spremembam. Zaradi nenehno spreminjajočega se poslovnega okolja, so se v zgodovini različne metodologije za sledenje potrebam naročnikom in sledenje oziroma premagovanje konkurence. (IT knowledge portal, b.l.)

Vsak projekt izdelave programske opreme gre skozi različne faze ali tako imenovan življenjski cikel razvoja programske opreme (ang. *Software development lifecycle*, v nadaljevanju SDLC). Ghaharai (2015) pravi, da je SDLC proces izdelave programske

opreme. SDLC sestavlja več faz in vsako fazo sestavljajo druge aktivnosti. Kot faze SDLC pa omenja:

- analiza zahtev,
- načrtovanje,
- implementacija,
- testiranje ter
- namestitev in vzdrževanje.

Turban, McLean in Wetherbe (1999, str. 604) pravijo, da ne obstaja standardiziran in univerzalen življenjski cikel razvoja programske opreme. SDLC je sestavljen iz splošnih kategorij, ki prikazujejo večje korake. Znotraj teh kategorij pa najdemo posamezne naloge. Te naloge niso prisotne na vseh projektih, saj za določene projekte ne igrajo nobene vloge. Predvsem se to vidi pri manjših projektih.

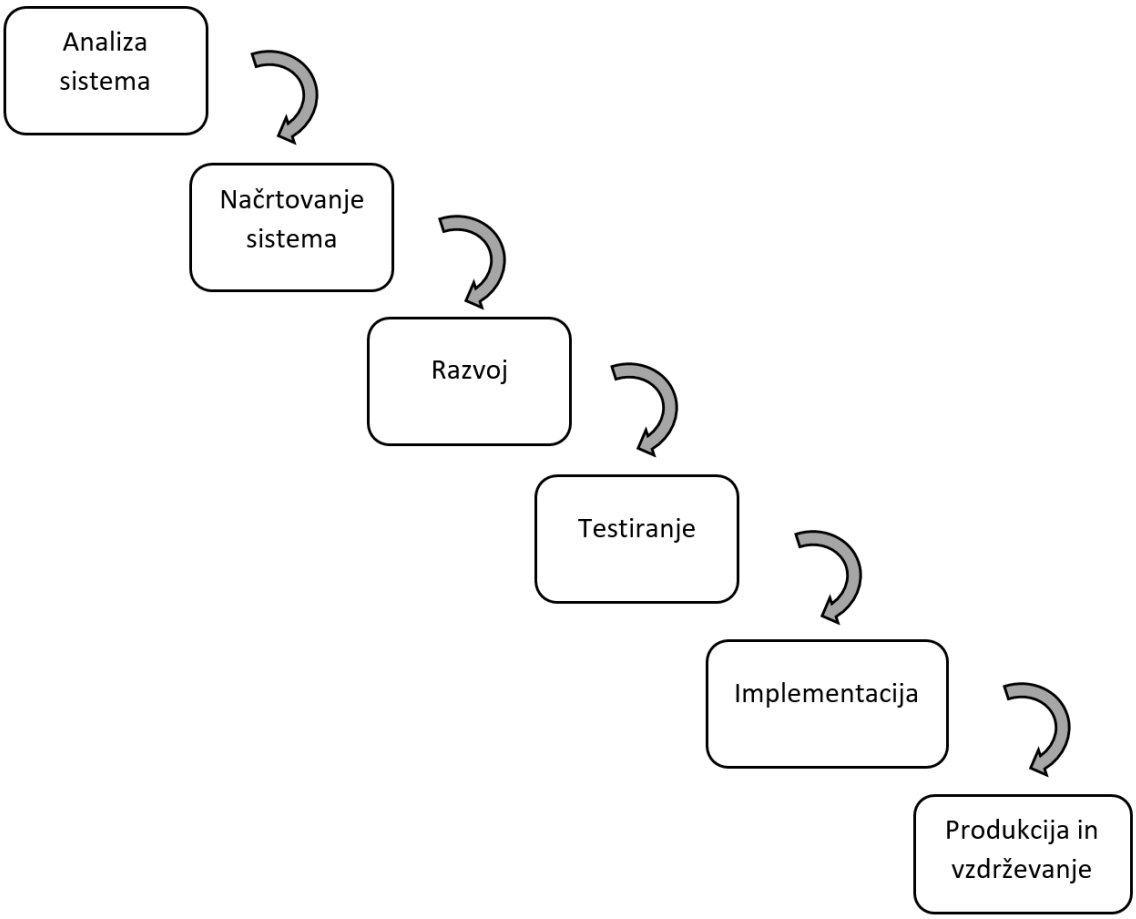
1.1.1 Tradicionalni življenjski cikel razvoja sistema

Tradicionalni življenjski cikel razvoja sistema ali tudi slapovni model (ang. *Waterfall*) se je razvil v šestdesetih letih prejšnjega stoletja, v času, ko so bili programski jeziki okorni, računalniki zelo počasni in z omejeno zmogljivostjo strojne opreme. Slapovni model so zaznamovale dobro definirane razvojne faze, kvalitetno napisane funkcionalne specifikacije. Dobro napisana dokumentacija z zahtevami je pomenila osnovo za komunikacijo med razvijalci v velikih ekipah. Projekti v slapovnem modelu so bili dolgoročni, obvezni del projektov so bile dobro definirane pogodbe. Ko so bile pogodbe podpisane, se je pričakovalo, da se zahteve ne bodo spreminjale (Jiang & Eberlein, 2009, str. 3735).

Slapovni model predvideva, da ima projektna skupina skoraj popolne informacije o zahtevah projekta, rešitvah in cilju. Spremembe v zahtevah niso bile spodbujane in so pomenile velik strošek. Zaporedje korakov, kot ga predvideva slapovni model, je bilo redkokdaj v praksi tako, kot ga predvideva teorija. Slapovni model ni učinkovit pri iskanju rešitev za potrebe strank, upravljanja hitro spreminjajočih zahtev, časa dostave in stroškov projekta (Lei, Ganjeizadeh, Jayachandran & Ozcan, 2015, str. 1).

Skozi literaturo najdemo različne interpretacije stopenj slapovnega modela. Slapovni model, kot si ga zamišljata Laudon in Laudon (2014, str 537) je prikazan na spodnji sliki.

Slika 1: Slapovni model



Vir: Laudon & Laudon (2014).

Pri iskanju definicije slapovnega modela naletimo na več interpretacij, saj veliko avtorjev zagovarja svoje različice slapovnega modela. Enotna definicija je težko dosegljiva, saj se izkušnje sodelujočih v razvoju programske opreme razlikujejo, prav tako pa je vsak projekt zgodba zase. Tudi v izbranem podjetju, ki ga opisujem v nadaljevanju, je podjetje najprej sledilo slapovnemu modelu in se je pri vsakem projektu (pa čeprav gre za isti produkt) soočalo z različnimi izzivi. Model oziroma del vpeljanega modela, ki je na primer ustrezal enemu projektu, se je lahko na naslednjem projektu izkazal za neuporabnega in ga je bilo potrebno prikrojiti novemu projektu.

Silver (2014) kot prednosti slapovnega modela našteva:

- je lahko razumljiv,
- model je lahko upravljati,
- naenkrat lahko poteka samo ena stopnja modela,

- dobro se obnese pri majhnih projektih, kjer so zahteve dobro razumljene,
- mejniki so dobro razumljeni,
- delovne naloge je lahko razporediti ter
- proces in rezultati so dobro dokumentirani.

Kot slabosti slapovnega modela pa se pojavljajo sledeče lastnosti (Tutorialspoint, b.l.):

- delujoča programska oprema je na voljo šele pozno v življenjskem ciklu,
- visoko tveganje in negotovost,
- slab model za kompleksne in objektno orientirane projekte,
- slab model za dolge in že obstoječe projekte,
- ni primeren za projekte, kjer obstaja velika verjetnost sprememb zahtev,
- težko je meriti napredek znotraj posamezne stopnje,
- ni možno spreminjati zahtev oziroma je sprememba zahtev zelo draga,
- sprememba obsega projekta med življenjskim ciklom lahko konča projekt in
- integracija je narejena šele pri koncu cikla kar onemogoča identifikacijo tehničnih in poslovnih težav in ovir zgodaj v projektu.

1.1.2 V - Model

Tako kot pri slapovnem modelu, si tudi pri V-modelu posamezne faze sledijo ena za drugo, prav tako je lahko v izvajanju le ena faza naenkrat. Model je dobil ime po sami oblikovni zasnovi faz, ki tvorijo črko V.

Powell-Morse (2016) poudarja, da se med vsako fazo načrtovanja programske opreme v V-modelu izdeluje tudi testne scenarije, ki pa se jih nato izvaja v poznejših fazah. Fazam, namenjenim načrtovanju, sledi faza implementacije in razvoja, tej fazi pa sledijo faze testiranja prej definiranih testnih scenarijev.

Kot prednosti V-modela navaja:

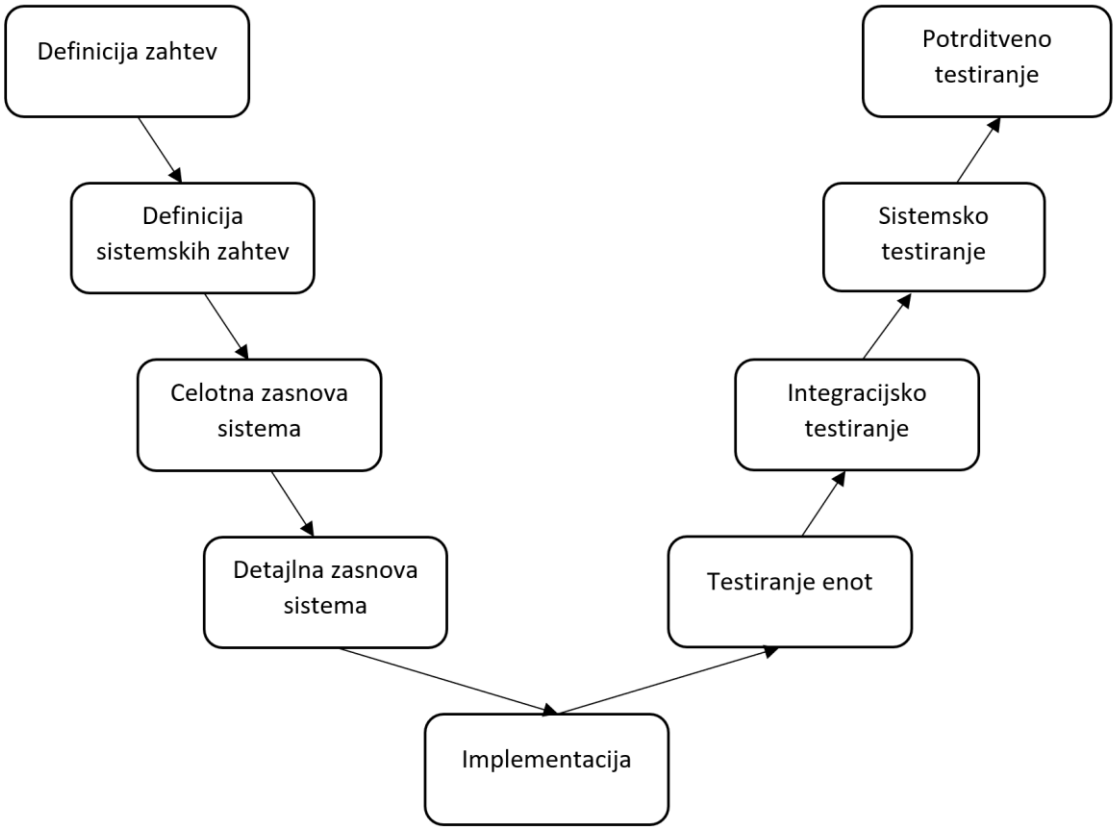
- primeren je za projekte, kjer je dolžina in obseg projekta natančno definiran, s stabilno tehnologijo. Prav tako pa je dokumentacija in načrt razvoja dobro specificiran,
- primeren je za projekte s strogo zastavljenimi časovnimi roki ter
- lahko razumljiv.

Kot slabosti V-modela pa dodaja:

- je neprilagodljiv,

- testiranje je postavljeno v zadnjo fazo, kar pomeni da se zmeraj kompenzira na kvantiteti in kvaliteti testiranja, da podjetje ujame časovni rok in
- ni primeren za časovno dolge projekte.

Slika 2: V-model



Vir: Balaji & Sundararajan Murugaiyan (2012).

Vsaki fazi pripada določen tip testiranja. Ob vsaki fazi se sočasno izdeluje tudi pripadajoče testne scenarije, ki pa se jih v izvaja v zaporedju od testiranja enot proti potrditvenemu testiranju.

Čeprav je tudi v V-modelu testiranje potisnjeno na konec cikla, pa v primerjavi s tradicionalnim življenjskim ciklom predvideva vpletenost testiranja že v začetni fazi cikla. Priprava posameznih tipov testnih scenarijev poteka vzporedno z fazami razvoja, medtem ko je testiranje izvedeno na koncu.

1.1.3 Prototipna metoda

Prototipna metoda je proces razvoja programske opreme, kjer razvijalci razvijejo določen del funkcionalnosti in jo predstavijo končnim uporabnikom in na podlagi povratne

informacije popravijo oziroma dopolnijo funkcionalnost. Uporabniki lahko tako na predstavitvi funkcionalnosti vidijo, kako naj bi končna rešitev delovala in dobijo lažjo predstavo o novi programski opremi in njeni uporabi. S tem razvijalci dobijo potrditev, da so razumeli poslovne zahteve stranke in lahko tako zagotovijo ustreznost rešitve s poslovnimi pričakovanji (Bowman, b.l.).

Prototipna metoda se izkaže za zelo učinkovito v primeru velikih in kompleksnih projektov. Prototip večinoma ni celotni sistem in veliko detajlov ni narejenih. Cilj je le predstaviti sistem s celovito funkcionalnostjo (ISTQB Exam certification, b.l.).

Nehal (2009) kot prednosti prototipne metode našteva:

- uporabniki so aktivno udeleženi v procesu razvoja,
- uporabniki lažje razumejo, kako se razvija njihova končna rešitev,
- napake se odkrije hitreje v procesu,
- uporabniki lahko podajo povratno informacijo hitreje v procesu,
- manjkajoče funkcionalnosti se hitreje odkrije ter
- zmanjša se tveganje slabe komunikacije in nerazumevanja med uporabniki in razvijalci.

Kot slabosti prototipne metode pa našteva:

- obstaja verjetnost za povečanje kompleksnosti samega projekta od začetno postavljenih načrtov,
- delovno razvita funkcionalnost lahko deluje drugače kot pa je načrtovana uporaba polno razvite funkcionalnosti,
- nenačrtovano podaljšanje razvojnega časa za razvoj prototipa,
- ker zamenja formalno analizo in korak za analizo dizajna, to pomeni manj formalne dokumentacije med projektom. Težava nastane, ko je čez čas potrebno ta program vzdrževati, vendar je dokumentacija pomanjkljiva,
- ko uporabniki vidijo zadnjo verzijo prototipa, ne razumejo zakaj je potrebno vložiti še dodatno delo v razvoj, saj je v njihovih očeh program že končan.

1.2 Razlogi za neuspeh obstoječih metodologij

Okoli in Carillo (2010, str. 5) kot težavo tradicionalnega življenjskega cikla razvoja sistema navajata, da se zanaša na to, da so zahteve in želje stranke dobro razumljene in se v času razvoja programske opreme ne bodo poglavitno spreminjale. Ko te predpostavke ne držijo, stranka dobi sistem, ki ne zadostuje njenim potrebam in željam.

Težavo pa predstavlja tudi dejstvo, da celoten proces lahko traja od nekaj mesecev do tudi več let, kar pomeni, da stranka v vsem tem času ne vidi končnega izdelka. To predstavlja veliko tveganje, da bo projekt presegel časovni okvir, zastavljen proračun ali pa bo celo prekinjen. Tukaj se zopet vrnemo do tveganja, da se bodo potrebe in želje stranke v tem času spremenile, kar pa lahko celoten projekt zelo podraži.

Poslovni svet je se je močno spremenil in zahteva hiter odziv podjetij na spreminjajoč se trg in konkurenco. Prav tako je tehnologija močno napredovala in je postala močnejša. Prav tako pa sedaj vključuje uporabniške vmesnike, intranet in odjemalec/strežnik arhitekturo (ang. *Client/Server architecture*), ki je včasih ni bilo (Turban, McLean & Wetherbe, 1999, str. 610).

Turban in drugi (1999, str. 611) omenjajo še ostale težave SDLC:

- SDLC zahteva veliko shranjevanja podatkov in vnosa podatkov, kar pa zahteva precej človeških virov, ki bi morali biti namenjeni novemu razvoju, ne pa vzdrževanju podatkov,
- SDLC ne prenese dobro sprememb, ki pa so v dandanašnjem času zelo hitre in mora podjetje znati nanje hitro odreagirati in
- naloge si v SDLC sledijo ena za drugo in jih ni možno opravljati več hkrati.

Iz lastnih izkušenj in opazovanja različnih projektov v opazovanem podjetju, lahko pritrdim zgornjim navedbam. Na opazovanih projektih se je izkazalo, da stranke ne vedo kaj potrebujejo oziroma ne znajo svojih potreb in želja razložiti in opisati analitikom, projektnim vodjem in razvijalcem.

Šele, ko stranka oziroma končni uporabnik v roke dobi izdelano programsko opremo, ugotovi, kakšne so njene pomanjkljivosti in kje je prišlo do nerazumevanja med naročnikom in izvajalcem. Ker je od same podaje zahteve in do uresničitve zahteve minilo precej časa, popravki in dodelave zopet vzamejo veliko časa in lahko tudi zamaknejo dokončanje celotnega projekta. Popravljanje obstoječih funkcionalnosti oziroma dodelave pomanjkljivih funkcionalnosti, predstavljajo tudi velik strošek, saj zopet razvijalci razvijajo zahtevo, za katero so menili, da so jo že zaključili, torej gre za dvojno delo in trpijo funkcionalnosti, ki jih je potrebno še razviti.

Zaradi pozne dostave dogovorjenih funkcionalnosti je zmeraj najbolj trpelo testiranje, ki je bilo v večini primerov neobstoječe, saj je zanj zmeraj zmanjkalo časa in pa tudi denarja, planiranega za projekt.

S tem se je podjetje znašlo v situaciji, ko je bila stranki dostavljena programska oprema, ki je vsebovala napake in za odpravo katerih stranka ni želela plačati. Odprava napak je za podjetje predstavljala dodaten, nepotreben strošek, ki bi ga lahko odkrili že hitreje v samem razvoju programske opreme.

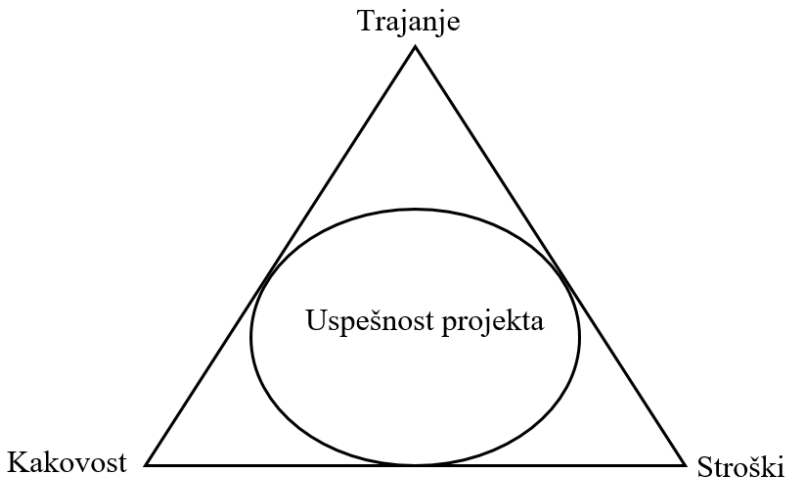
2 RAZVOJ PROGRAMSKE OPREME

V zadnjih desetletjih so razvojna podjetja vložila veliko truda in časa v iskanje dobrih praks, modelov in metod, ki bi vodile v bolj učinkovit razvoj programske opreme (Jiang & Eberlein, 2009). V glavnem se metodologije razvoja programske opreme delijo v dve kategoriji:

- klasične metodologije, ki so neokretne, delujejo po vnaprej pripravljenih načrtih, potrebujejo vnaprej definirane zahteve, dokumentacijo in detajlne plane. Sem sodita slapovni model (ang. *Waterfall model*) in spiralni model (ang. *Spiral model*) in
- agilne metodologije, ki pa so okretne in agilne. Sem sodita Ekstremno programiranje (ang. *Extreme programming*) in Scrum.

Pri razvoju programske opreme se podjetje srečuje s tremi omejitvami, ki jih prikazuje spodnja slika.

Slika 3: Trikotnik omejitev uspešnosti projekta

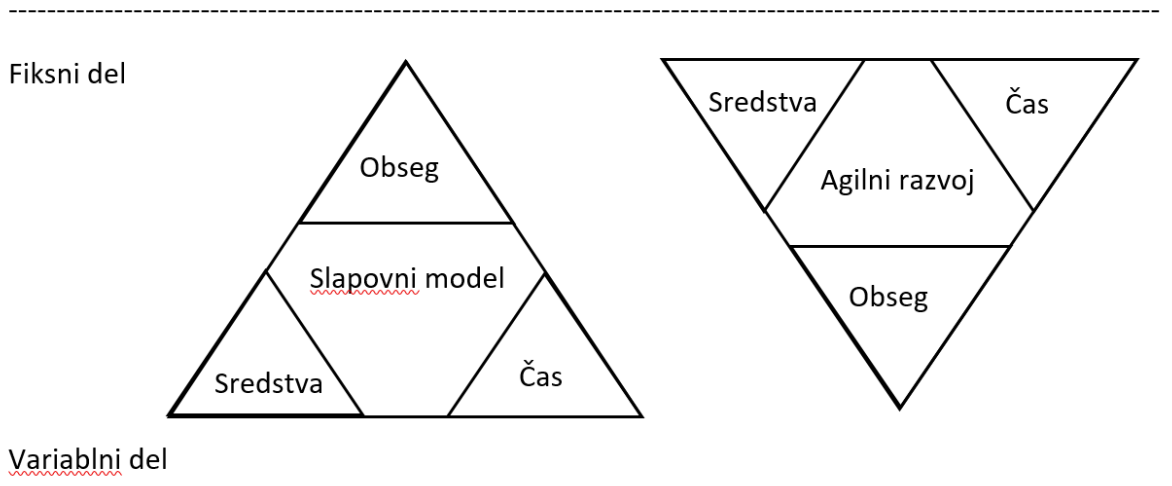


Vir: Gradišar, Jaklič & Turk (2007).

Vsak krak trikotnika predstavlja en cilj in podjetje lahko naenkrat doseže le dva cilja od treh. Na primer, podjetje lahko izveden projekt kakovostno in hitro, vendar bodo stroški visoki ali pa izvede projekt hitro z nizkimi stroški, vendar ob tem trpi kvaliteta projekta. (Kovačič et al., 2004, str. 64).

Leta 1969 je dr. Martin Barnes predlagal prvotni trikotnik, ki je sledil slapovnemu modelu. Trikotnik je predstavljal fiksni obseg projekta, sredstva in čas pa sta bila variabilna. Trikotnik se lahko aplicira tudi na agilni razvoj, v katerem pa sredstva in čas predstavljata fiksni del trikotnika, medtem ko je obseg variabilni. Razvijalci v agilnem razvoju so vezani na časovno omejene iteracije, prav tako pa agilni razvoj zagovarja konstantnost velikosti skupine. Torej, če bodo člani posamezne skupine ostali enaki in se ne bodo menjavali, to pomeni da bodo postali konsistentni in bolj učinkoviti (Atlassian, b.l.).

Slika 4: Primerjava trikotnika v slapovnem modelu in agilnem razvoju



Vir: Prirejeno po Aljaber (brez datuma).

2.1 Agilni razvoj

Agilni razvoj programske opreme se je razvil kot alternativa vnaprej planiranim in kompleksnim metodam. Agilne metode si delijo enake vrednote in načela. Agilne metode cenijo posameznike in interakcije, delujočo programsko opremo, sodelovanje s stranko, možnost odgovora na spremembe skozi proces, dokumentacijo, pogodbe in načrte. Agilni razvoj daje poudarek kratkim razvojnim ciklom, pogosti dostavi programske opreme, nenehni osebni komunikaciji in učenju. Dve priljubljeni metodi agilnega razvoja programske opreme sta Scrum in Ekstremno programiranje (ang. *Extreme Programming*). Vitki razvoj programske opreme in Kanban pa si z agilnimi metodami delita podobne vrednote in načela (Kupiainen, Mäntylä & Itkonen, 2015, str.145).

Agilni razvoj programske opreme se je razvil iz osebnih izkušenj svetovalcev in vodilnih v svetu razvoja programske opreme. Začetniki uporabe agilnega razvoja so se borili s pomanjkanjem teoretičnega dela s tega področja, saj s tem niso imeli teoretične podpore, ki bi kazala prednosti vpeljave agilnega razvoja pri doseganju ciljev podjetij (Dingsøyr, Nerur, Balijepally & Moe, 2012, str. 1217).

Pomanjkanje teoretične osnove razumevanja agilnega razvoja še zmeraj ostaja ena izmed pomanjkljivosti agilnega razvoja. Prav tako je zelo malo raziskav narejenih na temo ali je agilni razvoj alternativa tradicionalnim metodam. Čeprav obstajajo razne prednosti agilnega razvoja, njihovo razumevanje še vedno ni popolno. Prav tako raziskovalci agilnemu razvoju očitajo, da ni znanstveno podprtih izvlečkov raziskav, ki bi podprle številne trditve skupnosti, ki zagovarja agilni razvoj. Poleg tega je te trditve težko aplicirati v realnem poslovnem svetu in tudi vsako podjetje, ki prevzame agilni razvoj, mora najti sebi primeren način vpeljave agilnega razvoja, saj nekega preverjenega recepta za to ni (Yu & Petter, 2014, str.911).

Prehod iz klasične metodologije na agilno predstavlja podjetjem svojevrsten izziv. Prvi in najbolj opazen izziv je sam prehod iz klasičnega pristopa dela na agilnega in se držati agilnosti ter ne izničiti dosedanjega dela in truda v izboljšanju programske opreme in sistemov. Podjetja, ki imajo istočasno skupine razvijalcev, ki delajo po agilni metodologiji in skupine razvijalcev, ki delujejo po klasični metodologiji, imajo lahko težave pri integraciji narejenega dela obeh skupin. Življenjski cikli med obema metodologijama se precej razlikujejo in prehod na agilne metodologije lahko predstavlja izziv pri sprejemanju krajših iteracij. Uvedba agilnih metodologij na že obstoječih projektih, ki so bodisi v fazi razvoja ali pa že v fazi podpore, predstavlja svojevrsten izziv. Take projekte je težko refaktorirati in prilagoditi ustaljene poslovne procese na agilni način dela (Boehm & Turner, 2005, str. 31).

Agilni način dela zahteva vpletenost vseh skupin v vsaki iteraciji. Tradicionalni način dela pa predvideva vpletenost posamezne skupine samo v določenem fazi projekta. Pri agilnem načinu dela je zato pomembno, da so skupine v organizacijski strukturi majhne, hierarhija podjetja je čimbolj vodoravna, z več skupinami in manj nivojev managementa. Skupine morajo biti medsebojno povezane, to pomeni, da mora en član vsake skupine sodelovati na vseh ključnih sestankih drugih skupin. Agilni način dela omogoča skupinam, ki so medsebojno povezane, samo-organizirane in vodoravno organizirane, da tesneje sodelujejo med seboj, brez nepotrebnih zapletanj. Cilj agilnega načina dela je povečati sodelovanje in koordinacijo skupin in njenih članov za povečanje uspešnosti končanja projekta (Papadopoulos, 2015, str. 456).

Jochen Krebs (2009, str. 21) navaja ključne dejavnosti v agilnem razvoju :

- iterativno-inkrementalni razvoj,
- testno voden razvoj,
- neprekinjeni proces gradnje (ang. *Continuous Integration*) in
- osebna komunikacija (ang. *Face-to-face communication*)

2.1.1 Scrum

V nadaljevanju je nekoliko več pozornosti namenjene teoriji Scruma, saj izbrano podjetje magistrske naloge deluje po načelih Scruma.

Za začetnika Scruma velja Jeff Sutherland, ki je skupaj s Kenom Schwaberjem kasneje tudi spisal vodič po Scrumu (2011, str. 3), v katerem definirata Scrum kot procesni okvir, ki se od zgodnjih devetdesetih uporablja za vodenje razvoja kompleksnih izdelkov. Scrum ni proces ali tehnika za razvoj izdelkov, ampak je okvir, znotraj katerega lahko uporabimo različne procese ali tehnike.

Scrum je zasnovan na osnovi empirizma, kar pomeni, da se znanje pridobiva z izkušnjami in odločitvami, sprejetimi na podlagi znanega. Scrum uporablja iterativni, inkrementalni pristop za optimizacijo predvidljivosti in nadzora tveganja.

Scrum je ogrodje znotraj katerega lahko ljudje rešujejo kompleksne probleme pri tem pa produktivno in kreativno razvijajo izdelke najvišje možne kvalitete. Scrum je:

- vitek,
- enostaven za razumevanje ter
- zelo zahteven, če ga hočemo res obvladati.

Empirični nadzor procesov podpirajo trije stebri:

- transparentnost,
- pregled in
- prilagoditev.

Tisti, ki so odgovorni za rezultat, morajo imeti na voljo vidike procesa. Ti vidiki morajo biti usklajeni, da se opazovalci strinjajo, kaj vidijo. Uporabniki Scruma morajo pogosto pregledati Scrum artefakte in preveriti ali je napredek v smeri cilja. Najbolj učinkoviti pregledi se izvajajo na delovnem mestu, ki jih izvajajo usposobljeni inšpektorji. V kolikor

inšpektor ugotovi prevelika odstopanja pri enem ali več vidikih procesa, končni izdelek pa bo posledično nesprejemljiv, se mora prilagoditi proces ali material, ki se obdeluje. Prilagoditev je potrebno opraviti čimprej.

Krebs (2009, str. 175) k definiciji Scruma dodaja, da se Scrum pravila, vloge, produkti in besedišče ne prilagajajo posamezni organizaciji ampak se organizacija prilagodi Scrumu.

Schwaber in Suttherland (2011, str. 4) predpisujeta štiri uradne možnosti za pregled in prilagoditev:

- sestanek za načrtovanje sprinta,
- dnevni Scrum,
- revizija sprinta in
- retrospektiva sprinta.

Scrum je sestavljen iz Scrum ekip in z njimi povezanih vlog, dogodkov, artefaktov in pravil. Vsaka sestavina služi posebnemu namenu in je ključna za uspeh in uporabo Scruma.

Vsaka Scrum ekipa je sestavljena iz lastnika izdelka, razvojne ekipe in Scrum učitelja (ang. *Scrum master*). Scrum ekipe so samo-organizirane in navzkrižno funkcionalne. Člani Scrum ekipe sami odločajo o tem, kako najbolje opraviti delo in imajo vse sposobnosti za dokončanje dela. S takim načinom dela se optimizira fleksibilnost, kreativnost in produktivnost, saj na delo znotraj ekipe ne vplivajo drugi, ki niso njeni člani (Schwaber in Suttherland, 2011, str. 5).

V izbranem podjetju je v Produktni skupini dobro razvidno, da so sledili Scrum strukturi. Celotna Produktna skupina je razdeljena na več manjših ekip, ki so razdeljene po vsebinskih sklopih. Vsaka ekipa je, tako kot predvideva teorija, sestavljena iz lastnika izdelka, Scrum učitelja, razvijalca in pa vsaj enega testerja.

Malenkost drugačno zasnovo ekip pa sestavljajo ekipe znotraj Implementacijske skupine. Implementacijske ekipe so sestavljene glede na projekt. Vsaka ekipa je sicer prav tako sestavljena iz lastnika izdelka, Scrum učitelja in razvijalcev, vendar pa ena ekipa pokriva vse vsebinske sklope. Prav tako pa je v vsako ekipo vključen tudi projektni vodja. Vsaki ekipi pripada tudi po en tester, ki pa je obenem tudi član FLS skupine.

Lastnik izdelka (ang. *Product owner*) je odgovoren za maksimiranje vrednosti izdelka in dela razvojne ekipe. Je tudi edina oseba, odgovorna za upravljanje seznama zahtev izdelka (ang. *Product backlog*) (Schwaber & Suttherland, 2011, str. 5).

Krebs (2009, str. 175) definira seznam zahtev izdelka kot dinamičen koš funkcionalnosti, tehničnih delov (ang. *Technical items*) in težav, ki jih je potrebno implementirati. Vsak član ekipe lahko doda novo zahtevo v seznam, vendar lastnik izdelka prioritizira zahteve na seznamu tako da so najbolj pomembne zahteve vedno ocenjene in prioritizirane. Vsak projekt ima samo en seznam zahtev.

Seznam zahtev sprinta je seznam zahtev izbranih iz seznama zahtev izdelka, ki so planirane, da se jih uresniči v sledečem sprintu. Med potekom sprinta lahko samo razvojna ekipa spreminja seznam zahtev sprinta, v kolikor določene zahteve niso več aktualne za izvedbo. Razvojna ekipa nosi vso lastništvo nad seznamom zahtev sprinta (Schwaber & Suttherland, 2011, str. 14).

Razvojno ekipo sestavljajo strokovnjaki, ki delajo na tem, da na koncu vsakega sprinta zagotovijo razširitev »dokončnega« izdelka, ki je potencialno pripravljen na izdajo. Značilnosti razvojne ekipe so:

- se samo-organizirajo. Nihče (niti Scrum učitelj) ne določa razvojni ekipi, kako uporabiti seznam zahtev izdelka za ustvarjanje razširitve funkcionalnosti, ki je potencialno pripravljena na izdajo,
- razvojne ekipe so navzkrižno funkcionalne; imajo vse sposobnosti, ki so potrebne za ustvarjanje razširitve izdelka,
- Scrum v razvojni ekipi ne priznava nobenih nazivov razen naziva razvijalec, ne glede na delo, ki ga oseba opravlja; izjem tega pravilna ni,
- posamezni člani razvojne ekipe imajo lahko specializirane sposobnosti in področja, na katera se osredotočajo, vendar pa je razvojna ekipa odgovorna kot celota,
- razvojne ekipe ne vključujejo skupin, ki bi bile namenjene posebnim področjem, npr. testiranju ali poslovni analizi.

Velikost razvojne ekipe naj bo omejena na število članov med tri do devet. V ekipah z manj kot tremi člani se zmanjša interaktivnost in posledično zmanjša produktivnost. V ekipah, ki štejejo več kot devet članov, pa problem predstavlja prevelika koordinacija in prevelika kompleksnost za upravljanje empiričnega procesa. V velikost ekipe nista všteta lastnik produkta in Scrum učitelj (Schwaber & Suttherland, 2011, str. 6).

Scrum učitelj je odgovoren, da je Scrum razumljen in sprejet. To zagotovi tako, da se Scrum ekipa drži Scrum teorije, praks in pravil. Prav tako je Scrum učitelj vodja Scrum ekipe. Scrum učitelj pomaga lastniku izdelka, razvojni ekipi in organizaciji.

Schwaber in Suttherland (2011, str. 7) kot naloge Scrum učitelja opisujeta:

- iskanje tehnik za učinkovito upravljanje s seznamom zahtev izdelka,
- jasno izražanje vizije, ciljev in predmetov s seznama zahtev izdelka razvojni ekipi,
- poučevanje razvojne ekipe, kako ustvarjati jasne in jedrnate predmete na seznamu zahtev izdelka,
- razumevanje dolgoročnega načrtovanja v empiričnem okolju,
- razumevanje in prakticiranje agilnosti,
- omogočanje Scrum dogodkov, kot je zahtevano oziroma potrebno.
- poučevanje razvojne ekipe na področju samo-organizacije in navzkrižne funkcionalnosti,
- poučevanje in vodenje razvojne ekipe pri ustvarjanju visoko kakovostnih izdelkov,
- odstranjevanje ovir pri napredku razvojne ekipe,
- vodenje in poučevanje organizacije pri uvajanju Scruma,
- načrtovanje izvršitve Scruma v organizaciji,
- pomoč zaposlenim in nosilcem interesov pri razumevanju ter sprejemanju Scruma in empiričnega razvoja izdelkov,
- povzročanje sprememb, ki povečujejo produktivnost Scrum ekipe,
- sodelovanje z ostalimi Scrum učitelji pri povečevanju uspešnosti uporabe Scruma v organizaciji.

Dogodki se v Scrumu uporabljajo za ustvarjanje stalnosti in zmanjševanje potreb po sestankih, ki s Scrumom niso določeni. Dogodki so časovno omejeni in imajo določeno maksimalno časovno okno. Vsak dogodek je v Scrumu priložnost za pregled in prilagoditev nečesa. Dogodki zagotavljajo kritično transparentnost in pregled. V kolikor se ne izvede katerega izmed dogodkov, to vodi v zmanjšano transparentnost in izgubljeno priložnost za pregled ter prilagoditev.

Glavni del Scruma je sprint, dogodek, ki je časovno omejen na največ en mesec, med katerim je ustvarjena razširitev, ki je »dokončana«, uporabna in potencialno pripravljena na izdajo. Trajanje sprintov je vedno enako in po končnem sprintu se vedno začne nov sprint. Sprint je sestavljen iz:

- sestanka za načrtovanje sprinta (ang. *Sprint Planning*),
- dnevnega Scruma (ang. *Daily Scrum*),
- razvoja,
- demonstracije sprinta (ang. *Sprint Review*) in
- retrospektive sprinta (ang. *Sprint Retrospective*).

Med trajanjem posameznega sprinta veljajo pravila, ki se jih ne sme kršiti:

- ne uvaja se sprememb, ki bi vplivale na cilj,
- sestava razvojne ekipe se ne sme spremeniti,
- kvaliteta se ne sme zmanjšati,
- s spoznavanjem problematike med sprintom, lahko lastnik izdelka in razvojna ekipa spremenijo obseg dela znotraj posameznega sprinta.

V izbranem podjetju so se odločili, da posamezen sprint traja dva tedna, tako v Produktni skupini, kot tudi v Implementacijskih skupinah. Predvsem v implementacijskih ekipah se projektni vodje skupaj s stranko sproti dogovarjajo, kdaj se bo stranki namestilo novo verzijo.

Na sestanku za načrtovanje sprinta sodeluje celotna Scrum ekipa in je namenjen pripravi nalog, ki se jih bo izvajalo med naslednjim sprintom. Naloge za posamezne razvojne ekipe se pripravi iz seznama zahtev izdelka.

Cilj posameznega sprinta je dokončanje zahtev, ki so bile iz seznama zahtev izdelka določene za posamezen sprint.

Dnevni Scrum traja 15 minut dnevno in je namenjen uskladitvi dela znotraj razvojne ekipe. Posamezni člani ekipe na dnevnem Scrumu razložijo, kaj so počeli prejšnji dan in kaj bodo počeli tekom dneva. Dnevni Scrum je namenjen povečevanju verjetnosti, da bodo cilji sprinta doseženi. Schwaber in Suttherland (2011, str. 10) trdita, da se s prakticiranjem dnevnih Scrumov povečuje komunikacija, zmanjšuje se potreba po drugih sestankih, ki niso definirani s Scrumom, odstrani se potencialne ovire pri doseganju zastavljenih ciljev, hitro odločanje stopi v ospredje, poveča se znanje razvojne ekipe.

Po koncu vsakega sprinta se Scrum ekipa in ključni deležniki sestanejo na demonstraciji sprinta (ang. *Sprint review*). Demonstracija sprinta je neformalen sestanek, na katerem se pregleda zadolžitve zadnjega sprinta in rezultate. Razvojna ekipa razloži probleme, s katerimi se je soočala med sprintom in dobre prakse, s katerimi se je srečevala. Na demonstraciji sprinta se prav tako pogovori o nadaljnjem delu razvojne ekipe (Schwaber & Suttherland, 2011, str. 11).

Demonstracija sprinta predstavlja dobro prakso spoznavanja delovanja programske opreme tako med člani ekipe kot tudi višjega vodstva. V izbranem podjetju opažam, da se je s prisotnostjo članov ekipe na demonstracijah sprinta dvignil nivo splošnega poznavanja programske opreme in vsebinskega znanja. Razvijalci niso več strogo vpeti samo v svoje naloge in zadolžitve ampak tako spremljajo tudi delo sodelavcev in pridobivajo na vsebinskem znanju, ki je pomembno pri sodelovanju s stranko.

Retrospektiva sprinta je zadnji v vrsti sestankov in sklican po reviziji sprinta ter pred naslednjim sestankom za načrtovanje sprinta. Namenjen je samooceni Scrum ekipe in pripravi plana za izboljšanje dela v sledečem sprintu (Schwaber & Suttherland, 2011, str. 12).

Krebs (2009, str.177) opozarja na izzive, ki jih uvedba Scruma lahko prinese. Člani skupin, ki so predhodno delali na projektih s klasično metodologijo dela, se lahko spopadajo z neprilagodljivostjo novemu načinu dela. Na dnevnih Scrumih si člani skupin postavijo prevelike cilje, ki jih želijo doseči tekom dneva. Veliko ovir, ki bi v tradicionalnih organizacijah ostale skrite ali pa precej neopazne, v Scrumu lahko postanejo velika ovira za delo posameznih članov in lahko ogrozijo uspešnost Sprinta.

V izbranem podjetju se trudijo držati urnikov sestankov in jih ne odpovedovati. S Scrum načinom dela je prišlo do velikega povečanja sestankovanja razvijalcev, ki prej tega niso bili vajeni. Prilagajanje na novi način dela in na dnevno poročanje ter aktivno sodelovanje na več sestankih za določene člane skupin predstavlja negativno spremembo načina dela. Povečana količina sestankov je najbolj opazna ob koncu posameznega Sprinta in takrat je tudi pritisk na razvijalce največji, tako z vidika dokončanja nalog v iztekajočem se Sprintu kot sodelovanju pri vseh sestankih.

2.1.2 Ekstremno programiranje

Ekstremno programiranje vodi načela in prakse do ekstrema. Ekstremno programiranje zagovarja avtomatsko testiranje enot (ang. *Unit testing*), programiranje v paru in nenehno izboljševanje kode. V kolikor se upošteva tak način razvoja programske opreme, ni težav pri vpeljavi sprememb poslovnih zahtev. Za komunikacijo je poskrbljeno s tesno povezanostjo skupin, testi enot, programiranjem v paru in dostopnostjo stranke, ki je vedno na voljo za pridobivanje potrebnih informacij na želenih poslovnih zahtevah (Kupiainen, Mäntylä & Itkonen, 2015, str. 145).

Jeffries (2001) definira ekstremno programiranje kot disciplino programiranja, ki zagovarja enostavnost, komunikacijo, povratno informacijo in pogum.

Ekstremno programiranje si pri planiranju pomaga z dvema vprašanjema, in sicer kaj bo dokončano do postavljenega roka in katere naloge se lotiti naslednje. Planiranje je dalje razdeljeno na dva dela, načrtovanje izdaje (ang. *Release planning*) in načrtovanje ponovitev (ang. *Iteration planning*). V fazi načrtovanja izdaje stranka predstavi razvijalcem željene funkcionalnosti, razvijalci ocenijo težavnost naloge, čas potreben za izvedbo in stroške. Z vsemi podanimi podatki stranka pripravi načrt projekta. Skupine ekstremnega programiranja delujejo po principu dvotedenskih ciklov, na koncu katerega stranki dostavijo

delujoč in uporaben program. Stranka pred začetkom cikla predstavi naloge, katere želi, da se v prihodnjih dveh tednih izpolnijo. Razvijalci podrobno razbijejo želje na manjše naloge in jih ocenijo. To sodi v načrtovanje ponovitev.

Programiranje v ekstremnem programiranju poteka v paru, ker zagovorniki trdijo, da dva razvijalca skupaj razvijeta kvalitetnejšo kodo, kot pa en sam razvijalec. Poleg kvalitetnejše kode pa je pozitivna stran programiranja tudi v tem, da se s tem širi znanje, saj se pari konstantno menjajo in si s pogovorom med programiranjem delijo znanje.

Pierce (2016) kot prednosti ekstremnega programiranja navaja:

- manjše stroške,
- zmanjšanje časa potrebnega za razvoj programske opreme,
- malo dokumentacije,
- boljši management tveganja in
- enostavnost.

Kot slabosti ekstremnega programiranja pa navaja:

- detajlno načrtovanje,
- slabo merjenje zagotavljanja kakovosti kode,
- problem podvajanja kode,
- težko osvojljiva praksa in
- osredotočenost na kodo.

Bistvo ekstremnega programiranja je visoko kvalitetna koda, za doseganje tega pa je pomembno, da se sledi načelom programiranja v paru, refaktoriranja, vzdržnega tempa razvoja in polno pokrito testiranje na dveh nivojih. Testiranje se zagotavlja z testiranjem enot in testiranje ustreznosti (ang. *Acceptance testing*).

Pri ekstremnem programiranju se zagovarja načelo, da morajo razvijalci poskrbeti, da dokažejo strankam, da koda deluje pravilno in ne da stranka dokazuje razvijalcem, da dela koda napačno oziroma je pokvarjena (Extreme programming, 2009).

2.1.3 Testno voden razvoj

Kent Beck je v poznih 90. letih prejšnjega stoletja razvil Testno voden razvoj, ki je del ekstremnega programiranja (Fowler, 2005).

Testno voden razvoj (ang. *Test driven development*) je tehnika programiranja, kjer razvijalec najprej napiše test, nato pa šele napiše kodo na podlagi katere napisani test tudi uspe (Agile Data, b.l.).

Ko test uspe, sledi refaktoriranje kode, po tem je ponovno potrebno izvesti test. Taka iteracija se izvaja dokler ni funkcionalnost dokončana. Testno voden razvoj je del ekstremnega programiranja. Testno voden razvoj zagotavlja, da so vse enote v aplikaciji stestirane za optimalno delovanje, tako posamezno kot tudi v skupni povezavi (Rouse, 2017).

James Shore pravi, naj test ne vsebuje več kot pet vrstic kode, enako pravilo pa naj velja tudi za kodo, ki jo razvijalec spiše za vsak test. Med pisanjem kode glavni cilj ni lepo spisana koda, ampak delujoč test. Ko test oz. skupina testov uspe, takrat pride na vrsto refaktoriranje kode. V tem delu razvijalec pogleda prej spisano kodo in jo poskuša izboljšati. Refaktoriranje pa ne sme povzročiti, da bi katerikoli izmed prej spisanih in uspešnih testov, sedaj bil neuspešen (Shore, 2010).

Med prednosti testno vodenega razvoja štejemo (Ghahrai, 2016):

- vodi razvijalca v boljšo arhitekturno zasnovo kode,
- olajša vzdrževanje kode, ki je nastala pod okriljem testno vodenega razvoja,
- že v zgodnji fazi razvoja se odkrijejo pomanjkljivosti v podanih zahtevah,
- pomaga razvijalcem razumeti njihovo kodo,
- pomaga pri razjasnitvi zahtev,
- testi so varovalo proti slabo napisani kodi ter
- nepotrebne napake so odkrite že v fazi razvoja in jih je lažje odpraviti.

Hill (2015) pa kot slabosti testno vodenega razvoja navaja:

- potrebno je vzdrževati teste, kar pomeni, da je potrebno vložiti veliko časa in truda,
- določene teste je težko napisati,
- na začetku je potrebno vložiti veliko časa za sprejemanje novega načina dela in posledično razvoj poteka počasneje,
- v kolikor se razvoj zelo hitro spreminja, je potrebno zelo hitro spreminjati tudi teste,
- v samem razvoju se lahko porabi veliko časa za pisanje testov za nove funkcionalnosti, ki pa se jih na koncu zavrže.

1.1.4 Kanban

V svetu razvoja programske opreme prihaja do razhajanj ali je Kanban agilna metoda ali ne. Nekateri ga uvrščajo v vitki razvoj, drugi v agilni razvoj, spet tretji pa pravijo, da je Kanban alternativa agilnemu razvoju (Anderson, 2013).

Kanban izhaja iz Japonske, natančneje iz podjetja Toyota, in v prevodu »Kan« pomeni vizualno, »Ban« pa pomeni kartica ali tabla (Patton, 2009).

Patton (2009) pravi, da Kanban v razvoju programske opreme odstranjuje nepotrebno delo. Pri uporabi Kanbana v razvoju programske opreme:

- ni časovno definiranega razvoja,
- zgodbe so večje in jih je posledično manj,
- ocena je opcijska ali neobstoječa in
- hitrost zamenja hitrost cikla.

Marschall (2015) razlaga Kanban kot metodologijo, ki je manj strukturirana kot je Scrum. Za Kanban velja, da se ga lahko uvede na katerikoli že obstoječ proces, prehod nanj pa je manj boleč kot prehod na Scrum. Glavna sestavina Kanbana je Kanban tabla (ang. *Kanban board*), ki je razdeljena na štiri stolpce:

- v delu,
- testiranje,
- pripravljeno na izdajo in
- izdano.

Patton (2009) pa dodaja, da ima lahko vsako podjetje tudi več dodanih stolpcev in so zgoraj omenjeni le osnovni. Za obvladljivost dela je potrebno definirati maksimalno število zgodb, ki so lahko na tabli. Znotraj te številke pa je potrebno še definirati maksimalno število zgodb, ki so lahko istočasno v posameznem stolpcu. V posamezni ekipi so razvijalci, analitiki, testerji, vsi, ki so neposredno odgovorni za razvoj in prenos funkcionalnosti v produkcijo. Formula za izračun maksimalnega števila zgodb je torej prepolovljeno število vseh članov skupine. Definirano število zgodb v vsakem stolpcu pomaga poiskati ozka grla in s tem lahko ekipa takoj pristopi k zmanjševanju nakopičenega dela v posameznem stolpcu.

Kanban sistema ne gre zamenjevati s slapovnim modelom, saj v Kanbanu člani ekipe med seboj sodelujejo in si pomagajo ter nimajo strogo ločenih nalog. Potek premika zgodb na

Kanban tabeli gre z leve proti desni. Lahko pa se v katerikoli fazi zgodbe vrnejo na prejšnji stolpec, oz. stopnjo, če član ekipe meni, da delo na posamezni zgodbi v prejšnji stopnji ni bilo dokončano oz. pravilno narejeno. Na primer, če tester v zgodbi najde napake, lahko vrne zgodbo nazaj v delo (Patton, 2009).

Za razliko od Scruma, Kanban nima časovno omejenih ciklov ampak se meri čas posamezne zgodbe. Za posamezno zgodbo se vodi čas, ko je bila zgodba dodana v čakalno vrsto na tablo, datum, ko je zgodba vstopila v prvi stolpec in datum, ko je zgodba prešla v stolpec »Izdano«. Tako dobimo čas, ki je bil potreben za dokončanje zgodbe, skupaj s časom v čakanju. Za izračun časa, ko se je na zgodbi delalo, pa vzamemo datum, ko je bila zgodba dodana v prvi stolpec in datum, ko je zgodba prešla v stolpec »Izdano« (Patton, 2009).

V Kanbanu prav tako obstajajo dnevni sestanki, kjer pa skupina vsak dan pregleda stanje na tabli in se pregleda, katere zgodbe se bodo v tekočem dnevu prestavile iz enega stolpca v naslednjega (Patton, 2009).

Kanban je metoda, ki omogoča podjetjem, da so prilagodljiva. Kanban je izključno fokusiran na konstantno izboljševanje delovnih procesov posamezne organizacije. Kanban se ne fokusira na prilagajanje spremembam zahtev, ampak na prilagajanje poslovnim pogojem, tržnim tveganjem in zahtevam storitev (Anderson, 2013).

Kanban ni ne metodologija za razvoj programske opreme, ne pristop projektnega managementa. Kanban se lahko aplicira na obstoječe procese in s tem sam proces izboljša in spremeni na bolje. Je evolucijski proces, ki dovoljuje vsaki implementaciji Kanbana, da je drugačna, glede na posamezne potrebe (Kupiainen in drugi, 2015, str. 145).

Za potrebe podpore strankam na projektih, ki so že v produkciji, je izbrano podjetje ugotovilo, da Scrum ne omogoča fleksibilnosti, ki jo posamezen projekt oziroma stranka potrebuje, ko se sooča s težavami programske opreme v produkciji. Podjetje je zato na določenih projektih vpeljalo tudi Kanban za reševanje sprotnih napak, ki se pojavljajo v produkciji. Takega načina dela ni možno načrtovati vnaprej, prav tako stranka za odpravo določenih napak ne more čakati več tednov in je potrebno napake odpraviti kar se da hitro.

Prav tako Kanban omogoča spremembo vrstnega reda reševanja nalog, kar je pri reševanju produkcijskih napak ključno, saj so napake, ki bolj vplivajo na delo končnih uporabnikov postavljene na sam vrh prioritetnega reševanja.

2.1.3 Agilni razvoj na daljavo

Izbrano podjetje deluje na več različnih lokacijah, kar samo komunikacijo in delo posameznikov in skupin tudi otežuje. Shrivastava in Rathod (2015) taka podjetja uvrščata v Razpršen razvoj programske opreme (ang. *Distributed software development*, v nadaljevanju DSD). DSD pomaga podjetjem pri izboljšanju delovanja na različnih časovnih pasovih, so bližje mednarodnim strankam, prav tako pa na račun razpršenosti zaposlujejo cenejšo delovno silo.

V kolikor podjetje deluje po agilnem principu razvoja, DSD celoten razvoj oteži. Težava je v tem, ker DSD deluje na principu formalne komunikacije med skupinami, agilni princip pa temelji na neformalni komunikaciji članov, ki so na isti lokaciji. Prav tako v agilne uspešne prakse delovanja uvrščamo osebno komunikacijo, samo organizirane skupine, retrospektive postanejo zahtevne, v kolikor je vpleten še DSD.

Težave, ki so posledica DSD in agilnega razvoja so slaba socialna interaktivnost, zamude zaradi različnih časovnih con, pomanjkanje koordinacije, pomanjkanje podpore orodij in infrastrukture, odvisnost med skupinami in težave pri širjenju znanja zaradi razpršenosti skupin na različnih geografskih področjih (Shrivastava & Rathod, 2015, str. 374).

Raziskave so pokazale tudi, da večja kot je geografska razpršenost, večje je tveganje za težave v komunikaciji in koordinaciji, kar pa vodi v nizko uspešnost (Shrivastava & Rathod, 2015, str. 374).

Med glavne težave, s katerimi se sooča izbrano podjetje zaradi razpršenosti, lahko omenim predvsem težave v komunikaciji in koordinaciji. Prav tako zelo velik izziv predstavlja prenos znanja med zaposlenimi, ki se nahajajo na različnih lokacijah. Čeprav angleščina nikomur v podjetju ne predstavlja težav, pa je z vidika komunikacije z drugimi člani ekip lahko včasih problem, saj je včasih določene stvari lažje razložiti ali se dogovoriti v maternem jeziku. In tukaj prihaja tudi do šumov v komunikaciji.

2.2 Zagotavljanje kakovostnega razvoja

Bolj kot izbrano podjetje raste in večje in zahtevnejše stranke pridobiva, bolj se zaveda pomembnosti kvalitetnega produkta, ki ga nudi. Vsem strankam je v današnjih časih kvaliteta popolnoma samoumevna. Turban, McLean in Wetherbe (1999) omenjajo filozofijo, imenovano Total quality management (v nadaljevanju TQM), v kateri se vsi zaposleni v organizaciji zavedajo, da je kvaliteta del njihove odgovornosti.

TQM je filozofija managementa za nenehno zagotavljanje vodstva, treningov in motivacije za izboljšanje organizacijskega managementa in produktno orientiranih procesov za zadovoljevanje internih in eksternih strank. Cilji TQM so delo brez napak, zmanjševanje časa za proizvodnjo, nižji stroški.

Pod aktivnosti zagotavljanja kakovostnega razvoja štejemo vse aktivnosti, ki se izvajajo med procesom razvoja programske opreme z namenom odkritja in odstranitve napak. Cilji zagotavljanja kakovostnega razvoja so povečanje učinkovitosti testiranja, izboljšanje načrtovanja in nadzorovanja aktivnosti zagotavljanja kakovostnega razvoja in izboljšanje celovite kakovosti programske opreme (Elberzhager, Münch & Nha, 2011, str. 2).

Jochen Krebs pravi, da lahko kvaliteto sistema izmerimo glede na število odprtih napak. S kreacijo testov enot razvijalec zagotovi, da bodo testni scenariji obstajali, ko bodo komponente izdane. Avtomatizacija teh testov skozi neprekinjen proces gradnje (ang. *Continuous build process*) zagotavlja, da je sistem s temi testi stestiran z vsako iteracijo. Vsaka neuspešna gradnja (ang. *Build*) tako razkrije eno ali več napak (Krebs, 2009, str. 79).

Izbrano podjetje je po reorganizaciji začelo vlagati trud v izboljšanje kakovosti produkta in tudi same kode, ki je za produkt spisana. Poleg tega, da so v podjetju pridobili več testerjev, ki bedijo nad pravilnim delovanjem produkta z uporabniškega vidika, so se lotili tudi izboljševanja kvalitete kode.

Zavedati so se začeli obsežnega tehničnega dolga, ki se je skozi leta razvoja nakopičil, in se lotili sistematičnega in načrtovanega zmanjševanja tehničnega dolga. Prav tako so uvedli obvezno pravilo, da mora vsak razvijalec, preden potrdi svojo kodo v skupno kodo, pridobiti pozitivno mnenje vsaj enega sodelavca. Za svojo kodo mora razvijalec prav tako spisati različne teste, ki preverjajo delovanje kode.

2.2.1 Tehnični dolg

Tehnični dolg je metafora, ki se uporablja v svetu razvoja programske opreme, in se nanaša na finančne posledice, ki so jih podjetja pripravljena sprejeti za čim hitrejši prehod v produkcijo in prevladajo nad dobro (in pravilno) kodo in dobro zasnovo programske opreme. Tehnični dolg se kopiči skozi vse faze razvoja. Prevelik tehnični dolg lahko pomeni nerentabilnost posameznega projekta in zato morajo imeti podjetja svoj tehnični dolg pod nenehnim nadzorom (Ar.Ampatzogloua, Ap. Ampatzogloua, Chatzigeorgioub &Avgerioua, 2015, str. 53).

Tehničnemu dolgu se ni moč izogniti. Strošek sprememb eksponentno narašča z razvojem sistema. Najlažja pot za zajezitev oziroma za minimalni tehnični dolg je da se v začetnih

fazah ne ubira bližnjic. S tem se poskrbi, da se gradi projekt na dobri in tehnično pravilni osnovi. Agilni razvoj zagovarja, da imajo podjetja med razvojem programske opreme delujoč prototip na katerem se lahko učijo zaposleni na projektu in tudi stranka. S tem se spoznavajo s programsko opremo od samega začetka in lažje definirajo funkcionalnosti. Pomembno vlogo pri tehničnem dolgu igra tudi dokumentacija. Bolj kot je projekt zrel, težje je dokumentacijo pisati za nazaj, torej za vse obstoječe funkcionalnosti, koncept projekta in tehnično dokumentacijo. Krivdo za nastanek tehničnega dolga ni mogoče pripisovati samo razvijalcem, ampak so za to soodgovorni tudi vodje, ki želijo končati projekt s čim nižjimi stroški in čim večjim dobičkom, in stranke, ki imajo željo čim hitreje in čim ceneje dobiti končen produkt (Allman, 2012, str. 4).

Tehnični dolg vpliva na vse vpletene v razvoju programske opreme. Stranke, oziroma končni uporabniki so tisti, ki so zaradi tehničnega dolga najbolj prizadeti. Zaradi tehničnega dolga lahko trpi njihovo delo, v določenih primerih lahko izgubijo tudi svoje poslovne stranke. Prav tako lahko tehnični dolg vpliva na varnost podatkov, ki so vneseni v sistem. Stranke, ki se soočajo z nedelujočo programsko opremo, se obrnejo na službo za pomoč uporabnikom (ang. *Help desk*), ki ima zaradi tehničnega dolga na eni strani nezadovoljne stranke, na drugi strani pa pomanjkanje dokumentacije, slabo delujočo programsko opremo in nimajo direktne komunikacije s skupino ljudi, zadolženo za popravljanje napak.

Razvijalci se delijo na dve vlogi, in sicer na razvijalce, ki so lastniki kode, in na razvijalce, ki so zadolženi za popravljanje obstoječe kode.

Oddelek za marketing ima direkten stik strank in tako občuti njihovo nezadovoljstvo z nedelujočo programsko opremo. Prav tako pa je marketinški oddelek velikokrat soodgovoren za nastanek tehničnega dolga, saj zaradi velikega pritiska strank, pritiska na razvijalce, da dostavijo nove funkcionalnosti in popravke kar se da hitro.

Management mora skrbeti za ugled podjetja in v kolikor se v javnosti pojavi slaba publiciteta podjetja zaradi nedelujoče programske opreme, ki je posledica tehničnega dolga, mora management reševati ugled podjetja (Allman, 2012, str. 7).

2.2.2 Statična analiza

Za izvajanje statične analize ne potrebujemo programske opreme ali izvajanja kode, saj ta preverja dokumentacijo zahtev, arhitekturno zasnovo ali kodo brez izvajanja (Elberzhager, Münch & Nha, 2011, str. 2).

Ghahrai (2016a) dalje razlaga, da se statično analizo izvaja po napisani kodi in pred izvedbo testov enot. Statično analizo se lahko izvaja avtomatsko s programsko opremo ali pa analizo

opravi razvijalec, ki avtorju kode pregleda pravilnost in ustreznost novo napisane kode. S pomočjo statične analize lahko razvijalci dovolj zgodaj odkrijejo napake v kodi in s tem preprečijo marsikatero nadaljnjo napako v programski opremi.

Lahko pa statična analiza pomeni velik časovni zalogaj v kolikor se jo izvaja ročno. Pri avtomatski izvedbi pa obstaja tveganje, da program ne zazna vseh napak v kodi ali pa javlja napačne rezultate.

2.2.3 Dinamična analiza

Dinamična analiza, za razliko od statične analize, zahteva, da se izvajajo programi oziroma deli programov (Elberzhager, Münch & Nha, 2011, str. 2).

Ghahrai (2016a) kot del dinamične analize omenja izvajanje testov enot.

Dinamična analiza se izvaja med samim delovanjem programske opreme in preverja sistemski spomin, obnašanje funkcionalnosti, odzivni čas in splošne zmogljivosti sistema (DuPaul, 2013).

Dinamična analiza lahko odkrije veliko bolj skrite napake in ranljivosti sistema kot statična analiza. Slabost dinamične analize pa je v tem, da lahko odkrije napake oziroma pomanjkljivosti v delu kode, ki se izvaja (DuPaul, 2013).

3 TESTIRANJE PROGRAMSKE OPREME

Testiranje je proces ugotavljanja ali programska oprema deluje tako kot želimo. Proces je lahko zelo dolgotrajen in zahteven. Pri testiranju ni pomembno le to, da napako odkrijemo, ampak tudi da odkrijemo vzrok napake. Pri samem testiranju v večini primerov ne moremo biti prepričani ali smo res odkrili vse napake (Gradišar & Resinovič, str. 216).

Programska oprema je razvita in poslana k stranki ali končnemu uporabniku brez analize obnašanja v vseh možnih scenarijih. Testiranje programske opreme je močno omejeno z določenim proračunom in omejenim časovnim rokom, medtem ko je testiranje zelo obsežno in kompleksno. To vodi v visoko intenzivno proizvodnjo programske opreme, ki je nedokončana, nekonsistentna, polna napak, ki lahko vodijo tudi v varnostne pomanjkljivosti programske opreme (Lina, Hea, Zhanga & Song, 2015, str. 257).

Pogosta zmota o testiranju programske opreme je strošek. Večina razvijalcev pozna samo dve možnosti, nič testov in preveč testov. Iskanje napak v kodi pomeni strošek. V kolikor

podjetje ne testira programske opreme, je strošek za testiranje neobstoječ in se ta strošek prenese v popravljanje najdenih napak s strani stranke oziroma končnih uporabnikov. Če pa podjetje vlaga v testiranje programske opreme pa hitro ugotovi, da strošek testiranja eksponentno narašča bližje, ko je podjetje 100-odstotnemu testiranem produktu (Wells, 2009).

V izbranem podjetju je bilo pred reorganizacijo razvijalcem poznano le zgoraj omenjeno stanje, nič testov. Velik izziv v podjetju je bilo premakniti miselnost razvijalcev, da je to nujni del, ki bi moral obstajati že od samega začetka. Po določenem času so razvijalci začeli razumevati, da testerji prinašajo prednosti, saj je to pomenilo, da se je znanje začelo širiti. Slavchev (2016c) se zaveda, da testiranje ni samo odkrivanje napak v delovanju programske opreme. Da lahko tester najde napako, mora najprej razumeti sistem, pridobiti tako vsebinska kot tudi tehnična znanja o sistemu, dokumentirati sistem. Testiranje je preučevanje tveganj, ki jih predstavljajo napake v sistemu.

Testiranje je ključnega pomena pri zagotavljanju kakovosti programske opreme, obenem pa velja tudi za eno najbolj zahtevnih in dragih aktivnosti med razvojem programske opreme in njenim vzdrževanjem (Burnstein, 2003, str. 6).

Testiranje je ključni element procesa razvoja programske opreme za vodstvo in ocenjevanje kvalitete produkta. Zagotavljanje delovanja celotnega sistema zahteva tudi kompleksno testiranje. Tako celovito testiranje zahteva veliko časa in je tudi precej drago, obenem pa redkokdaj odkrije napake, vendar pa je tako testiranje zavarovalni proces za podjetje, da produkt deluje. Dolgo trajajoče testiranje pa je v konfliktu s strategijo podjetja, ki stremi h krajšim ciklom izdaje verzij programa (Herzig, Greiler, Czerwonka & Murphy, 2015).

Naloga testerja je izpostaviti nepravilnosti v delovanju programske opreme, raziskati nepravilnosti in dokumentirati proces delovanja in odkriti razloge za nepravilnosti, poučiti razvijalce o nepravilnostih in opozoriti uporabnike o nepravilnostih (Slavchev, 2016b).

Testiranje ne daje dodatne denarne vrednosti programski opremi, ampak pomaga omejiti napake v programski opremi, ki lahko močno zaznamujejo kredibilnost ponudnika programske opreme, kot tudi stranke, ki programsko opremo na koncu tudi uporablja. Programska oprema je rešitev za strankine probleme, naloga testerjev pa je zagotoviti, da programska oprema res rešuje strankin problem in ga rešuje na najbolj optimalen način glede na dan čas, sredstva in omejitve, ki jih ima ponudnik rešitve (Slavchev, 2016d).

Burnstein (2003, str 27) omenja 11 načel testiranja:

- testiranje je proces uporabe določenega dela programske opreme s skupkom vnaprej določenih testnih scenarijev z namenom odkrivanja napak in ocenjevanja kvalitete programske opreme;
- kadar je namen testa odkriti napako, potem je dober testni scenarij tisti, ki ima visoko verjetnost, da se bo z njegovo pomočjo napako tudi odkrilo. Pri kreiranju novih testnih scenarijev morajo imeti testerji v mislih cilj, ki ga s posameznim testnim scenarijem želijo doseči. Tester mora s testom dokazati, da napaka obstaja ali pa da del programske opreme deluje. Na podlagi začrtanega cilja pri posameznem testu, mora tester definirati vhodne parametre, končne rezultate in test na koncu tudi izvesti;
- rezultate testov je potrebno podrobno preveriti, saj se lahko zgodi, da se spregleda napako in se test potrdi kot uspešen, čeprav temu ni tako. Možnost obstaja tudi, da tester javi napako, za katero se na koncu izkaže da to ni, in se s tem zgubi precej časa na odkrivanju napake, ki ne obstaja. Prav tako lahko pride do nerazumevanja rezultatov testa, ki pa za odkrivanje pravilnega delovanja, vzamejo veliko časa;
- testni scenarij mora vsebovati pričakovan rezultat. Če je pred izvedbo testa podan pričakovan rezultat, lahko tester vidi ali je odkril napako in če je bil test uspešen, oziroma neuspešen;
- testni scenarij bi moral biti sestavljen tako za veljavne in neveljavne vhodne pogoje. Velikokrat se lahko napake v programski opremi najde tudi s pomočjo vnosa neveljavnih vhodnih pogojev. Lahko se zgodi, da uporabniki ne razumejo povsem delovanja programske opreme, lahko pa se tudi dogodi, da programska oprema proizvede napačne vhodne parametre v drugem sklopu delovanja;
- število še neodkritih napak je proporcionalno številu že najdenih napak v določeni komponenti programske opreme. Napake v kodi se velikokrat pojavljajo v skupinah in so posledica kompleksne in slabo zasnovane kode;
- testiranje bi morala izvajati skupina zaposlenih, ki ni vpletena v skupino razvijalcev. Razvijalci imajo lahko težave pri testiranju lastne kode, saj si težko predstavljajo uporabo programske opreme z uporabniškega vidika. Tukaj se lahko upošteva tudi človeški faktor, pri katerem razvijalci težko priznajo zmote v lastnem delu;
- testi morajo biti ponovljivi in namenjeni večkratni uporabi. To načelo je zelo pomembno, saj mora biti razvijalec zmožen ponoviti vsak test, ko dobi prijavljeno napako. Prav tako mora tester po popravku napake ponovno ponoviti isti test, da lahko ali potrdi pravilnost delovanja ali pa ponovno prijavi napako;
- testiranje bi moralo biti planirano. Testiranje mora biti planirano v projektnem planiranju. Testerji morajo vedeti, kdaj bo nova ali pa popravljena funkcionalnost na voljo za testiranje. Prav tako se s planiranjem testiranja testna skupina lahko pripravi na prihodnje testiranje, si splanira dovolj časa glede na obsežnost funkcionalnosti in si pripravi teste;

- aktivnosti, namenjene testiranju bi morale biti integrirane v življenjski cikel programske opreme. Testiranje se mora začeti že v fazi analize in nadaljevati skozi celotni življenjski cikel, vzporedno z razvojem;
- testiranje je kreativna in zahtevna naloga. Od testerja se pričakuje, da ima obsežno znanje računalniškega programiranja, torej mora razumeti kako je programska oprema specificirana, zasnovana in razvita. Prav tako mora biti tester pozoren na detajle in imeti vsebinsko razumevanje programske opreme. Prav tako mora biti zmožen planiranja, organizacije testiranj in obvladati tudi komunikacijo s strankami. Za učinkovito in uspešno testiranje mora biti tester vedno v koraku s časom na področju novih orodij in tehnologij za testiranje.

Za večino programske opreme velja, da je število potencialnih testnih primerov, ki pokrivajo različne načine obnašanja programa, skoraj neskončno, kar pomeni, da je popolna pokritost s testnimi scenariji nemogoča (Masri & Zaraket, 2016, str. 80).

Testerji prilagajajo merila pokritosti (ang. *Coverage criteria*) in zahteve pokritosti znotraj posamezne zbirke testov, da v celoti pretestirajo določeno funkcionalnost (ang. *Validation testing*), zagotovijo delovanje v preteklosti odkritih napak (regresijsko testiranje) in povečajo verjetnost odkrivanja še ne odkritih napak (ang. *Defect testing*) (Masri & Zaraket, 2016, str. 81).

Testiranje programske opreme je del večje slike razvoja programske opreme in je definirano kot aktivnost za zmanjševanje tveganja, ki je umeščena med razvojem programske opreme ali med vzdrževanjem. Testiranje je sestavni del procesa potrjevanja in se smatra za eno glavnih aktivnosti v razvoju programske opreme. Testiranje ima velik vpliv na kvaliteto končnega izdelka (Opiyo, Horváth & Vergeest, 2002, str. 203).

Cilji testiranja so zagotavljanje delujočih funkcionalnosti, ki jih je naročila stranka, doseganje strankinih pričakovanj in zagotavljanje konsistentnosti in celovitosti delovanja programske opreme. Testi pomagajo razvijalcem pri odkrivanju napak oziroma hroščev na različnih delih in ravneh sistema, pomagajo pri razumevanju uporabniškega vidika in razumevanju delovanja sistema v različnih okoljih (Opiyo, Horváth &Vergeest, 2002, str. 203).

Vsaka napaka v kodi je strošek za podjetje. Dlje kot je napaka skrita, večje je število uporabnikov, na katere lahko ta napaka negativno vpliva in strošek napake se tako le veča. Bližje kot je datum izdaje produkta ali pa verzije produkta, večji je strošek napake. Več časa kot mine med samo kreacijo napake in odkritjem napake, večji je strošek odkrivanja vira napake, kajti večji kot je razpon časa, več nove kode in spremenjene kode je bilo v tem času

producirano in je težje odkriti pravi vzrok napake (Herzig, Greiler, Czerwonka & Murphy, 2015).

3.1 Načini testiranja

Opiyo, Horváth in Vergeest (2002, str. 203) delijo testiranje na klasične teste in na metode pregleda. Pod klasične teste uvrščajo vse teste, ki so del aktivnosti za zmanjševanje tveganj in ocenjujejo celovitost, konsistentnost ali uporabnost specifikacij, oblikovnosti ali kode. Sem se uvršča teste enot, integracijske teste, sistemske teste, teste zahtev (ang. *Requirement tests*), stres teste, teste zanesljivosti (ang. *Reliability tests*), regresijske teste, performančne teste, uporabnostne teste in operativne teste.

Testiranje programske opreme se, predvsem v večjih sistemih, izvaja v treh do štirih nivojih ali fazah testiranja, in sicer gre za testiranje enot, integracijsko testiranje, sistemsko testiranje (ang. *System test*) in potrditveno testiranje (ang. *Acceptance test*). Vsak nivo testiranja lahko vsebuje še podnivoje. Vsak nivo ima določene testne cilje. Cilj testiranja enot je stestirana posamezna enota, znotraj katere testiranje poskuša identificirati funkcionalne in strukturne napake v posamezni enoti. Pri integracijskem testiranju se testira skupek komponent in se raziskuje pravilnost delovanja interakcije med komponentami. Pri sistemskem testiranju se testira celotni sistem. Potrditveno testiranje pa je testiranje strankinih zahtev v programski opremi (Bernstein, 2003, str. 133).

Enota je najmanjša možna komponenta, ki se jo lahko stestira. Ker je komponenta tako majhna in enostavna, je lahko ustvariti testni scenarij, komponento stestirati in analizirati rezultate testa. V kolikor je odkrita napaka, je precej enostavno poiskati napako v kodi in jo popraviti (Bernstein, 2003, str. 138).

Integracijsko testiranje ima dva cilja, in sicer odkrivanje napak, ki se zgodijo na vmesnikih enot in sestavljanje posameznih enot v delujoče podsisteme, kasneje pa v celoten sistem, ki je nared za sistemsko testiranje. Integracijsko testiranje mora zmeraj priti na vrsto po testiranju enot (Bernstein, 2003, str. 153).

Po zaključenem integracijskem testu je na vrsti sistemsko testiranje, ki je zelo kompleksno in obsežno ter zahteva veliko število resursov. Cilj sistemskega testiranja je zagotavljanje delovanja sistema glede na podane zahteve. Sistemsko testiranje testira tako delovanje funkcionalnosti kot tudi kvaliteto sistema, npr. zanesljivost, uporabnost, učinkovitost in varnost sistema (Bernstein, 2003, str. 163). Bernstein (2003, str. 164) dalje deli sistemsko testiranje na:

- funkcionalno testiranje,

- performančno testiranje,
- stresno testiranje,
- konfiguracijsko testiranje,
- testiranje varnosti in
- testiranje ponovne vzpostavitve sistema (ang. *Recovery testing*).

Ko se zaključijo vsa zgoraj našteta testiranja, se programsko opremo oziroma posamezne funkcionalnosti preda v testiranje končnim uporabnikom. Potrditveno testiranje je namenjeno ugotavljanju ali programska oprema zadostuje željam in zahtevam uporabnikov. Uporabnikom mora biti zagotovljena dokumentacija, ki jim je v pomoč pri testiranju in spoznavanju s programsko opremo (Bernstein, 2003, str. 176).

Pregled (ang. *Review*) je procedura zbiranja in analiziranja informacij o kvaliteti koncepta ali končnega izdelka. Pregledi se izvajajo z namenom ugotavljanja ali predlagane oziroma implementirane funkcionalnosti ustrezajo zahtevam stranke in zadostujejo tehnološkim zahtevam. Cilji pregleda so določiti potencialno učinkovitost novih metod in orodij, preučiti delovanje novih produktov in vedenje uporabnikov do njih ter najti možnosti za izboljšavo. Med tehnike pregleda se štejejo analitične metode, vprašalniki, strokovno ocenjevanje, opazovano ocenjevanje in eksperimentalno ocenjevanje (Opiyo, Horvath & Vergeest, 2002, str. 204).

Kempka, McMinn in Sudholt (2015, str. 1) delijo testiranje funkcionalnosti na dva tipa, testiranje z metodo črne škatle (ang. *Black box*) in testiranje z metodo bele škatle (ang. *White box*). Pri metodi črne škatle se tester drži specifikacij, pri metodi bele škatle pa se vhodne parametre pridobiva s pomočjo programske kode. Oba tipa testiranja sta med seboj dopolnjujoča. Metoda črne škatle pregleduje, da je vsa pričakovana funkcionalnost v programski opremi, metoda bele škatle pa testira ali se programska oprema obnaša drugače kot je pričakovano glede na specifikacije.

Pri naključnem testiranju tester izbere naključne vhodne parametre relevantne za področje, ki ga testira. Pri takem načinu testiranja je zmeraj vprašanje ali so naključno izbrani parametri res v dovolj velikem številu in so najbolj primerni, da se uspešno zaključi testiranje določenega sklopa oziroma funkcionalnosti.

Yusufoğlu, Amannejad in Can (2015, str. 125) delijo naloge testiranja na pet različnih tipov:

- načrtovanje testnih scenarijev,
- pisanje testnih skript ali kode za izvedbo testnih scenarijev,
- izvajanje testov,

- ocenjevanje izvedenih testov in
- poročanje rezultatov izvedenih testnih scenarijev razvijalcem.

Vsako izmed zgoraj naštetih nalog se lahko izvaja ročno, avtomatizirano ali pa delno avtomatizirano.

Avtomatizirano testiranje programske opreme uporablja temu namenjeno programsko opremo, s pomočjo katere se izvaja teste in nato tudi analizira testne rezultate (Garousi & Mäntyläc, 2016, str. 92).

Avtomatizirano testiranje in razvoj testne kode (skript) je v zadnjem času postalo velik hit in pomembna dejavnost. Skripte za avtomatizirano testiranje imajo več prednosti, npr. ponovljivost, predvidljivost in učinkovito izvajanje testov. Na drugi strani, pa je sam razvoj skript naporen, nagnjen k napakam in zahteva veliko časovno investicijo (Yusufoğlu, Amannejad & Can, 2015, str. 124).

3.2 Umestitev testiranja v slapovnem modelu

V slapovnem modelu je delo QA skupine postavljeno na konec razvoja. Analitik in razvijalec najprej pregledata zahteve, ki jih je potrebno razviti, v naslednjem koraku razvijalec razvije zahtevano funkcionalnost in jo na koncu šele preda QA skupini v testiranje. QA skupina prediskutira razvito funkcionalnost, pripravi testne scenarije in se nato loti testiranja. Ko (če) tester najde napako, jo javi razvijalcu, ta pa nato takoj popravi napako, napako postavi v čakalno vrsto popravkov ali pa zavrne napako (Shalloway, Beaver & Trott, 2009, str. 3).

Organizacije, ki postavljajo testiranje na konec razvojnega cikla kmalu ugotovijo, da to ne prinaša kvalitetne programske opreme, ki jo lahko dostavijo stranki v dogovorjenih časovnih rokih in znotraj začrtanega proračuna (Burnstein, 2003, str. 253).

3.3 Umestitev testiranja v agilnem razvoju

Vloga testerjev v agilnem razvoju je preprečevanje napak in ne odkrivanje napak. Vloga QA skupine je izboljšava procesov, da v prihodnje ne prihaja do napak in da z najdenimi napakami izboljša procese, ki so povzročili napake (Shalloway, Beaver & Trott, 2009, str. 3).

Postavitev vloge QA skupine na začetek procesa razvoja funkcionalnosti v programski opremi lahko zmanjša oziroma odstrani nepotreben razvoj, zmanjša slabo komunikacijo in zamudo pri dostavo funkcionalnosti stranki (Shalloway, Beaver & Trott, 2009, str. 4).

Ob vsaki novi zahtevi, se mora skupina vprašati, kako vem, da je dosežena želena funkcionalnost? Odgovor na to vprašanje mora biti podan z eksplicitnimi vhodnimi parametri in izhodnimi parametri, ki se jih spremeni v test. Odgovore na to vprašanje mora priskrbeti stranka ali pa jih mora potrditi. Torej, odgovor na to vprašanje postane potrditveni test. Prednost takega načina dela je, da se zmanjša količina usklajevanja vseh članov ekipe, razvijalec ima s tem definiran cilj proti kateremu stremi, v kolikor se lahko test avtomatizira, se ga lahko večkrat izvaja in s tem konstantno preverja ustreznost kode (Shalloway, Beaver & Trott, 2009, str. 4).

Stevenson (2014) zagovarja trditev, da je tester del ekipe in mora v njej sodelovati od začetka projekta. Testerji morajo, poleg testiranja, tudi zagotavljati pomoč in podporo ostalim članom skupine.

V izbranem podjetju ima vsaka skupina vsaj enega testerja, ki je v skupini zadolžen za testiranje. Ker je tester član skupine, pomeni, da se lahko osredotoči na vsebinsko delo, ki ga skupina opravlja. Bolje tako pozna vsebino, lažje najde in opozarja na napake. S poglobljenim vsebinskim znanjem je tester zmožen tudi predvideti določene scenarije in opozoriti na morebitne težave, ki bi se lahko pojavile pri razvoju novih funkcionalnosti.

Crispin in Gregory (2009, str. 15) pravita, da v agilnem razvoju niso samo testerji tisti, ki skrbijo za kvaliteto izdelka, ampak se za to trudijo vsi člani skupine. Cilj agilnega razvoja je izdelava visoko kvalitetne programske opreme v časovnem okviru, ki maksimizira njeno vrednost. To pa je naloga za celotno skupino in ne samo za testerje ali QA skupino.

Agilna skupina mora imeti vse potrebne veščine in znanja, da lahko razvije kvalitetno kodo, za razvoj funkcionalnosti, ki jih zahteva organizacija. To pomeni, da je v skupini vključenih več strokovnjakov z različnih področij. Vsako nalogo je tako sposoben narediti vsak član skupine ali pa več članov skupaj. Tako skupina prevzame vso odgovornost za testiranje, od avtomatiziranega testiranja do ročnega raziskovalnega testiranja (Crispin & Gregory, 2009, str. 15).

3.4 Testna skupina

Primarna naloga članov testne skupine je zagotavljanje učinkovitega in produktivnega testiranja, ki temelji na zagotavljanju kvalitete. Testerji ne smejo biti programerji ali analitiki. Testerji ne popravljajo kode, vendar dodajajo vrednost programski opremi v obliki

višje kvalitete in zadovoljstva strank. Burnstein (2003) opozarja, da je potrebno testerjem zagotavljati izobraževanja in delavnice. Med zadolžitve testne skupine pa navaja:

- razvijanje in apliciranje testnih standardov,
- sodelovanje pri zahtevah novih funkcionalnosti, načrtovanju in pregledu kode,
- testno planiranje,
- načrtovanje testiranja,
- izvajanje testov,
- nadzorovanje testiranja,
- nadzorovanje napak in vodenja evidence ter zgodovine napak,
- iskanje in uporabljanje novih orodij, tehnik in metodologij testiranja,
- nadzorovanje in učenje novih zaposlenih in
- poročila testiranj.

Osebnostne kvalitete in sposobnosti, ki jih mora imeti tester so:

- organizacijske,
- natančnost,
- odločnost,
- delo z ljudmi,
- ustna in pisna komunikacija,
- prilagajanje različnim situacijam ter
- kreativnost.

Tehnične lastnosti, ki jih mora imeti tester pa so:

- splošna razgledanost s področja razvoja programske opreme,
- razumevanje strukture kode,
- razumevanje testnih načel in praks,
- razumevanje testnih strategij, metod in tehnik,
- zmožnost načrtovanja in testiranja na različnih stopnjah,
- poznavanje mrež, baz podatkov, operacijskih sistemov,
- poznavaje upravljanja nastavitev,
- poznavanje dokumentacije s področja testiranja,
- zmožnost zbiranja podatkov in
- zmožnost in želja dela z orodji za pomoč pri testiranju.

Ustanovitev testne skupine v podjetju je velika odločitev, ki mora imeti podporo višjega managementa. Še predno se skupino ustanovi, mora podjetje definirati, kako bo skupina organizirana, takšne so možnosti za napredovanje članov in kako se skupina vključuje v obstoječo organizacijo podjetja (Burnstein, 2003, str. 241).

Tudi v izbranem podjetju je bila odločitev o ustanovitvi testne skupine velika odločitev. Predvsem je bil zelo velik izziv dejstvo, da je bilo potrebno skoraj vse testerje zaposliti. Veliko testerjev se do priključitve skupini še nikoli ni srečalo s testiranjem, še manj pa z vsebino produkta, ki ga izbrano podjetje razvija.

Pri zaposlovanju novih članov testne skupine se podjetje ni moglo zanašati na to, da bi lahko zaposlilo osebe, ki imajo veliko vsebinskega znanja, zato so se v podjetju osredotočili na lastnosti kandidatov, ki jih omenja tudi Baiju (2007), in sicer lastnost analitičnega in logičnega razmišljanja, razumevanja poslovnih zahtev in procesov, radovednost, katera mu omogoča nenehno izobraževanje in izboljševanje, zmožnost kritičnega razmišljanja, zmožnost časovnega planiranja in komunikacijske veščine.

Burnstein (2003, str. 245) pravi, da je testna skupina velik strošek za podjetje in zahteva močno zavezanost obstoječi strategiji podjetja. Vendar pa se mora podjetje zavedati, da je testna skupina obvezni del podjetja, ki razvija kompleksno programsko opremo in ima s tem močan vpliv na družbo. Testna skupina ima odgovornost in motivacijo, da:

- načrtuje testiranje,
- nadzira in meri testiranje, da je narejeno pravočasno in znotraj začrtanega proračuna,
- meri proces in atribute produkta,
- vodstvu dostavlja neodvisne informacije o stanju produkta in procesov,
- načrtuje in izvaja teste brez nepotrebnega truda,
- avtomatizira teste,
- sodeluje v pregledih in s tem zagotavlja kvaliteto,
- sodeluje z analitiki, načrtovalci, razvijalci in strankami in s tem zagotavlja doseganje kvalitetnega produkta,
- vzdržuje prostor namenjen testnim informacijam ter
- podpira izboljševanje procesov.

Crispin in Gregory (2009, str. 22) navajata glavna načela, ki se jih mora vsak agilni tester držati:

- zagotavlja nenehno povratno informacijo,
- dostavlja vrednost strankam,

- ima osebni pristop in komunikacijo z vsemi udeleženci na projektu,
- pogum,
- ne komplicira (ang. *Keep it simple*),
- skrbi za nenehno izboljševanje,
- odziva se na spremembe,
- samo-organiziran,
- osredotoča se na ljudi in
- uživa.

Testna skupina bi morala biti sestavljena iz testnega managerja, testnega vodje, testnega inženirja in testnega inženirja začetnika.

Testni manager skrbi za vse vidike testiranja in zagotavljanja kakovosti. Odgovoren je za politiko testiranja, izobraževanje, testno dokumentacijo, nadzor testiranja, pregled testiranja, kadrovanje testne skupine. Testni manager tesno sodeluje z višjim managementom podjetja in projektnimi vodjami.

Testni vodja pomaga testnemu managerju in sodeluje na projektih s preostalimi člani skupine. Testni vodja je zadolžen za načrtovanje testiranja, nadzor članov in poročanje o delu testne skupine. Testni vodja sodeluje pri razvoju testov, njihovem izvajanju, pregledu in komunikaciji s stranko.

Testni inženir načrtuje, razvija in izvaja teste, postavlja testna okolja, sodeluje pri načrtovanju testov.

Testni inženir začetnik je oseba, ki je na novo zaposlena in šele pridobiva znanje s sodelovanjem pri načrtovanju testov in izvajanju testov (Burnstein, 2003, str. 246).

4 ANALIZA STANJA V IZBRANEM PODJETJU

Analizirano podjetje je uspešno mednarodno podjetje na področju informacijske tehnologije, ki razvija lastno programsko opremo in je prisotno v več državah sveta. Zaradi nenehno spreminjajoče se dinamike poslovanja in različnega nabora strank, je podjetje primorano iskati načine za čim učinkovitejši razvoj in organiziranost.

Zaradi anonimnosti podjetja, bo podjetje v nadaljevanju naslovljeno kot izbrano podjetje. Tako kot ostala podjetja v vseh panogah, mora tudi izbrano podjetje ostati konkurenčno oziroma stremeti k prehitevanju konkurence. Temu primerno je podjetje v zadnjih letih prešlo več reorganizacij, tako manjših kot tudi eno večjo.

Izbrano podjetje sledi agilnemu načinu razvoja, predvsem Scrumu. Prav s prehodom na Scrum so se pričele manjše reorganizacije zaposlenih in skupin. Izbrano podjetje je že pred tem imelo vpeljan proces nenehne gradnje, vendar je v procesu reorganiziranja zamenjalo orodje za proces.

4.1 Opis podjetja za razvoj programske opreme

Podjetje se na vsakodnevni ravni sooča z več različnimi izzivi:

- raznolikost strank,
- delovanje v različnih državah,
- nenehna rast števila zaposlenih,
- fluktuacija zaposlenih,
- dislocirane pisarne,
- jezikovne prepreke,
- razvoj in izboljšava produkta ter
- učinkovit in uspešen razvoj.

Podjetje je pred leti začelo v svoje delovne procese vpeljevati agilni način razvoja programske opreme. Potreba se je pojavila v obdobju, ko je podjetje za svoj produkt začelo pridobivati vedno več strank, ki pa so se med seboj zelo razlikovale in prihajale tudi iz različnih okolij.

Stari način dela ni več zadoščal vsem novih potrebam in izzivom, zato se je podjetje usmerilo v agilni način razvoja programske opreme.

Programska oprema, ki jo podjetje razvija in trži, je sestavljena iz več različnih vsebinskih sklopov. Prvotna organizacija je bila namenjena samo eni stranki in naravna razdelitev razvijalcev vključenih na projekt je bila po posameznih vsebinskih sklopih. Tako je vsak razvijalec dobro poznal vsebinski del, ki ga je pokrival.

Ko se je nato projektu priključila nova stranka, so se že začeli nakazovati prvi problemi z obstoječo organizacijo. Težava je bila pri postavljanju prioritet dela in nenehnem menjavanju razvojnega okolja posameznega razvijalca.

Prvotna stranka je produkt uporabljala v produkciji, poleg razvoja novih funkcionalnosti se je s prehodom v produkcijo začela podpora končnim uporabnikom na vsebinskem področju in tudi na področju reševanja napak programa. S priključitvijo nove stranke, pa so se pojavili

pritiski s strani posamezne stranke in po prioritetnem reševanju njihovih težav, razvoju novih funkcionalnosti za prvotno stranko in funkcionalnosti za novo stranko. Vsaka stranka je imela svojega projektnega vodjo, ki so dajali zadolžitve razvijalcem. Vsak projektni vodja je do razvijalca pristopil s pristopom, da ima njegova naloga najvišjo prioriteto, to pa je velikokrat vodilo do frustracij s strani razvijalcev, ker niso vedeli katera naloga in kateri projektni vodja ima večjo prioriteto.

Poleg organizacijskih težav so na plano začele prihajati tudi tehnične težave, saj sta si obe stranki delili enako kodo, kar je pomenilo, da je prihajalo pri eni in drugi stranki do nedelovanja že obstoječih in delujočih funkcionalnosti, ker se je za potrebe ene stranke razvila nova zadeva ali pa popravila obstoječa, kar pa je pokvarilo delovanje drugi stranki.

S prihodom novih in novih strank, so se ti problemi le še povečevali in povzročali preglavice vsem vpletenim v sam razvoj.

Zaradi povečanega obsega dela, se je zaposlovanje v podjetju močno povečalo. S prihodom novih zaposlenih se je pojavil dodatni problem, in sicer prenos znanja iz izkušenih razvijalcev na nove zaposlene. Ob manjšem številu zaposlenih na prvotnem projektu, sam prenos znanja nikoli ni povzročal težav. Ko pa se je projekt širil, in so obstoječi zaposleni bili pod vedno večjimi pritiski in vedno bolj obremenjeni z delom, je širjenje znanja postal velik problem. Ker gre pri dotičnem programu za zelo specifičen produkt, je tudi pridobivanje znanja zelo oteženo. Nepoznavanje samega vsebinskega dela se je odražalo tudi pri kvaliteti kode in posledično razvitih funkcionalnostih, ki so bile pomanjkljivo ali narobe razvite in polne hroščev (ang. *Bug*).

Prihod dodatnih tujih strank je zahteval tudi lokalno prisotnost in podjetje je začelo odpirati pisarne tudi v tujih državah. Dislociranost skupine je delo le še otežila. Poleg dislociranosti in dela na daljavo so se težave pokazale tudi v jezikovnih preprekah in kulturnih razlikah.

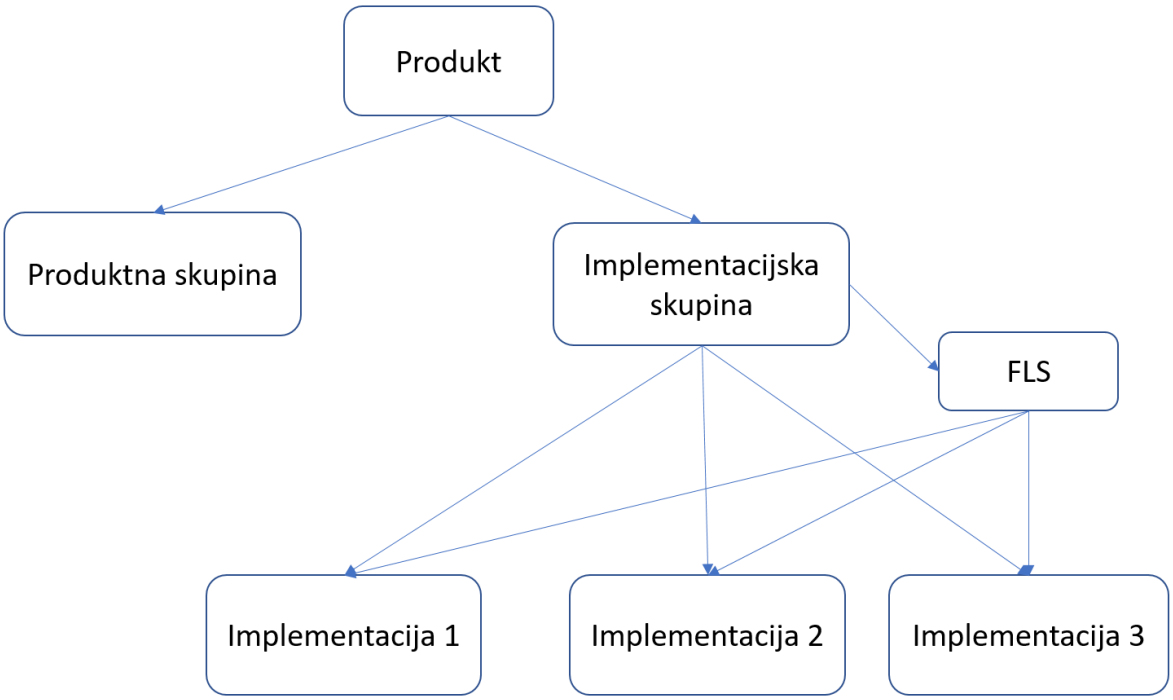
Zaradi vedno večjega števila organizacijskih težav, je podjetje začelo vpeljevati agilni način razvoja. Temu je sledilo več manjših reorganizacij. Glavna reorganizacija v podjetju pa se je zgodila leta 2015, ki je prinesla precejšnje organizacijske spremembe.

4.2 Organizacija razvojnih skupin

Podjetje je organiziralo zaposlene v dve glavni skupini, produkt in implementacija. Produktna skupina se dalje deli na več manjših skupin, ki se delijo glede na vsebinsko področje, arhitekturno in infrastrukturno skupino. Implementacija pa je tudi razdeljena na več skupin, ki pa se delijo glede na posamezno stranko. Poleg tega pa se je ustanovila še testna skupina, tako imenovana FLS skupina.

Vse skupine uporabljajo orodje Jira, ki je namenjeno projektnemu vodenju. Jira je v podjetju edino orodje s katerim vodijo zahteve za nove funkcionalnosti, prijavo novih napak, napredek reševanja posameznih zahtevkov, vnos porabljenih ur vsakega posameznega zaposlenega. Samo orodje je prilagojeno tudi Scrum načinu in s pomočjo Jire se lahko sledi napredku vsakega posameznega sprinta.

Slika 5: Organizacija razvojnih skupin



Vir: lastno delo.

4.2.1 Implementacijske skupine

Posamezna implementacijska skupina je samozadostna, sestavljena je iz projektnega vodje, skupine razvijalcev, ki pokrivajo vse vsebinske sklope in vsaj enega testerja.

Obstajajo tri možne faze projekta, v katerih se posamezna implementacija nahaja:

- faza razvoja,
- faza razvoja in delna vzdrževalna faza ter
- vzdrževalna faza.

V razvojni fazi razvijalci posamezne implementacije razvijajo nove zahteve in vrzeli, ki so bile odkrite med analizo. Tester skrbi za sprotno testiranje, pisanje testnih scenarijev in uporabniških navodil.

Prehod v vzdrževalno fazo je zelo zahteven in tvegan za vsako stranko. Zato je v večini primerov prehod v vzdrževalno fazo postopen. To pomeni, da stranka začne v produkciji uporabljati le določene, vnaprej dogovorjene funkcionalnosti ali sklope. V tej fazi mora implementacijska skupina skrbeti tako za podporo stranki za funkcionalnosti v produkciji, kot tudi dalje razvijati nove funkcionalnosti in vrzeli kot v razvojni fazi. Z vsakodnevno uporabo programske opreme stranka tudi šele dobi občutek ali nova programska oprema zadošča potrebam in pričakovanjem in v tej fazi se pojavi tudi veliko novih zahtev, ki jih je potrebno umestiti v plan izvedbe.

Zadnja faza je celotna vzdrževalna faza, v kateri stranka pri vsakodnevnem poslovanju uporablja programsko opremo z vsemi sklopi in dogovorjenimi funkcionalnostmi. V tej fazi celotna implementacijska skupina sodeluje v podpori stranki in pri odpravljanju odkritih napak. Prav tako se poslovni procesi strank skozi čas spreminjajo in v skladu s tem stranka želi spremembe tudi v programski opremi in poda zahtevek za implementacijo sprememb in novih funkcionalnosti.

V obeh vzdrževalnih fazah vsi člani skupine sodelujejo s stranko zaradi lažje in hitrejše komunikacije.

Krmiljenje med popravljanjem obstoječih napak in razvojem novih funkcionalnosti se je pojavilo kot velik izziv že v času pred reorganizacijo in se je dalje obdržal tudi pri vsaki posamezni implementacijski skupini.

Implementacijske skupine sledijo Scrum načinu dela in v vsakem sprintu rezervirajo določeno število dni, oziroma ur tudi za podporo stranki z novo prijavljenimi napakami.

Pred reorganizacijo dela je vsak razvijalec bil del posameznega vsebinskega sklopa, kar pomeni, da je vsebinsko bolje poznal en sklop, ostale pa precej slabo, oziroma jih sploh ni. Po reorganizaciji pa so vsi razvijalci v skupini bili primorani začeti delati tudi z vsebinskimi sklopi, s katerimi do sedaj niso imeli ne stika in ne izkušenj.

Težava je bila, ker so že vpeljani razvijalci na začetku porabili veliko več časa od predvidenega za dokončanje posameznih nalog. Prav tako pa so rabili veliko vsebinske pomoči od sodelavcev. Obenem pa so bili tudi sami tisti, ki so morali drugim nuditi pomoč.

Vsi razvijalci so se lahko zanašali le na pomoč sodelavcev in njihovih dosedanjih izkušenj, saj stanje programske opreme, tako vsebinsko kot tudi tehnično ni bilo dokumentirano. Čeprav vsi delajo na istem produktu, so posamezne implementacije v različnih razvojih fazah, projekti različno dolgo trajajo, prav tako pa ima vsaka implementacija drugačno verzijo produkta, na kateri je začela graditi. Tudi zahteve posameznih strank se med seboj precej razlikujejo. Čeprav več strank vpeljuje enak poslovni proces, pa se posamezne aktivnosti znotraj procesa tako razlikujejo, da je potrebno vse prilagoditi vsaki posamezni stranki.

Zaradi vseh teh razlik v različnih programskih verzijah in pa različnega pogleda na določene funkcionalnosti, ki so bile po različnih implementacijah tudi različno implementirane, je prihajalo do vsebinskih neskladij med zaposlenimi. Prav tako je redko kateri razvijalec popolnoma razumel in poznal določeno funkcionalnost. Zaradi tega je prihajalo do napačnih informacij ali pa celo nasprotujočih si mnenj.

Zaposleni, ki je iskal informacije, je tako težko zaupal prejetim informacijam in jih je moral preveriti na vsaj še dveh mestih. To je prinašalo dodatno izgubo časa vseh vpletenih.

V podjetju je bila peščica zaposlenih, ki je dobro poznala posamezno vsebinsko področje. Po reorganizaciji pa so ti zaposleni dobili različne vloge in zadolžitve in zato so njihovo dosedanje delo prevzeli drugi. Težava se je pojavila, ker so poznavalci področij postali težje dostopni ostalim za pomoč in za predajo znanja. Tudi to je bil eden izmed razlogov za upočasnitev dela in večje število napak, ki so se na začetku pojavljale.

Zaposleni so bili po starem načinu dela navajeni, da so bili del vsebinskega sklopa za vse stranke naenkrat in so tako vedeli, kaj točno se razvija za katero izmed strank. Po reorganizaciji pa so imeli kontakt samo s svojo skupino. Pojavile so se težave, oziroma podvojeno delo, ker implementacijske skupine med seboj niso komunicirale in so na primer eno zahtevo implementirali dvakrat za dve različni stranki, čeprav bi lahko skupina samo vzela obstoječo rešitev in si s tem prihranila veliko časa.

Implementacijske skupine imajo tedenski sestanek, kjer vsak vodja skupine pojasni stanje na posameznem projektu. Na tem sestanku prisostvuje tudi vodja FLS skupine, ki s pomočjo vodij implementacijskih skupin poskuša koordinirati delo članov FLS skupine.

Čeprav se projektni vodje vsak teden sestanejo na sestanku, pa to vseeno včasih ni dovolj, saj imata vodji vseh implementacijskih skupin preveč dela tudi z drugimi aktivnostmi in se ne moreta dovolj kvalitetno posvečati napredku in težavam posameznih implementacijskih skupin.

4.2.2 Testna skupina

FLS skupino trenutno sestavlja osem članov. Vsak član pomaga eni izmed implementacijskih skupin, kjer skrbi za testiranje novih dodelav, kot tudi popravkov. V tisti implementacijski skupini, kjer pa se produkt že uporablja v produkciji, pa FLS član skrbi tudi za direktno podporo uporabnikom, komunikacijo s strankami, testiranjem prijavljenih nepravilnosti.

Pred reorganizacijo podjetja vloga testerja ni obstajala. Z ustanovitvijo FLS skupine se je pojavilo s strani obstoječih zaposlenih precej negodovanja in nezaupanja v novo osebo na projektu, ki nima izkušenj s programiranjem in tudi ne bo programirala.

Zaposleni so bili do reorganizacije zelo obremenjeni in vsak nov član skupine je na začetku predstavljal novo breme, saj je bilo potrebno novega člana uvesti. Dejstvo, da je novi član razvijalec, je bilo veliko lažje sprejeto, kot pa dejstvo, da je novi član tester. Do sedaj nihče od zaposlenih ni imel izkušenj s sodelovanjem na projektu, kjer je sodeloval tudi tester, in redko kdo je v tem videl prednost.

Na začetku so novega testerja v skupino težko sprejeli, osebi se niso posvečali in so ji nudili zelo malo podpore pri deljenju vsebinskega znanja. Poleg začetnega odklonilnega odnosa sodelavcev, je bil velik izziv tudi izobraziti nove člane.

Na samem projektu je bilo hudo pomanjkanje kakršnekoli dokumentacije projekta, tako tehnične, kot tudi vsebinske. Člani FLS skupine so bili večinoma vsi novo zaposleni brez vsebinskega znanja, potrebnega na projektu in skoraj vsi so bili tudi brez izkušenj na področju testiranja.

FLS skupina se trenutno za testiranje ne poslužuje nobenega orodja za organizacijo testiranja. Sam način testiranja se razlikuje od implementacije do implementacije in ni poenotenega procesa testiranja.

Prav tako se zaradi poslovnih odločitev in pomanjkanja časa na enem izmed projektov v produkciji testiranje ne izvaja. Tester na tem projektu nastopa le v vlogi podpore uporabnikom in reprodukcije javljenih napak s strani končnih uporabnikov.

Vsaka implementacijska skupina ima trenutno zelo kratek seznam testnih scenarijev, ki so jih testerji pripravili, vendar obstoječi testni scenariji ne pokrijejo vseh funkcionalnosti. Razlog za pomanjkanje testnih scenarijev je tako v pomanjkanju časa testerjev, da bi testne scenarije pripravili kot tudi pomanjkanju volje projektnih vodij, da testerjem zagotovijo čas, namenjen pripravi testnih scenarijev.

Na drugi strani tudi stranke same testirajo verzijo programa in kreirajo lastne testne scenarije. Ker lahko prihaja do razlik med testnimi rezultati testerja podjetja in testerji stranke, testerji občasno izdelujejo testna poročila, v obliki slik korakov in rezultatov ali pa v obliki kratkih posnetkov, ki snemajo korake.

FLS skupina ima sestanke na tedenskem nivoju, kjer vsak član pove, s čim se je v preteklem tednu ukvarjal, kakšni so plani na projektu za sledeči teden in ali so se na določenem projektu pojavile težave, npr. preobremenjenost posameznega testerja.

FLS skupina izvaja le ročno testiranje. Avtomatsko testiranje se pojavlja le v obliki testov enot in integracijskih testov, ki pa jih pišejo in posodabljajo razvijalci. Počasi se uvaja tudi avtomatizacija testnih scenarijev, ki simulirajo ročno izvedbo testnih scenarijev.

Ob ustanovitvi FLS skupine je bil vsak član določen, da se pridruži eni izmed implementacijskih skupin. Ker je tester moral na svoji implementaciji pridobiti veliko novega, predvsem vsebinskega znanja, se prenos znanja med člani ni nikoli naredil. Sedaj, v primeru, da tester zboli ali pa gre na dopust, nihče drug v skupini ne more začasno prevzeti nalog testerja, saj nima dovolj poglobljenega vsebinskega znanja posamezne implementacije.

Kadar je posamezen član odsoten, delo na njegovi implementaciji stoji in se s testiranjem čaka, da se tester vrne v službo. Če pride do situacije, ko je potrebno nujno pretestirati določeno dodelavo ali popravek, to naredijo ostali člani implementacijske skupine.

Na začetku je vsak član potreboval več časa za testiranje posameznih zahtevkov in za samo razumevanje zahtev. Z vedno več učenja in testiranja, se je ta čas zmanjšal, povečala se je tudi produktivnost testerja in pa zmožnost odkrivanja napak. S tem se je zmanjšalo število napak, ki jih je nato stranka sama odkrila s testiranjem in povečalo število napak, najdenih z internim testiranjem.

Prav tako testerji na projektih, ki so v produkciji, zelo velik delež javljenih napak rešijo sami in s tem razbremenijo razvijalce, ki se jim ni potrebno več ukvarjati neposredno z uporabniki in se lahko posvetijo zahtevnejšim nalogam. Napake, ki pa jih dobijo v reševanje, pa tester predhodno že sam analizira in pretestira, da razvijalcu pripravi kar se da veliko podatkov za lažje reševanje napak. S tem se izboljša tudi sama produktivnost razvijalcev, saj ne namenjajo več veliko časa sami vsebinski analizi in pripravi okolja, kjer lahko napako tudi ponovijo.

FLS skupina ima skupen elektronski naslov, do katerega imajo dostop vsi člani skupine. Prvotno je bilo zamišljeno, da bo ta elektronski naslov enotna vstopna točka za javljanje napak s strani stranke. Ideja živi samo na eni izmed implementacij, medtem ko na drugih do uporabe tega elektronskega naslova ni prišlo zaradi nesodelovanja projektnih vodij pri sami ideji.

Samo delo članov FLS ekipe je tako precej nenadzorovano s strani vodje, prav tako pa je obveščanje članov FLS med seboj omejeno in skopo z informacijami.

Pri vsaki implementacijski skupini trenutno deluje je le po en član FLS skupine. Potreb po povečanju članov trenutno ni. V kolikor se zgodi, da ima en član preveč dela, za katerega ve, da ga ne more narediti pravočasno in kvalitetno, na to opozori projektnega vodjo in vodjo FLS skupine. Oba nato poskušata najti člana FLS skupine, ki lahko priskoči na pomoč pri testiranju.

4.3 Življenjski cikel razvoja programske opreme

Ob začetku novega projekta, oziroma ob pridobitvi nove stranke, se najprej začne analiza, ki jo pokrivajo poslovni analitiki podjetja. Vsaj en analitik pokriva eno vsebinsko področje. Po končani analizi vrzeli (ang. *Gap analysis*), analitik vse nove zahteve vnese v program Jira. Dalje sledi pregled zahtevkov s strani arhitektov in posameznih lastnikov domenskega področja. Ti odločijo ali je zahtevana sprememba del osnovnega produkta ali je dodelava specifična za implementacijo.

Proces je od tu dalje enak za obe skupini, produktno in implementacijsko. Posamezno zahtevo oziroma dodelavo se analizira, razbije na več manjših smiselnih zahtevkov. Zahtevke se doda v seznam zahtev izdelka in ko pride zahtevek na vrsto za samo realizacijo znotraj sprinta, se zahtevek dodeli skupini, ki bo zahtevek realizirala. Skupina nato zahtevek analizira na sestanku za grobo ocenitev nalog (ang. *Grooming meeting*). Zahtevek je nato dodeljen posameznemu razvijalcu, ki zahtevek tudi dokonča znotraj enega ali več sprintov.

S pričetkom razvoja, ali pa tudi kasneje, ko je projekt že v produkciji, dobijo stranke dostop do orodja Jira in tudi same vnašajo zahtevke za razvoj novih funkcionalnosti ali pa popravkov obstoječih funkcionalnosti. Projektni vodja odloči ali je zahtevek upravičen ali ne. Če zahteva ni upravičena, je to potrebno stranki pojasniti in skupaj najti zadovoljivo alternativo. Ob upravičeni zahtevi pa se ponovi zgornji proces dela, zahtevek je potrebno najprej umestiti ali k produktni skupini ali pa k implementaciji in jo nato dodati v seznam zahtev izdelka. Pred pričetkom sprinta se nato izbere zahtevek iz seznama in se nadaljuje z realizacijo zahtevka.

Ko razvijalec zaključi s programiranjem kode, napiše teste enot, stestira dokončano funkcionalnost ali popravek pri sebi lokalno. Šele ko je zadovoljen s funkcionalnostjo, jo potrdi v skupno kodo (ang. *Code commit*).

Ko je koda potrjena, se preko strežnika Teamcity zažene nova programska verzija. Teamcity je strežnik, ki omogoča zagon testov enot, avtomatskih testov in združevanja nove kode z obstoječo kodo. V kolikor ne pride do napak, se kreira nova testna programska verzija, z vsemi novimi popravki in dodelavami.

Tester obvezno pretestira nove dodelave in popravke s povsem uporabniškega vidika. Poleg novosti tester pretestira tudi že obstoječe funkcionalnosti, saj je pomembno, da nove dodelave ne pokvarijo delovanja obstoječih funkcionalnosti.

Ko tester potrdi ustreznost delovanja verzije programa, le-to namestijo k stranki, ki nato tudi sama izvede testiranja.

Stranka izvede lastna testiranja in ob potrditvi verzije, se le-to nato namesti na produkcijsko okolje ali pa nadaljuje z razvojem.

V primeru, da se napake pojavijo že na Teamcityju, je gradnja verzije označena kot neuspešna in o tem je obveščena celotna skupina. Razvijalec, ki je zakrivil napako, jo mora nato čimprej odstraniti in ponovno potrditi.

V kolikor napako odkrijeta tester ali pa stranka, zahtevek, na podlagi katerega je bilo testiranje izvedeno, ponovno odpreta in o tem obvestita razvijalca. Razvijalec se nato vrne nazaj v točko pregleda kode in popravljanja kode.

Ko se prične razvoj za novo stranko, implementacijska skupina vzame zadnjo najbolj stabilno verzijo produkta in začne graditi na njej. Od tam dalje implementacijska skupina sama razvija funkcionalnosti za stranko. Zaradi specifike programske opreme in pa specifike posamezne stranke, je zelo težko narediti funkcionalnost, ki bi bila lahko splošna za več različnih strank. Zato mora posamezna implementacijska skupina vložiti precej časa v spreminjanje in nadgrajevanje obstoječih funkcionalnosti do te mere, da zadosti potrebam stranke.

Vizija podjetja je, da bi lahko implementacijske skupine prevzemale nove verzije produkta brez večjih zapletov. Do sedaj se je to izkazalo za neuresničljivo, saj do sedaj še nobena implementacijska skupina ni nadgradila na novo verzijo produkta. Zadržki se pojavljajo predvsem z vidika prevelikega odstopanja med obstoječo verzijo implementacije in pa

najnovejšo verzijo produkta, ki trenutno prinaša zelo veliko sprememb. To pa pomeni velik finančni zalogaj, ki pa je na strani podjetja in ne stranke.

4.4 Ugotovitve obstoječega stanja v podjetju

Podjetje je bilo primorano iti v reorganizacijo projekta in po več kot letu dni se kažejo pozitivni znaki, da je bila reorganizacija prava pot za rast podjetja.

Velik izziv v podjetju predstavlja vpeljava novih zaposlenih in prenos znanja na nove zaposlene, predvsem iz vsebinskega vidika projekta.

Število implementacijskih skupin vztrajno narašča, s tem pa se povečuje tudi tveganje nepotrebnega podvajanja dela. Več kot je implementacijskih skupin, večjo težavo predstavlja komunikacija in koordinacija med posameznimi implementacijskimi skupinami, predvsem zato ker so skupine razpršene po več državah.

Prav zaradi razpršenosti zaposlenih, skupin in raznolikosti strank, je dobro, da podjetje uporablja enoten sistem za vodenje zahtevkov za nove funkcionalnosti in prijavo napak, Jiro. S tem obdržijo sled implementacije posameznega zahtevka, zgodovino sprememb.

Medtem ko se komunikacija med implementacijskimi skupinami izboljšuje, povečuje medsebojno sodelovanje, pa obstaja težava v komunikaciji med implementacijskimi skupinami in produktno skupino. Komunikacije je zelo malo, prav tako implementacijske skupine nimajo vpogleda v načrte produktne skupine in ne vedo kako produktna skupina razvija nove funkcionalnosti in izboljšuje obstoječe funkcionalnosti.

Še zmeraj obstajajo preveliki zadržki za prevzemanje novejših verzij produkta, ki bi nudile strankam najnovejše funkcionalnosti. Zadržki se pojavljajo predvsem zato, ker so med verzijami programa, ki jih uporabljajo implementacijske skupine in verzijo, ki jo razvija produktna skupina nastale tako velike vrzeli, da bi posamezne implementacijske skupine potrebovale več mesecev, da bi novo verzijo programa prilagodile svojim strankam.

5 PREDLOGI ZA IZBOLJŠAVE IN ANALIZA PO UVEDBI SPREMEMB

Svetovni trend v majhnih in srednje velikih podjetjih narekuje, da podjetja oblikujejo t.i. hibridno strukturo, kjer so testerji integrirani v posamezne projektne skupine in tako tesneje sodelujejo s projektno skupino. Potreba po taki integraciji se je pojavila z razlogom, da

imajo testerji tesnejši stik s poslovnimi eksperti, ampak tudi vpeljava agilnega načina dela zahteva, da so testerji bližje skupini, s katero sodelujejo (Buenen & Teje, 2014, str. 7).

5.1 Uvedba podpornih orodij za organizacijo testiranja

Podjetje trenutno nima nobenega orodja za organizacijo testiranja, zato predlagam uvedbo le-tega. Kot prednosti uvedbe takega orodja Ramadoss (2015) omenja:

- orodje za organizacijo testiranja ima enotno lokacijo, kjer so shranjeni vsi testni scenariji, vse izvedbe testnih scenarijev in vse posodobitve testnih scenarijev. S tem so vsi sodelujoči vedno na tekočem z zadnjim aktualnim stanjem,
- orodje za organizacijo testiranja zmanjšuje verjetnost ponavljajočega dela ter
- povečuje produktivnost v kratkem časovnem roku.

Organizacija testiranja je organizacija testnih scenarijev. Orodje za organizacijo testiranja pa je orodje, s pomočjo katerega organiziramo testiranje in testne scenarije. Z orodjem lahko organiziramo tako ročno kot avtomatsko testiranje. Orodje uporablja QA skupina (Kumar, 2012).

Pri ročnem testiranju tester ročno izvaja testiranje in sledi korakom, navedenim v testnem scenariju, ki je zaveden v orodju za organizacijo testiranja. Ko pride do odstopanja od pričakovanega rezultata, tester označi test kot neuspešen, obenem pa tudi označi korak, v katerem je prišlo do napake (Kumar, 2012).

Pri avtomatskem testiranju je lahko orodje za organizacijo testiranja integrirano z orodjem za avtomatsko testiranje, ki si med seboj sporočata, kakšni so testni scenariji in kakšni so rezultati izvedbe posameznega testa (Kumar, 2012).

Podjetje že dolgo časa uporablja orodje Jira za vodenje zahtevkov, zato je potrebno pri uvedbi orodja za organizacijo testiranja iskati orodje, ki bo podpiralo integracijo z Jiro, saj to pomeni zmanjšanje časa, potrebnega za prijavo napak, povezljivost testnih scenarijev in izvedb testiranj s posameznimi zahtevki.

Za namene magistrske naloge in tudi samega izboljšanja delovanja FLS skupine, sem pregledala ponudbo obstoječih orodij za organizacijo testiranja, ki se pojavljajo na trgu.

Podjetje mora povečati obseg testiranja zato, da zagotovi funkcionalen produkt, preden ga dostavi stranki. Ker pa je obsežno testiranje lahko dolgotrajno in je v nasprotju s filozofijo podjetja, ki sledi agilnemu principu dela, mora vsaka skupina za testiranje izbrati ustrezne

testne scenarije za testiranje verzije. Vendar pa na račun zmanjšanja števila izvedenih testov ne sme trpeti kvaliteta produkta (Herzig, Greiler, Czerwonka & Murphy, 2015).

Da pa bi podporno orodje res bilo koristno in v pomoč testni skupini in s tem tudi posameznim implementacijskim skupinam, mora najprej FLS skupina pripraviti bazo testnih scenarijev.

Vsaka implementacija ima svoje specifične funkcionalnosti in tudi splošno poslovanje posamezne stranke se razlikuje od ostalih. Zato je vsaka programska oprema do potankosti prilagojena posamezni stranki. Vsaka implementacija tako potrebuje svoj nabor testnih scenarijev. Še zmeraj obstajajo določeni testnih scenariji, ki jih tester lahko uporabi na vsaki implementaciji.

Kot osnova za posamezno implementacijo morajo biti splošni testni scenariji za vse implementacije, na katerih nato posamezna implementacija gradi svojo bazo scenarijev. Cilj baze scenarijev je, da v danem trenutku pokrijejo testiranje vseh obstoječih funkcionalnosti in lahko katerikoli član skupine s pomočjo napisanih testnih scenarijev pretestira programsko opremo.

Proces izdelave testnih scenarijev se začne že v fazi analize vsakega projekta, kjer poslovni analitik v analizi napiše osnovne testne scenarije pokritih poslovnih procesov in testne scenarije za odkrite vrzeli (ang. *Gap*). Ko skupina pred implementacijo posamezne vrzeli sestavi celotno uporabniško zgodbo in jo definira in razčleni detajlno, je naloga testerja in poslovnega analitika, da dopolnita testne scenarije. Ti scenariji so nato osnova za testiranje razvite funkcionalnosti, tako ročno testiranje kot tudi avtomatsko testiranje.

Z vsako novo razvito funkcionalnostjo se poveča tudi baza testnih scenarijev. Za lažji nadzor te baze, bi podjetje moralo razmisliti o uvedbi podpornega orodja za organizacijo testiranja. Ker podjetje za namene vodenja razvoja in popravljanja programskih napak že uporablja orodje Jira, je to tudi dobro izhodišče tudi za namene organizacije testiranja.

Pri izbiri načina testnega upravljanja, se lahko podjetje odloči za eno izmed treh možnosti:

- uporabi orodje Jira tudi kot orodje za testno upravljanje,
- kupi samostojno orodje za testno upravljanje, ki je združljivo z Jiro ali
- kupi dodatek k Jiri za testno upravljanje.

Jira

V kolikor se podjetje odloči, da bi organiziralo testiranje preko orodja Jira, bi bila to za podjetje najcenejša možnost, saj se to orodje vsakodnevno uporablja pri delu in za to ne bi bilo potrebno plačevati dodatnih licenc. Jiro se lahko organizira na dva načina.

1. Način

Vsak kreiran zahtevek za razvoj nove funkcionalnosti, mora vsebovati tudi podzahtevek, ki je namenjen temu, da tester vanj vnese enega ali več testnih scenarijev. Ko je zahtevek dokončan s strani razvijalca, nato tester novo funkcionalnost potestira in ustrezno označi zahtevek. V kolikor je zahteva ustrezno izvedena, tester označi zahtevek kot rešen. Če pa se med testiranjem odkrije napake ali pomanjkljivosti funkcionalnosti, tester zahtevek ponovno odpre in jasno opredeli razvijalcu odkrite napake oziroma pomanjkljivosti. Šele ko so vsi pod zahtevki ustrezno rešeni, se smatra zahtevek kot končan.

Prednost tega pristopa je predvsem poznavanje orodja Jira med zaposlenimi. Vpeljava novega načina dela bi bila tako precej nezahtevna za zaposlene in stranke ter cenovno neobremenjujoča.

Slabost takega načina je v tem, da je priprava testnega poročila precej otežena. Prav tako ni zgodovinskega pregleda rezultatov testiranj. Kot velika pomanjkljivost se pojavi tudi shranjevanje splošnih testnih scenarijev, ki jih mora tester stestirati ob regresijskem testiranju. Za regresno testiranje mora podjetje tako uporabljati še dodatno orodje, kjer bi shranjevalo vse testne scenarije. Za najbolj enostavno rešitev se lahko uporabi orodje Excel, ki se ga uporabi kot seznam obstoječih testnih scenarijev, kot tudi testno poročilo.

Pri zahtevkih, ki so kreirani na podlagi odkrite napake (napako prijavi tester v podjetju ali pa jo prijavi stranka), tester prav tako naredi pod-zahtevek za ročno testiranje. Ta zahtevek lahko vsebuje nov ali pa popravljen testni scenarij, lahko pa je namenjen le temu, da se sledi poslovnemu toku vodenja zahtevkov in testiranj.

2. Način

Za namene zagotavljanja kvalitete razvoja programske opreme, lahko podjetje v Jiri kreira nov projekt za vsako implementacijo posebej, na primer z imenom QA – implementacija XY. Na tem projektu se nato naredi nov zahtevek za posamezen vsebinski sklop programske opreme. Za vsak zahtevek pa se naredi nato več pod-zahtevkov, kjer vsak pod-zahtevek predstavlja en testni scenarij.

Prvotno kreiranim zahtevkom in pod-zahtevkom se nikoli ne spreminja statusa, ampak se jih za vsako iteracijo testiranja le kopira. Tako vsak kopirani zahtevek predstavlja iteracijo testiranja, pod-zahtevkom pa nato spreminjamo status glede na uspešnost ali neuspešnost izvedenega testa.

Ker se v vseh fazah posameznega projekta pojavljajo nove funkcionalnosti, je potrebno za novo razvite funkcionalnosti pripraviti testne scenarije. Testne scenarije lahko poslovni analitik ali pa tester v kateremkoli trenutku doda na osnovni zahtevek in ob naslednjem kopiranju zahtevka se tako skopirajo tudi novi pod-zahtevki.

Prav tako se lahko zahtevke, na podlagi katerih se razvija nova funkcionalnost, poveže s pod-zahtevki kreiranimi za testne scenarije. Tako lahko kdorkoli pogleda tako zgodovino in način implementacije zahtevka kot tudi testne scenarije, ki to funkcionalnost testirajo.

Prednost je enaka kot pri prvi možnost, se pa tukaj kot prednost pojavi tudi to, da ta način omogoča lažji pregled statusov testa. Pod-zahtevki so povezani na zahtevek in so zmeraj prikazani, ko odpremo posamezen zahtevek. Poleg vsakega pod-zahtevka so prikazani tudi statusi in tako dobimo takojšnji pregled stanja.

Slabost takega pristopa je lahko neorganiziranost zahtevkov in napačno poimenovanje. To lahko vodi do oteženega iskanja želenih iteracij testiranj, predvsem, ko se iteracij in s tem kreiranih zahtevkov nabere veliko.

Samostojno orodje

Pomemben pogoj pri izbiri samostojnega orodja za testno upravljanje je, da je združljivo z Jiro. Kot že omenjeno, podjetje uporablja Jiro za vodenje vseh zahtevkov in združljivost z Jiro bi pomenila nadgradnjo uporabe Jire.

Orodje, ki bi bilo primerno za podjetje, mora zadostiti sledečim zahtevam:

- čeprav bo podjetje moralo plačati orodje, to ne sme predstavljati prevelikega stroška,
- orodje mora biti kompatibilno z verzijo Jire, ki jo podjetje uporablja in to je verzija 7,
- omogočati mora povezljivost z orodjem za avtomatsko testiranje,
- omogočati mora izdelavo testnih poročil,
- omogočati mora večkratno uporabo istih testnih scenarijev in
- omogočati mora shranjevanje vseh testnih scenarijev.

Na trgu se pojavlja več takih orodij, spodaj so našteti le nekateri:

- QTest,
- PractiTest in
- TestRail.

Vsa orodja omogočajo tako integracijo z Jiro, kot tudi integracijo z orodjem Ranorex.

Sama povezljivost z Jiro omogoča, da uporabnik poveže posamezen zahtevek s testnim scenarijem. Ob sami izvedbi testnih scenarijev se nato rezultati testnih scenarijev prikažejo tudi v Jiri na posameznem zahtevku.

QTest orodje ima prilagodljiv cenik, kar pomeni, da ceno prilagodijo posamezni strani glede na njene potrebe in želje. Sam dodatek k Jiri pa je nato brezplačen.

PractiTest orodje je plačljivo glede na število uporabnikov, ki imajo dostop do samega orodja, cena pa znaša 45$ na uporabnika na mesec, kar pomeni 540$ na leto na uporabnika. Sam dodatek k Jiri pa je nato brezplačen.

Osnovna cena za TestRail znaša 240$ na uporabnika na leto, se pa cena na uporabnika niža glede na večje število uporabnikov. Tudi to orodje ponuja brezplačen dodatek k Jiri.

Dodatek k Jiri

Tretja možnost, ki bi bila primerna za podjetje je ta, da se odloči za enega izmed dodatkov k Jiri. Dodatki morajo zadostovati enakim kriterijem, kot samostojno orodje.

Na trgu obstaja kar nekaj takih dodatkov, med njih sodijo:

- XRay for Jira in
- SynapseRT

Izbrano podjetje ima zakupljenih 250 licenc za uporabo programa Jira. Za uporabo dodatka XRay for Jira bi moralo podjetje za ta dodatek dokupiti enako število licenc, kar bi naneslo 4000 dolarjev letno.

Za dodatek SynapseRT bi podjetje moralo za letni zakup licenc odšteti 3990 dolarjev.

Glede na primerjave med dodatkom XRay for Jira in SynapseRT sta si dodatka med seboj po funkcionalnosti precej podobna. Glavna negativna lastnost, ki jo ima XRay for Jira je slabo izdelano poročanje, ki pa je izbranemu podjetju pomemben faktor pri izbiranju podpornega orodja.

Velika prednost obeh orodij je vnos novih testnih scenarijev z uvozom Excel datoteke. Na ta način lahko podjetje pospeši izdelavo testnih scenarijev v izbranem orodju, saj trenutno že shranjuje testne scenarije v Excelovi datoteki.

5.2 Širjenje znanja

Chaffey in Wood (2005, str. 227) opisujeta upravljanje znanja (ang. *Knowledge management*) kot zmožnost posameznikov znotraj organizacije, da pridobijo znanje, ki je za njih pomembno, ga izboljšujejo in pridobljeno znanje nato na najbolj možen učinkovit način nudijo ostalim, ki ga potrebujejo, ti pa ga nato vnovčijo za dodajanje vrednosti njihovemu delu.

Strategija upravljanja znanja mora temeljiti na poslovnih ciljih (Chaffey & Wood, 2005, str. 227).

Chaffey in Wood (2005, str. 227) opisujeta pet ključnih procesov za upravljanje znanja, ki so bili razviti v European Framework:

- identifikacija znanja,
- ustvarjanje znanja,
- shranjevanje znanja,
- deljenje znanja in
- uporaba znanja.

Čeprav Scrum metodologija zagovarja minimalen nivo dokumentacije, tako velik in razpršen projekt potrebuje dokumentacijo že samo zaradi poenotenja terminologije, poznavanja in razumevanja posameznih procesov in širjenja predvsem vsebinskega znanja, ki je zelo pomanjkljivo.

Uradni jezik znotraj podjetja je postala angleščina in tudi vsa dokumentacija mora biti pisana v angleščini. Za potrebe določenih strank, se dokumentacijo nato prevede v lokalni jezik.

Trenutno dokumentacija ne obstaja, oziroma obstaja v zelo zanemarljivem in zastarelem obsegu. Za pomoč zaposlenim in strankam, torej končnim uporabnikom, bi podjetje moralo imeti vsaj tri tipe dokumentacije:

- uporabniška navodila,

- konfiguracijska navodila in
- navodila za razvijalce.

Pri pisanju posameznega tipa navodil je potrebno imeti v mislih, kdo je ciljna skupina bralcev teh dokumentov in temu skladno prilagoditi nivo pisanja. Uporabniška in konfiguracijska navodila so uporabna tako za interno uporabo kot tudi za dokumentacijo, ki se jo preda stranki. Navodila za razvijalce pa ostanejo le za interno uporabo.

Osnovni nivo navodil bi bilo potrebno pisati na podlagi stanja produkta. Posamezna implementacija nato nadgradi dokumentacijo s posebnostmi posamezne implementacije. Vsako dokumentacijo, ki se jo preda stranki, je potrebno pregledati in prilagoditi zahtevam in potrebam stranke. Posamezna stranka ne potrebuje vseh funkcionalnosti produkta ali pa je določene funkcionalnosti potrebno razviti glede na zahteve stranke, je s tem potrebno prilagoditi tudi posamezna navodila in odstraniti oziroma dodati navodila.

Ker pa se sam produkt spreminja, je potrebno tudi dokumentacijo verzionirati in jo posodabljati skupaj s spremembami produkta. Pisanje v programu Word zato ni najbolj primerno, saj lahko dokument kdorkoli in kadarkoli spreminja.

Primeren končni format dokumentacije bi lahko bil PDF (ang. *Portable Document Format*) format, ki je vsem dobro poznan. Pri pisanju dokumentacije je pomembno, da ne glede na to, kdo jo piše, se drži zmeraj enakega formata in stila pisanja. Za sledenje tem smernicam, se pokaže kot ustrezen format »Markdown«.

John Gruber (2004) razlaga Markdown na dva načina:

- Markdown je preprosto orodje za pretvorbo preprostega teksta v HTML format. Markdown omogoča format pisanja, ki je lahko berljiv in lahek za pisanje, nato pa ga uporabnik pretvori v HTML format ali pa v PDF format;
- Markdown je sintaksa za urejanje golega besedila (ang. *Plain text*).

Pretvorba iz Markdown v PDF ni avtomatska, zato je potrebno še dodatno orodje, ki lahko zagotovi pretvorbo. Primer ustreznega orodja je Pandoc, ki prav tako podpira izdelavo tabel, kazal, predlog dokumenta (Pandoc, b.l.).

Vsa dokumentacija, ki je oziroma bo napisana za namene dokumentiranja delovanja produkta, mora biti lahko dostopna vsem zaposlenim, kar pomeni, da mora biti objavljena na enem izmed strežnikov ali internih portalov, do katerega imajo vsi dostop.

Produktna skupina na tri mesece izda novo, večjo verzijo, ki pa vsebuje precej vsebinskih novosti v primerjavi s prejšnjo verzijo. Vsaki izdaji verzije bi morala slediti tudi nova verzija dokumenta, ki bi zajela novosti in spremembe produkta. S tem bi se obdržala zgodovina starejših verzij dokumenta in poskrbljeno bi bilo, da bi se sledilo izdaji verzij produkta.

Dokumentacija je pomemben del pri širjenju znanja med zaposlenimi in v pomoč novo zaposlenim, ki se lahko zanesejo na dokumentacijo in ne rabijo biti popolnoma odvisni od časa ostalih, zaposlenih za mentorstvo.

Dokumentacija pa ne more biti edini vir širjenja znanja, saj je vanjo težko zajeti vse odgovore. Pri širjenju znanja igra pomembno vlogo tudi osebni pristop, saj omogoča komunikacijo med zaposlenimi in lažjo interaktivnost.

Ob sami reorganizaciji so zaposleni organizirali vsebinska predavanja za ostale sodelavce, da bi si med seboj pomagali na vsebinskih področjih in maksimalno izkoristili čas na predavanjih. Ideja je bila pot v pravo smer, vendar so si predavanja sledila prehitro, saj sta bili na teden po dve predavanji. Vsak vsebinski sklop v programu je zelo obsežen in zahteva veliko posvečanja za dobro vsebinsko razumevanje. Eno predavanje težko zajame globino vsebine, saj je bil tudi profil prisotnih precej raznolik, prav tako je bil nivo znanja udeležencev precej raznolik, nekateri so bili popolni začetniki pri posamezni temi, drugi pa bi potrebovali že bolj napredno predavanje. Vsak predavatelj je tako razložil le osnove vsakega sklopa, vendar za zmožnost reševanja problemov, razvoja novih funkcionalnosti in testiranja funkcionalnosti, potrebuje zaposleni več poglobljenega znanja.

Predavanja so bila koristna z vidika splošnega pridobivanja razumevanja vsebine na kateri je nato vsak posameznik lahko gradil. Težava pa je bila, ker so ta predavanja potekala v slovenskem jeziku in so bila tako namenjena le zaposlenim v Sloveniji. Prav tako pa se teh predavanj v večini ni posnelo. Kar je predstavljalo zopet problem pri zaposlenih, ki so se podjetju pridružili kasneje. Novi zaposleni so težko prišli do osnovnega znanja, zaposleni, ki so imeli ta predavanja, pa so bili slabe volje, ker so ponovno morali razlagati enako vsebino, kot so jo že predavali.

Podjetje za sestanke na daljavo uporablja programsko opremo Cisco WebEx, ki pa tudi omogoča snemanje sestankov.

Cisco WebEx se izkaže za uporabno orodje, saj se organiziranemu sestanku oz. predavanju lahko pridružijo vsi, ki so prejeli povabilo in s tem tudi povezavi do virtualnega sestanka. Cisco WebEx prav tako omogoča gostitelju, da z drugimi udeleženci deli svoj ekran na katerem prikazuje predstavitev. Prav tako se posamezno predavanje ali pa sestanek snema in tako shrani posnetek (zvočni in vizualni) (WebEx, b.l.).

Ker se je od reorganizacije podjetja do sedaj zaposlilo precej novih ljudi, bi moralo podjetje ponovno organizirati predavanja, ki pa bi morala sedaj potekati v angleškem jeziku, da bi imeli koristi vsi zaposleni, ne glede na državo. Prav tako bi morali predavanja posneti, da si jih lahko zaposleni in novi zaposleni pogledajo kadarkoli.

Pri organizaciji novih predavanj pa je potrebno paziti, da jih vodijo osebe, ki imajo dovolj vsebinskega znanja za predstavitev določenega dela, so kvalitetno pripravljena in ne obremenijo preveč zaposlenih, ki se teh predavanj udeležujejo, torej da ne posegajo preveč v njihov delovnik in niso prepogosta.

5.3 Proces dela skupine

Ob ustanovitvi skupine so bili skoraj vsi člani skupine popolni začetniki tako na vsebinskem področju, kot tudi na področju samega testiranja. Sedaj, ko je skupina že vpeljana in sprejeta s strani ostalih zaposlenih, je potrebno začeti nadgrajevati samo organizacijo in način dela FLS skupine.

Projekti implementacijskih skupin, v katerih sodelujejo člani FLS skupine, se nahajajo v različnih fazah:

- faza razvoja,
- faza razvoja in delna vzdrževalna faza ter
- vzdrževalna faza.

V fazi razvoja tester sodeluje pri razvoju novih funkcionalnosti in testira delovanje programske opreme. V vzdrževalni fazi nudi podporo uporabnikom in tudi skrbi za testiranje popravkov ali novih funkcionalnosti ali sprememb, zahtevanih s strani stranke. Vmesna faza, kjer se prepletata razvoj in vzdrževanje, mora tester prav tako uspešno krmariti med testiranjem novih funkcionalnosti in nudenja podpore uporabnikom.

5.3.1 Testiranje in zagotavljanje kvalitete produkta v fazi razvoja

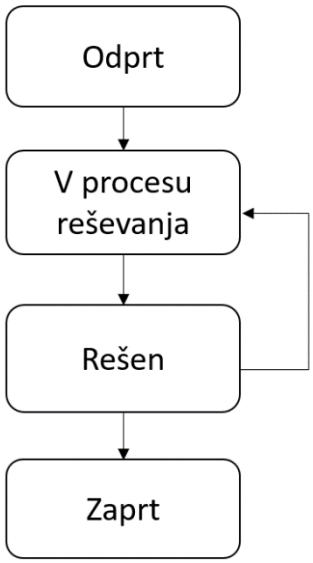
Glavna naloga testerja v razvojni fazi projekta je testiranje. Tester pripravlja nove testne scenarije glede na funkcionalnosti, ki jih razvijalci razvijajo. Vsaka novo razvita funkcionalnost mora imeti pripravljen vsaj en testni scenarij, s katerim bo tester pretestiral razvito funkcionalnost.

Izbrano podjetje pri pregledu zahtevkov, ki so v sprintu uporablja tabelo za pregled stanja posameznih zahtevkov. Tabela omogoča pregled, kateri zahtevki so v procesu razvoja, kateri še čakajo na razvoj in kateri so že zaključeni.

Za lažji nadzor nad Jira zahtevki, ki so že pretestirani in tistimi, ki še niso, predlagam uvedbo novega statusa. Z uvedbo novega statusa bi v sprint tabeli naredila tudi nov stolpec za prikaz zahtevkov, ki še čakajo na test.

Trenutno si statusi posameznega zahtevka sledijo v spodaj opisanem vrstnem redu:

Slika 6: Trenutni statusi v Jiri



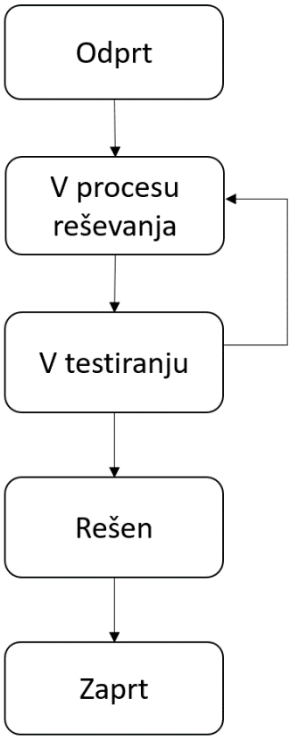
Vir: lastno delo.

Predlog podjetju je, da se doda nov status »V testiranju«. Ta status bi pomenil, da je zahtevek dokončan s strani razvijalca in ga mora tester pretestirati. Ne glede na to, za katero možnost uvedbe podpornega orodja za organizacijo testiranja se bo podjetje odločilo, predlagam uvedbo novega statusa, ki je prikazan na spodnji sliki.

Za vsak zaključen zahtevek s strani razvijalca, mora projektni vodja vedeti, ali je tester uspešno izvedel ročno testiranje ali avtomatsko testiranje programske opreme. Čeprav je testni scenarij, ki je kreiran na podlagi zaključenega zahtevka, dodan v seznam vseh obstoječih testnih scenarijev za regresijsko testiranje, mora vsak na novo rešen zahtevek skozi proces testiranja, da lahko zahtevek označimo kot uspešno zaključen.
Kot uspešno zaključen zahtevek se smatra tisti, ki:

- je zanj napisana koda,
- je zanj potrjena koda,
- je uspešno pretestiran, tako z unit testi, integracijskimi testi, ročnimi ali avtomatskimi funkcionalnimi testi,
- je na podlagi zahtevka, posodobljena dokumentacija:
  - uporabniška navodila,
  - konfiguracijska navodila,
  - navodila za razvijalce.

Slika 7: Predlagan potek statusov v Jiri



Vir: lastno delo.

V trenutnem procesu, ko zahtevek preide v status Rešen, nihče ne ve ali je tester res pogledal zahtevek in potrdil ustreznost delovanja zahtevka. Nov status bi pomenil lažji pregled nad delom testerja, statusi zahtevkov bi odražali pravo stanje zahtevkov. Nov status pa bi lahko bil tudi indikator ali ima tester na posamezni implementaciji preveč dela in ne zmore slediti tempu reševanja zahtevkov ali pa njegovo delo ni dovolj učinkovito.

Pred izdajo posamezne verzije ne zadošča samo testiranje rešenih zahtevkov, saj nikoli ne vemo, kako posamezna sprememba lahko vpliva na popolnoma drugo že obstoječo in delujočo funkcionalnost. Zato je zmeraj potrebno poskrbeti tudi za regresijsko testiranje, s

katerim preverimo splošno delovanje programske opreme. Za posameznega testerja je to lahko zelo velik zalogaj, poleg tega pa je tukaj vpleteno še tveganje, npr. da tester zboli, je ravno v tistem času na dopustu, noben drug tester ne more vskočiti namesto njega, …

Posamezna implementacija mora zato razmisliti o vpeljavi avtomatiziranega funkcionalnega testiranja (poleg obstoječih testov, ki jih pišejo razvijalci).

Produktna skupina je začela vpeljevati tudi avtomatizirano funkcionalno testiranje s pomočjo orodja Ranorex. Ranorex je orodje za razvoj in vodenje projektov, sestavljenih tako iz razvijalcev, kot tudi testerjev (Ranorex, b.l.).

Ranorex deluje tako, da razvijalec, oziroma tester, posname scenarij v programu, ki ga testira. Nato ta posnetek shrani in na podlagi posnetka, Ranorex sam izvede posnet testni scenarij. V kolikor je testni scenarij uspešen ali ne, Ranorex orodje pripravi testno poročilo, kjer pri neuspešnih testnih scenarijih pripravi ekranske posnetke za lažjo detekcijo napak.

Podjetje mora stremeti k temu, da se čim večje število ročnih testov, ki se izvajajo v sami programski opremi, avtomatizira. Z avtomatizacijo testiranja se razbremeni testerje in zmanjša se možnost napak pri testiranju.

Pri avtomatizaciji testnih scenarijev je potrebno premisliti ali je posamezen testni scenarij nagnjen k temu, da se lahko v prihodnosti večkrat spremeni zaradi narave funkcionalnosti, ki jo testira in ali je časovno smotrno, da se testni scenarij avtomatizira. V kolikor bi avtomatski testni scenarij potreboval več časa, da se izvede, kot če ga ročno izvede tester, je potrebno razmisliti o smiselnosti avtomatizacije.

Lahko se zgodi, da podjetje z avtomatizacijo testnih scenarijev pride do situacije, kjer bi iteracija testiranja potrebovala preveč časa in bi avtomatizirano testiranje izgubilo svoj namen.

5.3.2 Vzdrževanje (podpora uporabnikom)

Vsaka implementacija ima trenutno drugačen proces podpore končnim uporabnikom. Razlog za tako stanje je v tem, da nekega osnovnega principa delovanja in načina nudenja podpore ni bilo definiranega znotraj podjetja. Tako se je vsak projektni vodja v danem trenutku sam odločil in uskladil s stranko za najprimernejši način dela.

FLS skupina mora postati prva vstopna točka, na katero se obrne stranka s problemom. Trenutna situacija na večini implementacij izgleda tako, da lahko stranka kontaktira tako

projektnega vodjo kot tudi posamezne razvijalce. Komunikacija poteka preko elektronske pošte, telefona, preko zahtevkov v Jiri, v nekaterih primerih pa tudi preko programa Skype.

Neenoten način komunikacije je težaven zaradi sledečih razlogov:

- pri komunikaciji preko telefona lahko pride do različnega razumevanja samih težav uporabnikov,
- zaposleni v podjetju ob prejetem klicu nima časa, da se posveti javljeni napaki in lahko nanjo tudi pozabi,
- elektronska pošta je po navadi naslovljena le na enega prejemnika, ki pa je lahko takrat odstoten ali pa pozabi na prejeto pošto,
- prav tako lahko do težav pride, ko je zahtevek vnesen v Jiro, vendar pa ga prijavitelj dodeli direktno osebi in ta ali spregleda zahtevek ali pa je odsotna za dlje časa.

Za boljši pregled javljenih napak in kreiranih zahtevkov v Jiri, bi bilo potrebno za vsako implementacijo narediti vsaj eno skupino, v katero bi bili dodani vsi člani implementacijske skupine. Oseba na strani stranke, ki bi kreirala zahtevek, bi imela pravico zahtevek dodeliti le tej skupini. Tako bi bili vsi člani skupine obveščeni o novem zahtevku in vsi bi imeli dostop do njega.

Jira omogoča izdelavo filtrov, ki prikazujejo določen seznam zahtevkov. Ta seznam si lahko vsak uporabnik Jire pripne na začetno okno, čas osveževanja rezultatov posameznega filtra pa si lahko uporabnik določi sam. Tako imajo uporabniki Jire na začetnem zaslonu vedno pregled nad zahtevki in lažje nadzirajo stanje na projektu.

Vsak na novo kreiran zahtevek bi moral najprej skozi roke testerja, ki bi poskusil najprej ponoviti javljeno napako, poskusil sam rešiti napako, v kolikor je rešitev v njegovi domeni. Če pa gre za napako, ki zahteva popravek v kodi, preda zahtevek razvijalcu in poskusi razvijalcu zagotoviti čim več podatkov, scenarij za ponovitev napake, opis napake, izpis napake, itd. V kolikor potrebuje tester več informacij, lahko stopi tudi v kontakt s prijaviteljem napake. S tem razbremeni samega projektnega vodjo in razvijalce ter nase prevzame prvo komunikacijo s stranko.

Ko je napaka v zahtevku odpravljena, jo tester ponovno preveri in preveri pravilnost delovanja popravka. Po ustreznem popravku se v skladu z dogovorom s stranko namesti nova verzija programske opreme, ki vsebuje tudi prej omenjeni popravek.

Statusi zahtevkov bi tudi v tej fazi ostali tako, kot so predlagani v prejšnjem poglavju, torej z uvedbo novega statusa V testiranju. Ne glede na to ali gre v tej fazi projekta za popravek

napake v delovanju programske opreme, dodelavo obstoječe funkcionalnosti ali razvoj nove funkcionalnosti, vsak zahtevek mora tester pretestirati.

5.4 Končne ugotovitve

Podjetje mora še naprej graditi na avtomatizaciji testiranja. To lahko podjetju prinese večjo stroškovno učinkovitost, bolj kakovosten produkt in večje zadovoljstvo strank. Vse skupaj vpliva tudi na celotno klimo podjetja in zaposlenih. Vodilni in zaposleni lahko tako več časa namenijo razvoju novih funkcionalnosti in izboljševanju obstoječih.

Kvaliteten produkt je odvisen od več dejavnikov. Podjetje mora nuditi poslovnim analitikom ustrezno izobrazbo, tako interno kot tudi zunanjo. Analitik je tisti, ki sodeluje pri analizi in pripravlja analizo vrzeli. Dobro narejena analiza in dobro identificirane vrzeli so osnova za nadaljnji razvoj zahtev. Razvijalci veliko lažje razvijejo zahtevano funkcionalnost, če za funkcionalnost obstaja dobro definirana zahteva. Dober razvoj pomeni manj vsebinskih napak v razviti funkcionalnosti. Dobro definirana zahteva je podlaga tudi za testerja, ki pripravlja testne scenarije in jih nato tudi izvaja. Če tester ne razume funkcionalnosti in poslovne potrebe za funkcionalnost, je tudi ne more dobro in pravilno pretestirati. Vse skupaj pa na koncu vodi v boljši produkt in v manjšo količino podpore stranki v fazi produkcije.

Poleg dobro narejene analize je pomembno tudi poenotenje procesov razvoja produkta in implementacije, nastanek in izboljšava trenutne dokumentacije ter povečanje pokritosti s testnimi scenariji.

Pri uvajanju novih zaposlenih lahko urejena dokumentacija in posneta predavanja predstavljata velik del izobraževanja in uvajanja novih članov in s tem manjšo odvisnost od prostega časa ostalih zaposlenih. Zaposleni se lahko tako več časa posvečajo svojim nalogam, novi zaposleni pa se lahko neodvisno od volje in časa ostalih učijo in spoznavajo produkt.

Poenotenje dela na projektih in samega procesa vodenja zahtevkov olajša delo tako vodilnim, ki imajo s tem boljši pregled nad stanjem posameznega projekta, kot tudi članom skupin, ki se lahko lažje vključijo v posamezno skupino.

Trenutno stanje na področju testiranja, testnih scenarijev in upravljanja s procesom testiranja je v podjetju neurejeno. Po pregledu treh različnih možnosti in različnih orodij, ki so na voljo za organizacijo testiranja, podjetju predlagam uvedbo dodatka k Jiri SynapseRT.

SynapseRT je cenovno sprejemljiv, je dodatek k Jiri, ki ga podjetje že uporablja in so vsi zaposleni dodobra seznanjeni z delovanjem. Prehod na uporabo dodatka bi bil tako manj boleč kot pa pri uvedbi samostojnega orodja. Dodatek omogoča pripravo poročil o testiranju, pregled izvedenih testnih scenarijev v posameznem testnem ciklu. Prav tako lahko uporabnik posamezen testni scenarij poveže z zahtevkom. Prav tako je možno organiziranje testnih scenarijev glede na vsebinske sklope, kar precej olajša pregled in iskanje testnih scenarijev.

SKLEP

Uspešno rastoče podjetje, ki ima večje število zaposlenih in po možnosti še dislocirane pisarne, mora imeti urejene interne procese, kar je spoznalo tudi podjetje, ki ga obravnavam v magistrski nalogi.

Kot raziskovalno vprašanje magistrske naloge sem si postavila vprašanje ali je bila vpeljava in način organizacije FLS skupine v podjetju uspešna. Glede na to, da v podjetju pred nastankom FLS skupine testiranja niso poznali in glede na vso preučeno teorijo, ki daje testiranju zelo velik pomen na projektu, je odgovor da. Podjetje je z uvedbo FLS skupine naredilo korak v pravo smer, vendar pa FLS skupino sedaj čaka še veliko novih izzivov, ki jih je potrebno realizirati.

Za vsakega novo zaposlenega je pomembno, da ima podjetje zanj pripravljen material za učenje. V obravnavanem podjetju je poznavanje vsebine zelo pomemben dejavnik za uspešno delo. Novo zaposleni tako ni odvisen od časa zaposlenih in njihove volje po mentoriranju novih zaposlenih. S popisano dokumentacijo in posnetimi predavanji lahko novo zaposleni hitreje pridejo do novega znanja in podlage za bolj zahtevno specializacijo in učenje. S tem se podjetje tudi izogne, da bi mentor novim zaposlenim pozabil predati pomembne informacije.

Dokumentacija je pomembna tudi z vidika obstoječih in novih strank. Stranke potrebujejo zagotovilo, da bo podjetje priskrbelo vso potrebno dokumentacijo in navodila za uporabo programske opreme, ki jo bodo končni uporabniki uporabljali vsak dan. Poznavanje programske opreme lahko zmanjša tudi število napak, ki bi jih potencialno končni uporabniki naredili zaradi nepoznavanja programske opreme. Posledično to vodi v zmanjšanje števila prijavljenih napak in podpore končnim uporabnikom.

Zato je usmeritev časa in truda zaposlenih pomembna za pisanje dokumentacije, saj le ta olajša delo zaposlenim in olajša odnose s stranko in tako pomaga preseči strah strank pri uporabi nove programske opreme.

Za zadovoljstvo stranke poskrbi tudi dobro delujoča programska oprema. Podjetje mora interne vire in čas usmeriti v zgodnje odkrivanje napak. Vse se začne že pri dobri komunikaciji s stranko in dobro definirani analizi, kjer se specificira želje stranke in končno rešitev. Dalje v procesu igrata pomembno vlogo razvijalec in tester, ki poskrbita vsak na svoj način za dobro napisano kodo in za dobro testirano razvito funkcionalnost. Oba nato poskrbita tudi za novo dokumentacijo, oziroma ureditev obstoječe dokumentacije.

Eden izmed korakov za uspešno in učinkovito testiranje je tudi uvedba podpornega orodja za organizacijo testiranja. Na podlagi raziskanih možnosti ima podjetje več različnih opcij za katero se lahko odloči, moj predlog je uvedba orodja SynapseRT.

Urejena dokumentacija in urejeno testiranje so pomembni dejavniki k ureditvi stanja v podjetju in izboljšanju kvalitete produkta in komunikacije in sodelovanja s strankami. Prav tako veliko k nadaljnjim uspehom podjetja prispeva ureditev internih procesov, ki jih podjetja ureja z uvedbo agilne metodologije, kjer so definirane vloge vseh zaposlenih v podjetju. Sama uvedba ni dovolj, pomembna je konsistentnost pri izvajanju agilne metodologije, ki zagotavlja zaposlenim poznavanje pravil, organiziranosti in urejenosti.

V kolikor bo podjetje sledilo začrtanim izboljšavam, je na dobri poti k povečanju učinkovitosti in uspešnosti. Urejenost delovnih procesov optimizira delo posameznih zaposlenih in tako omogoča lažje planiranje dela zaposlenih in načrtovanje novih projektov.

LITERATURA IN VIRI

1. Agile Data. (brez datuma). *Introduction to Test Driven Development*. Pridobljeno 16. marca 2017 iz http://agiledata.org/essays/tdd.html
2. Aljaber, T. (brez datuma). *The iron triangle of planning.* Pridobljeno 5. januarja 2017 iz https://www.atlassian.com/agile/agile-iron-triangle
3. Allman, E. (2012, 23. marec). *Managing technical debt*. Pridobljeno 20. maja 2017 iz http://queue.acm.org/detail.cfm?id=2168798
4. Ampatzogloua, A., Ampatzogloua,A., Chatzigeorgioub, A. &Avgerioua, P. (2015). The financial aspect of manaing technical debt: a systematic literature review. *Information and Software Technology, 64*(2015), 52–73.
5. Anderson, D. (2013, 2. avgust). *Kanban – an alternative path to agility*. Pridobljeno 15. maja 2017 iz http://www.djaa.com/kanban-alternative-path-agility
6. Avison, D. & Fitzgerald, G. (2003). *Information systems development: Methodologies, Techniques and tools.* New York: McGraw -Hill Education.
7. Bach, J. (2013, 26. marec). *Testing and Checking Refined*. Pridobljeno 15. maja 2017 iz http://www.satisfice.com/blog/archives/856
8. Baiju, M. (2007, avgust). *Ten skills of highly effective software testers*. Pridobljeno 15. maja 2017 iz http://searchsoftwarequality.techtarget.com/tip/Ten-skills-of-highly-effective-software-testers
9. Balaji, S. & Sundararajan Murugaiyan. M. (2012). Waterfall vs V-model vs Agile: A comparative study. *International Journal of Information Technology and Business Management, 2*(2012), 1.
10. Beck, K., Beedle. M., van Bennekum, A., Cocburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C, Mellor, S., Schwaber, K., Sutherland, J. & Thomas, D. (2001). *Agile Manifesto*. Pridobljeno 2. februarja 2016 iz http://agilemanifesto.org/iso/sl/
11. Boehm, B. &Turner, R. (2005). Management challenges to implementing agile processes in traditional development organizations. *IEEE Software,22*(5), 30-39.
12. Bowman, D. (brez datuma). *What is Prototyping Methodology?* Pridobljeno 15. maja 2017 iz http://www.information-management-architect.com/prototyping-methodology.html
13. Buenen, M. & Teje, M. (2014). *World quality report*. Pridobljeno 15. maja 2017 iz https://www.uk.capgemini.com/thought-leadership/world-quality-report-2014-15
14. Burnstein, I. (2003). *Practical software testing*. New York: Springer-Verlag New York, Inc.
15. Chaffey, D. & Wood, S. (2005). *Business Information management: Improving performance using information systems*. Harlow: Financial Times Prentice Hall
16. Crispin, L. & Gregory, J. (2009). *Agile testing: A practical guide for testers and agile teams*. Boston: Pearson Education, Inc.

17. Dingsøyr, T., Nerur, S., Balijepally, V. & Moe, N. B. (2012). A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and Software, 85*(6), 1213-1221.
18. DuPont, N. (2013, 3.december). *Static Testing vs. Dynamic Testing*. Pridobljeno 15. maja 2017 iz https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing
19. Elberzhager, F., Münch, J. & Nha, V.T.N. (2011). A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Information and Software Technology, 54*(2012), 1–15.
20. Extreme Programming. (brez datuma). *Introducing Extreme Programming.* Pridobljeno 16. marca 2017 iz http://www.extremeprogramming.org/introduction.html
21. Fowler, M. (2005, 5. marec). *Test Driven Development*. Pridobljeno 3. januarja 2017 iz http://martinfowler.com/bliki/TestDrivenDevelopment.html
22. Garousi, V. & Mäntyläc, M.V. (2016). When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology, 76*(2016), 92–117.
23. Ghahrai, A. (2015, 14. januar). *Software development life cycle – SDLC Phases*. Pridobljeno 15. maja 2017 iz http://www.testingexcellence.com/software-development-life-cycle-sdlc-phases/
24. Ghahrai, A. (2016a, 5. maj). *Static Analysis vs Dynamic Analysis in Software development.* Pridobljeno 15. maja 2017 iz http://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/
25. Ghahrai, A. (2016b, 29. november). *Pros and Cons of Test Driven Development.* Pridobljeno 15. maja 2017 iz http://www.testingexcellence.com/pros-cons-test-driven-development/
26. Gradišar, M. & Resinovič, G. (1998). *Informatika v organizaciji*. Kranj: Moderna organizacija.
27. Gruber, J. (2004, 17. december). *Markdown*. Pridobljeno 20. aprila 2017 iz http://daringfireball.net/projects/markdown/
28. Herzig, K., Greiler, M., Czerwonka, J. & Murphy, B. (2015). *The Art of Testing Less without Sacrificing Quality.* Pridobljeno 21. aprila 2017 iz https://www.microsoft.com/en-us/research/wp-content/uploads/2015/05/The-Art-of-Testing-Less-without-Sacrificing-Quality.pdf
29. Hill, S. (2015, 23. februar). *The Pros and Cons of Test-Driven Development.* Najdeno 15. maja 2017 iz https://leantesting.com/resources/test-driven-development/
30. ISTQB Exam Certification. (brez datuma). *What is Prototype model- advantages, disadvantages and when to use it?* Pridobljeno 15. aprila 2017 iz http://istqbexamcertification.com/what-is-prototype-model-advantages-disadvantages-and-when-to-use-it/

31. IT Knowledge portal. (brez datuma). *Software Development Methodologies*. Pridobljeno 1. aprila 2017 iz http://www.itinfo.am/eng/software-development-methodologies/
32. Jeffries, R. (2001, 8. november). *What is Extreme Programming?* Pridobljeno 24. novembra 2016 iz http://ronjeffries.com/xprog/book/whatisxp/
33. Jiang, L. & Eberlein, A. (2009). An analysis of the history od classical software development and agile development. *Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics, 2009,* 3733-3738.
34. Kempka, J., McMinn, P. & Sudholt, D. (2015). Design and analysis of different alternating variable searches for search-based software testing. *Theoretical Computer Science, 605*, 1–20.
35. Kovačič, A., Jaklič, J., Indihar Štemberger, M. & Groznik, A. (2004). *Prenova in informatizacija poslovanja*. Ljubljana: Univerza v Ljubljani, Ekonomska fakulteta.
36. Krebs, J. (2009). *Agile portfolio management*. Redmond, Washington: Microsoft Press
37. Kumar, S. (2012, 12. januar). *What is test management? Advantages of using test management tools, list of tools available.* Pridobljeno 9. decembra 2017 iz http://www.mobileqazone.com/profiles/blogs/what-is-test-management-advantages-of-using-test-management-tools
38. Kupiainen, E., Mäntylä, M.V. & Itkonen, J. (2015). Using metrics in Agile and Lean Software development- a systematic literature review of industrial studies. *Information and Software Technology, 62*, 143–163.
39. Lauden, J.P. & Lauden, K.C. (2014). *Management Information Systems*. Harlow: Pearson Education Limited.
40. Lei, H., Ganjeizadeh, F., Jayachandran, P.K. & Ozcan, P. (2015). A statistical analysis of the effects of scrum and kanban on sofware development projects. *Robotics and Computer-Integrated Manufacturing, 43*, 59-67.
41. Lina, L., Hea, J., Zhanga, Y. & Song, F. (2015). Quality assurance through rigorous software specification and testing: case study. *Procedia Computer Science, 62*, 257 – 265.
42. Marschall, M. (2015, 25. julij). *Kanban vs Scrum vs Agile*. Pridobljeno 15. maja 2017 iz http://www.agileweboperations.com/scrum-vs-kanban
43. Masri, W. & Zaraket, F.A. (2016). Coverage-based software testing: beyond basic test requirements. *Advances in Computers, 103,* 79-142.
44. Myers, G.J. (2004). *The Art of Software Testing*. New Jersey: John Wiley & Sons, Inc.
45. Nehal, J. (2009, 24. november). *Advantages & Disadvantages of prototyping process model*. Pridobljeno 15. maja 2017 iz https://www.iotap.com/blog/entryid/124/advantages-disadvantage-of-prototyping-process-model
46. Net Objectives. (brez datuma). *Chapter 9. The Role of Quality Assurance in Lean-Agile Software Development*. Pridobljeno 3. aprila 2017 iz

http://www.netobjectives.com/files/books/lasd/role-quality-assurance-lean-agile-software-development.pdf
47. Okoli, C. & Carillo, K. (2010). *The best of adaptive and predictive methodologies: Open source software development, a balance between agility and discipline*. Pridobljeno 26. oktobra 2016 iz http://chitu.okoli.org/media/pro/research/pubs/OkoliCarillo2010IJAESD.pdf
48. Opiyo, E.Z., Horváth, I. & Vergeest, J.S.M. (2002). Quality assurance of design support software: review and analysis of the state of the art. *Computers in Industry, 49*, 195–215.
49. Pandoc. (brez datuma). *About Pandoc.* Pridobljeno 10.maja 2016 iz http://pandoc.org/
50. Papadopoulos, G. (2015). Moving from traditional to agile software development methodologies also on large, distributed projects. *Procedia - Social and Behavioral Sciences, 175*, 455-463.
51. Philipson, G. (brez datuma). *A short history of software*. Pridobljeno iz http://www.thecorememory.com/SHOS.pdf
52. Pierce, W. (2016, 21. maj). *Disadvantages and Advanatges of Extreme Programming.* Pridobljeno 15. maja 2017 iz https://atlaz.io/blog/disadvantages-and-advantages-of-extreme-programming/
53. Powell-Morse, A. (2016, 26. december). *V-Model: What is it and how do you use it?*. Pridobljeno 15. maja 1017 na spletnem naslovu https://airbrake.io/blog/sdlc/v-model
54. Patton, J. (2009, 20. april). *Kanban Development Oversimplified: How Kanban-style development gives us another way to deliver on Agile values*. Pridobljeno 15. maja 2017 iz http://jpattonassociates.com/kanban_oversimplified
55. Ramadoss, M. (2015, 17. avgust). *Comparision of Test Management Tools*. Pridobljeno 4. maja 2017 iz http://blogs.perficient.com/delivery/blog/2015/08/17/comparison-of-test-management-tools/
56. Rouse, M. (2017, 25. maj). *Test driven development (TDD)*. Pridobljeno 1. junija 2017 iz http://searchsoftwarequality.techtarget.com/definition/test-driven-development
57. Shalloway, A., Beaver, G. & Trott, J.R. (2009). *The role of Quality Assurance in Lean-Agile Software Development.* Pridobljeno 27. aprila 2017 iz http://www.netobjectives.com/files/books/lasd/role-quality-assurance-lean-agile-software-development.pdf
58. Silver, M. (2014, 7. Avgust). *Waterfall Methodology Advantages and Disadvantages.* Pridobljeno 12. aprila 2017 iz http://spectechular.walkme.com/waterfall-methodology-advantages-disadvantages/
59. Slavchev, V. (2016a, 28. marec). *Software testing is not…* Pridobljeno 10. maja 2017 iz http://mrslavchev.com/2016/03/28/software-testing-not/
60. Slavchev, V. (2016b, 25. april). *Software testing is not … part 2*. Pridobjeno 10. maja 2017 iz http://mrslavchev.com/2016/04/25/software-testing-not-part-2/

61. Slavchev, V. (2016c, 26. maj). *Software testing is not…finding bugs part 4*. Pridobljeno 10. Maja 2017 iz http://mrslavchev.com/2016/05/16/software-testing-not-finding-bugs-part-4/
62. Slavchev, V. (2016d, 30. maj). *Software testing is not…playing. Part 5*. Pridobljeno 10. maja 2017 iz http://mrslavchev.com/2016/05/30/software-testing-not-playing-pt5/
63. Shore, J. (2010, 26. marec). *The Art of Agile Development: Test-Driven Development.* Pridobljeno 13. aprila 2017 iz http://www.jamesshore.com/Agile-Book/test_driven_development.html
64. Shrivastava, S.V. &Rathod, U. (2015). Categorization of risk factors for distributed agile projects. I*nformation and Software Technology, 58*, 373- 387
65. Stevenson, J. (2014, 10. november). *The role of testers in an agile environment*. Pridobljeno 15. januarja 2017 iz https://www.stickyminds.com/article/role-testers-agile-environment
66. Turban, E., McLean, E. & Wetherbe, J. (1999). *Information technology for management. Making connections for strategic advantage.* New York: John Wiley & Sons Inc.
67. Tutorials point. (brez datuma). *Test driven development.* Pridobljeno 7. marca 2017 iz http://www.tutorialspoint.com/software_testing_dictionary/test_driven_development.htm
68. Tutorials point. (brez datuma). *SDLC - Waterfall model*. Pridobljeno 27. aprila 2017 iz https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm
69. WebEx. (brez datuma). *Why choose WebEx?* Pridobljeno 15. aprila 2017 iz https://www.webex.com/faqs.html#carousel-nav1
70. WebEx. (brez datuma). *Why are WebEx products my best choice for online collaboration?* Pridobljeno 15. aprila 2017 iz https://www.webex.com/faqs.html#carousel-nav1-q2
71. Wells, D. (2009). *Surprise! Software Rots!* Pridobljeno 22. maja 2017 iz http://www.agile-process.org/change.html
72. Yu, X. & Petter, S. (2014). Understanding agile software development practices using shared mental models theory. *Information and Software Technology, 56*, 911–921.
73. Yusufoğlu, V.G, Amannejad, Y. & Can, A.B. (2015). Software test code engineering: a systematic mapping. *Information and Software Technology, 58*, 123–147.