Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Uroš Krivec

Priporočene prvine razvoja programske opreme

diplomsko delo na visokošolskem strokovnem študiju

Ljubljana, 2016

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Uroš Krivec

Priporočene prvine razvoja programske opreme

diplomsko delo na visokošolskem strokovnem študiju

Mentor: viš. pred. dr. Igor Rožanc

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Priporočene prvine razvoja programske opreme

Tematika naloge:

Tehnologija programske opreme pozna veliko metodologij in pristopov, ki pripomorejo k učinkovitemu in pravočasnemu razvoju kakovostnih programskih izdelkov. V praksi pa se prepogosto pokaže, da je njihova uspešna uporaba vse prej kot enostavna, kar še zlasti velja za manjša podjetja za razvoj programske opreme. V tem primeru podjetje oblikuje svojstven način razvoja, ki temelji na specifičnih potrebah in lastnih izkušnjah. V diplomski nalogi predstavite skupek desetih priporočenih prvin razvoja programske opreme, ki ste jih identificirali kot ključne v svoji dolgoletni karieri kot tehnični vodja v manjšem slovenskem podjetju. Izbiro vsake ustrezno utemeljite, nato pa prikažite tudi nabor podpornih orodij in zgled uporabe le-teh na praktičnem zgledu. Nalogo zaključite s presojo koristi ob uporabi vseh prvin skupaj.

Izjava o avtorstvu zaključnega dela

Spodaj podpisani Uroš Krivec, vpisna številka 63020241, avtor zaključnega dela z naslovom:

Priporočene prvine razvoja programske opreme

(angl. *Recommended principles of software development*)

IZJAVLJAM

1. da sem pisno zaključno delo študija izdelal samostojno pod mentorstvom viš. pred. dr. Igor Rožanca;
2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;
3. da sem pridobil vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil;
4. da sem pri pripravi pisnega zaključnega dela študija ravnal v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil soglasje etične komisije;
5. soglašam, da se elektronska oblika pisnega zaključnega dela študija uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
6. da na Univerzo v Ljubljani neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija Univerze v Ljubljani;

7. dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.

V Ljubljani, dne 4. avgust 2016 Podpis študenta:

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
SDLC	Software Development Life Cycle	življenjski cikel razvoja programske opreme
RAD	Rapid Application Development	model hitrega razvoja programske opreme
IEEE	Institute of Electrical and Electronics Engineers	Svetovno združenje inženirjev elektronike in elektrotehnike
SQA	Software Quality Assurance	zagotavljanje kakovosti razvoja programske opreme
SQAP	Software Quality Assurance Plan	načrt zagotavljanja kakovosti razvoja programske opreme
SQALE	Software Quality Assessment based on Lifecycle Expectations	ocena kakovosti programske opreme na podlagi pričakovanj življenjskega cikla
ARA	Application Release Automation	avtomatizacija izdaje programskih rešitev
XP	eXtreme Programming	ekstremno programiranje

Povzetek

Namen diplomskega dela je predstaviti nabor prvin razvoja programske opreme, ki so potrebne za zagotavljanje kakovosti končne programske rešitve. Nabor prvin je osnovan na podlagi avtorjevih delovnih izkušenj. V diplomskem delu so predstavljeni trije modeli življenjskega cikla programske opreme: slapovni model, model hitrega razvoja programske opreme in modeli agilnega razvoja (poudarek na metodologiji Scrum). Opisana je opredelitev kakovosti razvoja programske opreme, ter pojma: tehnični dolg in ocena SQALE. Izpostavljene prvine so obravnava uporabniških zahtev, programski jezik, sistem za upravljanje z izvorno kodo, vejitve in načini dela z izvorno kodo, integracijski strežnik, sistem za hranjenje programskih komponent, avtomatski in osebni pregled izvorne kode, namestitev programske rešitve ter post-mortem analiza. V zadnjem delu je povezana uporaba dela predstavljenih prvin prikazana na primeru razvoja spletne trgovine. Ob tem je prikazan tudi način avtomatizacije dela razvoja programske opreme. Na koncu predstavimo prednosti in učinke uporabe priporočenih prvin.

Ključne besede: razvoj programske opreme, življenjski cikel razvoja programske opreme, Scrum, kakovost programske opreme, priporočene prvine.

Abstract

Purpose of this thesis is representation of software development principles that are needed in order to insure a quality software product. Selection of principles is based on author's work experiences. The thesis presents three different software development life cycle models: waterfall model, rapid application development model and agile models (focused on Scrum). Thesis describes software development process quality and two related concepts: technical debt and SQALE grade. Recommended principles are management of user requirements, programming language, version control system, branching and work flows, build server, build artifacts server, automatic and peer code review, deployment mechanism and postmortem analysis. In last segment the thesis shows combined use of recommended elements on a use case - development of an on-line store. An example how to automate software development process is shown on the same use case. Finally advantages and benefits of this approach are listed.

Keywords: software development, software development life cycle, Scrum, software quality, recommended principles.

Poglavje 1

Uvod

Pri razvoju programske opreme se pogosto srečujemo s pojmoma kakovost programske rešitve in zadovoljstvo uporabnika. Še pogosteje pa z nerealnimi časovnimi roki, slabo opredeljenimi uporabniškimi zahtevami in pojmom »stabilizacija programske rešitve« še po tem, ko je že bila nameščena v produkcijsko okolje. Pojmi, ki smo jih našteli med zadnjimi vodijo v nezadovoljstvo uporabnikov in odstopanja od začrtane časovnice projekta, predvsem pa v višje stroške razvoja.

Avtor je v svoji dosedanji karieri sodeloval pri razvoju mnogih programskih rešitev za različne naročnike. Nastopal je v vlogah (glavnega) razvojnika, vodje skupine razvojnikov, arhitekta programske rešitve ali tehničnega svetovalca. Ob delu je vedno iskal načine kako zagotoviti višjo kakovost končnega izdelka, še posebej pa ga navdušuje avtomatizacija postopkov pri razvoju programske opreme.

V diplomskem delu želimo izpostaviti nekaj priporočenih prvin, ki jih je avtor na podlagi svojih izkušenj ocenil kot nepogrešljive v vsakem tehnološkem podjetju, ki se ukvarja z razvojem programske opreme. Priporočene prvine so primerne za manjša in srednje velika podjetja, saj za vpeljavo ne zahtevajo velikih finančnih sredstev. Prvine se izboljšav razvoja programske opreme lotevajo iz metodološkega, tehničnega in (delno) poslovnega vidika.

Pred predstavitvijo prvin se bomo seznanili z izbranimi modeli življenjskega cikla programske opreme in zagotavljanjem kakovosti v procesu razvoja. Priporočene prvine bomo predstavili na hipotečnem primeru z izbranim naborom orodji. Opisan primer lahko služi kot načrt souporabe priporočenih prvin v podjetju. S predstavljeno uporabo prvin dosežemo preprečevanje oz. predčasno odkrivanje napak, enostavno sledenje napredku in visoko stopnjo avtomatizacije procesa razvoja. Olajšano je tudi uvajanje in izobraževanje novih članov projektne ekipe. Tako zastavljen proces razvoja programske opreme pozitivno vpliva na stroške razvoja, pravočasnost izvedbe in kakovost končnega izdelka.

Poglavje 2

Razvoj programske opreme

V nadaljevanju bomo predstavili osnove razvoja programske opreme. Spoznali bomo pojem življenjski cikel razvoja programske opreme in izpostavili tri modele življenjskega cikla, ki so po avtorjevem mnenju pustili največji pečat v načinu projektnega dela. Izpostavili bomo slapovni model, model hitrega razvoja programske opreme in agilne modele (s poudarkom na metodologij Scrum). Opredelili bomo pojem in izpostavili dva pomembna parametra kakovosti razvoja programske opreme.

Razvoj programske opreme je pojem, ki opisuje široko področje izdelave programskih rešitev. Izdelava programske rešitve praviloma vključuje naslednje faze: zbiranje zahtev, načrtovanje, izdelavo, testiranje in vzdrževanja programske rešitve. Razvoj programske opreme je proces.

2.1 Življenjski cikel razvoja programske opreme

Življenjski cikel razvoja programske opreme (*angl. Software Development Life Cycle - SDLC*) generično opredelimo kot zaporedje faz (korakov), ki so potrebni za zasnovo, razvoj in vzdrževanje programske rešitve. V ciklu si sledi šest faz: zbiranje in analiza zahtev, načrtovanje, izvedba, testiranje, namestitev in vzdrževanje [1].

Skupaj z razvojem informacijskih tehnologij se je izoblikovalo več modelov življenjskega cikla razvoja programske opreme.

2.2 Modeli življenjskih ciklov razvoja programske opreme

Vpeljava modelov življenjskih ciklov za razvoj programskih opreme standardizira proces razvoja.

Modeli se med seboj razlikujejo po marsičem, a zanimiva je ločitev na tiste, ki se lažje (*angl. adaptive*) in tiste, ki se težje prilagajajo (*angl. predictive*) na spremembe tekom trajanja projekta [2]. Modeli razvoja programske opreme se neprestano razvijajo (dopolnjujejo in nadgrajujejo) in se s tem prilagajajo novim potrebam v poslovnih okoljih.

Vsak model razvoja programske opreme vsebuje različno število faz (z več pod-fazami). Ponavadi obstajajo pri vsakem modelu trije večji sklopi faz: zasnova, izvedba in vzdrževanje. Bolje domišljeni modeli (ob sosledju faz) opredelijo tudi vloge (akterje), aktivnosti, izdelke in povezave med njimi. Z uporabo modela razvoja programske opreme pridobimo ogrodje, ki nam omogoča razvoj programske rešitve ter organizacijo, vodenje in spremljanje projekta [3].

Izbira pravilnega pristopa k razvoju programske rešitve je odvisna od več dejavnikov. Naštejmo najbolj pogoste:

- pogostost sprememb v uporabniških zahtevah,
- tip programske rešitve (nova, v vzdrževanju, razvoj produkta ali razvoj podpore naročnikovi storitvi),
- velikost projektne ekipe in čas,
- znanja in izkušenost projektne ekipe,
- količina sredstev.

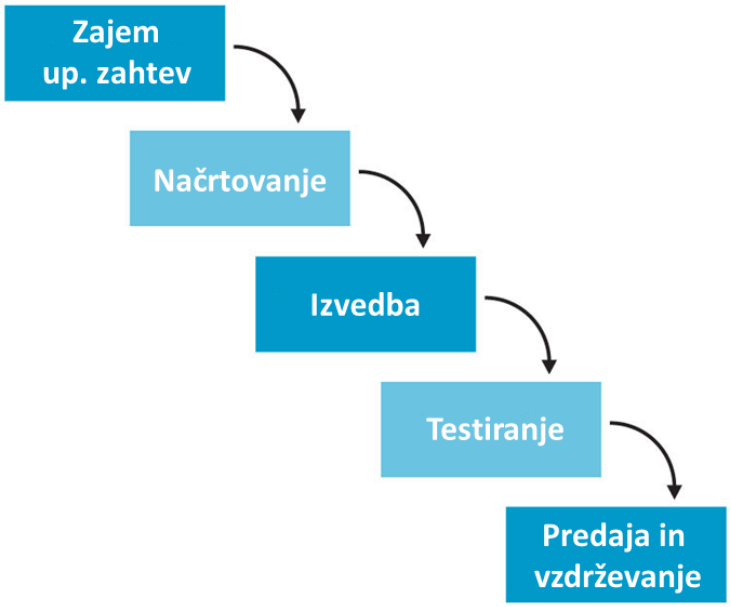
Ob izbiri se je potrebno zavedati, da idealen model ne obstaja in da se prvine izbranega modela pogosto prilagajajo med samim trajanjem projekta [4].

V nadaljevanju bomo predstavili tri najbolj prepoznane in hkrati najbolj pogosto uporabljene pristope k razvoju programske opreme.

2.2.1 Slapovni model

Slapovni model (*angl. waterfall model*) je dolgo časa veljal za najpogosteje izbranega pri razvoju programske opreme. Model posnema vzorce, ki ji je moč opaziti v tovarniških in gradbenih procesih, kot model razvoja programske opreme pa ga je prvi opredelil Winston W. Royce (1970).

Model sestoji iz petih faz, ki sledijo generičnem modelu razvoja programske opreme. Faze modela življenjskega cikla prikazuje slika 2.1.



Slika 2.1: Faze slapovnega modela razvoja programske opreme.

Značilnost slapovnega modela je, da so faze zaporedne in se med seboj ne prekrivajo. Ob koncu posamezne faze se rezultati preverijo in potrdijo.

Pogosto se na konce faz vežejo tudi pogodbene finančne obveznosti. Končni izdelki posamezne faze so gradivo za začetek faze, ki sledi.

Projekt, ki poteka po slapovnem modelu je preprost in lahek za planiranje (časovnica in potrebe po kadrih se lahko opredelijo pred začetkom prve faze) in se dobro obnese na manjših projektih, kjer so uporabniške zahteve dobro znane in definirane že ob začetku. Model se odsvetuje, če obstaja velika stopnja verjetnosti, da se bodo uporabniške zahteve pogosto spreminjale ali pa so bile nejasno izražene. Slabost modela je tudi, da se naročnik in končni (testni) uporabniki s programsko rešitvijo v živo prvič srečajo v fazi testiranja, ko so vse uporabniške zahteve že implementirane. Takrat je težko popraviti nekaj, kar ni bilo dobro opredeljeno že v fazi analize in načrtovanja. Naknadni popravki povzročajo zamik projektne časovnice in odstopanje od finančne konstrukcije projekta. Prav zaradi tega je model neprimeren tudi za velike, dolgotrajne projekte.

V praksi tak pristop pogosto uporabimo po prehodu v produkcijsko okolje, torej za vzdrževanje že stabilnih programskih rešitev. Nadgradnje že vpeljanih programskih rešitev naročnik pogosto bolje definira, saj ima pred seboj delujočo implementacijo in neposreden odziv končnih uporabnikov [5, 6].

2.2.2 Model hitrega razvoja programske opreme

Model hitrega razvoja programske opreme (*angl. Rapid Application Development - RAD*) temelji na hitrem razvoju v krajših ponovitvah in izdelavi prototipov delujoče programske opreme. Dandanes ga pogojno prištevamo v skupino agilnih modelov razvoja programske opreme, saj je njegov cilj naročniku v čim krajšem času predstaviti programsko rešitev.

Model je nastal kot odziv na togost in zaporednost faz slapovnega modela, kjer se pogosto zgodi, da se uporabniške zahteve spremenijo še preden v celoti zaključimo z implementacijo programske rešitve. Predstavil ga je James Martin leta 1991 [8].

Model vpeljuje koncept prototipa (*angl. prototype*) (predvsem pri načrtovanju uporabniškega vmesnika), ki olajša delo analitikom pri zajemanju in

preverjanju uporabniških zahtev. Takšen delujoč prototip je nato osnova v katerega implementiramo poslovna pravila. S tem pristopom zmanjšamo možnost napak pri zajemu uporabniških zahtev, prav tako uporaba delujočega prototipa pohitri razvoj celotne programske rešitve [7].

Model RAD vsebuje štiri faze, ki so predstavljene na sliki 2.2.

- **Načrtovanje zahtev** – faza, v kateri sta združena analiza in načrtovanje (opredelitev) poslovnih zahtev.
- **Funkcionalna zasnova** – faza, v kateri razvijemo delujoč prototip; pri izdelavi se opiramo na informacije poslovnih analitikov in uporabnikov in tako izdelamo prototip, ki vsebuje procese in zahteve končne programske rešitve.
- **Izvedba** – zaključek razvoja programske rešitve; uporabniki prototip testirajo in vračajo razvojni ekipi v dopolnitev tako dolgo, dokler se prototip ne razvije v končno programsko rešitev.
- **Uvajanje** – faza, v kateri so zajete vse dejavnosti, ki jih običajno povezujemo z zaključkom projekta: končna testiranja, migracija podatkov, predaja programske rešitve in usposabljanje končnih uporabnikov.

Prednosti uporabe modela RAD so predvsem hitrost izvedbe (faza analize zahtev se delno že prekriva s fazo razvoja programske rešitve, ko gradimo delujoč prototip), kakovost in osredotočenost na bistvene funkcionalnosti končne rešitve [7].

S cikličnim preverjanjem napredka delujočega prototipa pri uporabnikih zmanjšujemo stopnjo tveganja, da bi z implementacijo zašli od uporabniških zahtev. Sodelovanje z uporabniki med samim razvojem programske rešitve naredi model RAD bolj prilagodljiv spremembam.

Model ni primeren za zelo velike projektne ekipe in zahtevne projekte. Najbolje se izkaže pri malih in srednje velikih projektnih ekipah. Če uporabimo model RAD na velikem projektu s številčno projektno ekipo, so slabosti modela še bolj očitne. Prototipni pristop izdelave programske rešitve

Slika 2.2: Faze modela hitrega razvoja programske opreme.

lahko povzroči opuščanje razvojnih standardov (slaba zasnova jedra programske rešitve, odsotnost pouporabljivosti, modularnosti in skalabilnosti izvorne kode) in dobrih praks [9].

2.2.3 Modeli agilnega razvoja

Med agilne modele oz. metode prištevamo metode razvoja programske opreme, ki jim je skupen iterativen, evolucijski in prilagodljiv pristop.

Glavna razlika v primerjavi s klasičnimi metodami je dolžina trajanja ene ponovitve razvojnega cikla. Če pri klasičnih metodah iteracija traja npr. 3 do 6 mesecev, iteracija agilne metode načeloma traja od enega do štirih tednov. S krajšimi iteracijami razvoja programskih rešitev se pojavijo manjša tveganja, manjša je zahtevnost programske rešitve, od uporabnikov pa hitreje pridobimo povratne informacije.

Značilnosti agilnih metod so:

- boljša komunikacija v projektni ekipi,
- aktivna vloga vsakega posameznega člana projektne ekipe,

- projekta ekipa dosega skupne odločitve,
- transparentnost procesa razvoja,
- končna programska rešitev se gradi z majhnimi koraki,
- osredotočenost na redno dostavo programske rešitve končnim uporabnikom.

Poudarek je na zaključevanju nalog. Za zaključek naloge je potrebno vsako nalogo testirati (testiranje je del življenjskega cikla vsake naloge). Naloga tako ni končana, če je nismo prej preverili. Le po zaključku obstoječe naloge se lotimo nove.

Uporabniške zahteve niso opredeljene do zadnje podrobnosti – praviloma so podane na »visokem nivoju«, kot ga vidi in razume končni uporabnik. Zato sta sodelovanje in dobra komunikacija celotne projektne ekipe ključnega pomena za jasnost uporabniških zahtev.

Čeprav se uporabniške zahteve razvijajo – se časovni okvir ne spreminja. Če to lastnost primerjamo s klasičnimi modeli razvoja, kjer dodajanje novih zahtev praviloma podaljša časovnico, agilne metode spremembo upoštevajo, vendar se na račun spremembe iz časovnice odstrani neka druga naloga [11].

Agilne metode (predvsem metodi Scrum [13] in Kanban [14] oz. kombinacija obeh) so se izkazale za priljubljeno izbiro mladih tehnoloških podjetij. Izdelava programske rešitve, ki se inkrementalno dopolnjuje (in inkrementalno namešča) omogoča takim podjetjem zgodnji vir dohodka (npr. naročnina storitve, mobilna aplikacija v spletni prodajalni). Prednosti agilnih metod so tudi preglednost, kvaliteta (testiranje vgrajeno v življenjski cikel uporabniške zahteve), boljše obvladovanje tveganj, zmožnost prilagajanja na spremembe, obvladovanje stroškov (spremenjene zahteve ne spremenijo časovnega okvirja izvedbe ostaja enak) in vpletenost naročnika (ker je naročnik vpleten – sodeluje ob odločitvah in ima pregled nad stanjem in napredkom projekta, kar vpliva tudi na zadovoljstvo naročnika). Zaradi skupnih odločitev naročnika

in izvajalca se bo na koncu oblikovala končna programska rešitev, ki bo narejena po meri naročnika [12].

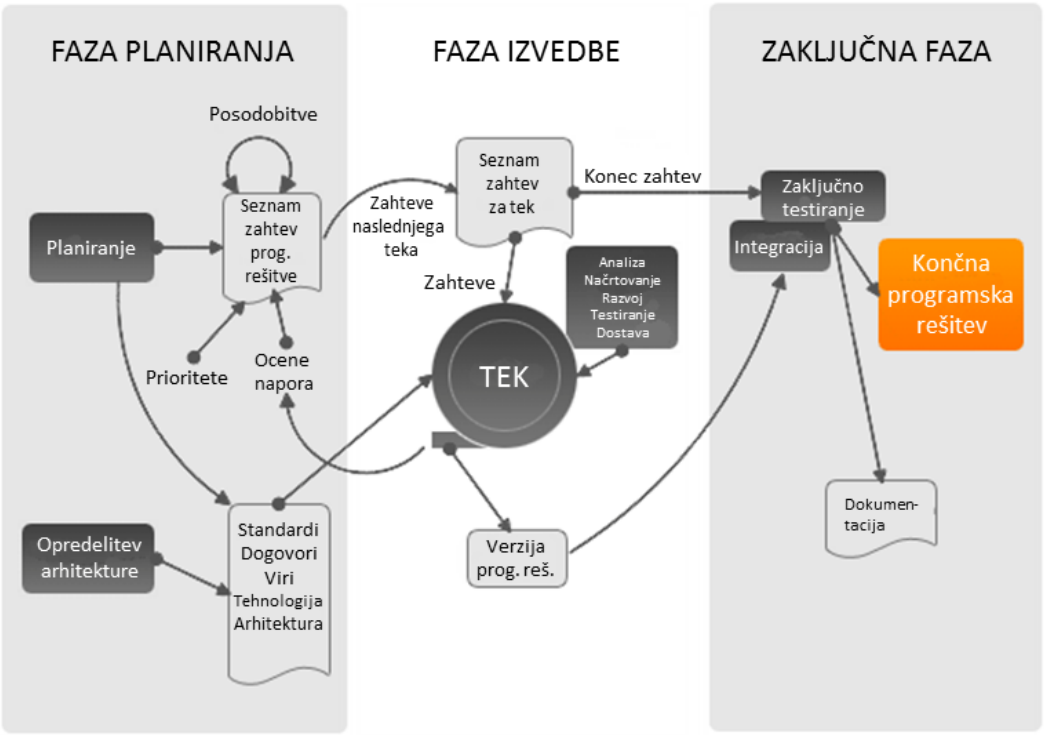
Scrum

Metoda Scrum [10, 13] velja za najpogosteje uporabljeno agilno metodo razvoja programskih rešitev. Izvira iz metode RAD (hitra izdelava prototipov), vendar je prilagojena za okolja, kjer so uporabniške zahteve že v začetku nepopolno opredeljene in obstaja velika verjetnost, da se bodo hitro spreminjale [7].

Optimalno okolje za uvedbo metode Scrum so manjše ekipe (do 10 ljudi). V tako veliki projektni ekipi je najlažje preverjati dnevni napredek in obvladovati tveganja. Poleg stalne projektne ekipe je za Scrum značilna serija tekov (*angl. sprint*). Tek je časovno obdobje med dvema do štirimi tedni in vedno traja enako dolgo. Vsak tek lahko smatramo kot mini projekt, v katerem želimo pripraviti nadgrajeno verzijo programske rešitve [15]. Metoda strogo opredeljuje časovne okvirje in navade (dolžina trajanja teka, redni dnevni in tedenski sestanki), ter spodbuja hitro dostavo izvedenih sprememb programske rešitve.

Vse postopke metode razdelimo v tri faze tekom življenjskega cikla razvoja programske rešitve, kot prikazuje slika 2.3.

Faza planiranja (*angl. planning phase – pre-game phase*), ta faza vsebuje dve razvojni stopnji projekta – opredelitev arhitekture in oblikovanje želenih lastnosti bodoče programske rešitve. Pri opredeljevanju arhitekture (na visokem nivoju) se ne omejimo le na uporabljeno tehnologijo, temveč opredelimo tudi standarde, projektno ekipo, uporabljena orodja in druge vire, ki bodo potrebni med trajanjem projekta. V tej fazi pripravimo seznam vseh lastnosti programske rešitve, ki jih je potrebno razviti (*angl. product backlog*). Seznam zapišemo v obliki uporabniških zgodb (*angl. user story*) urejenih po ustrezni prioriteti, ki skupaj tvorijo bolj obširne lastnosti (*angl. epics*). Vsaka uporabniška zgodba ima oceno (*angl. story points*), ki odraža zahtevnost izvedbe. Lastnik programske rešitve (*angl. product owner*) skrbi,



Slika 2.3: Faze metode Scrum [59].

da je seznam urejen, posodobljen z novimi zahtevami (v obliki novih uporabniških zgodb), dopolnjen s podrobnejšimi zahtevami in z bolj natančnimi ocenami. S prilagajanjem seznama skrbi tudi za to, da razvojna ekipa dela na tistih funkcionalnostih, ki imajo najvišjo dodano vrednost za naročnika.

V tej fazi je, s celotno projektno ekipo, opredeljena tudi pomembna definicija - definicija končanega (*angl. definition of done*). Definicija opredeljuje kdaj je uporabniška zgodba s seznama zaključena - tako vsebinsko (izpolnjuje vse kriterije sprejemljivosti), tehnično (opravljen pregled izvorne kode, testiranje enot pokriva 70% izvorne kode) kakor tudi postopkovno (opravljeno testiranje testne ekipe, opravljen pregled lastnika rešitve).

Faza izvedbe (*angl. development phase – mid-game phase*) je izvedbena faza, ki je razdeljena na teke, torej enako dolga obdobja izvedbe uporabniških

zgodb (slika 2.4). Za vsako tako obdobje člani ekipe izberejo seznam zgodb (*angl. sprint backlog*), ki jih bodo v tem obdobju zaključili. Lastnik programske rešitve lahko svetuje, katere zgodbe naj bodo vključene v tek, ne sme pa ukazovati. Načrtovanje teka poteka na skupnem sestanku (*angl. sprint planning*) ob začetku teka.

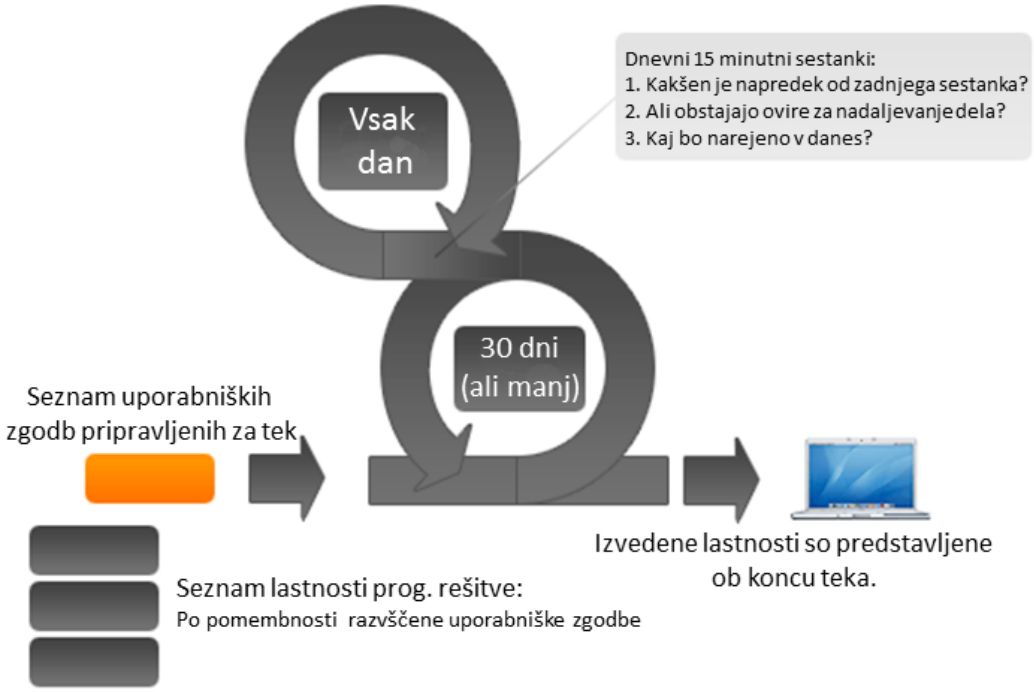
Nad potekom teka bdi skrbnik metodologije (*angl. scrum master*) [16]. Skrbnik je odgovoren za to, da tek poteka brez težav: ekipi pomaga odstraniti ovire, ki se pojavijo ob delu, skrbi, da člani ekipe niso moteni z zunanjimi vplivi in da vsi člani ekipe razumejo pravila Scrum metodologije.

Dnevno se člani ekipe zberejo na kratkem petnajst minutnem sestanku (*angl. daily scrum*), ki vedno poteka na dogovorjenem mestu ob določeni uri. Na sestanku vsak član ekipe predstavi aktivnosti svojega dne in povzame dosežke preteklega. Opozori lahko tudi na morebitne težave, ki jih ima pri svojem delu. Napredek teka se dnevno prikazuje s pomočjo t.i. grafa izvedbe (*angl. burn down chart*). Graf prikazuje količino zaključenih uporabniških zgodb v odvisnosti časovnega okvirja enega teka.

Ob koncu teka se opravi predstavitev opravljenega dela (*angl. sprint review*). Na tem sestanku razvojni del ekipe predstavi razvite funkcionalnosti vsej ekipi. Glavni namen predstavitve je pridobiti povratno informacijo vseh vpletenih. Povratna informacija je v pomoč lastniku programske rešitve pri urejanju seznama uporabniških zgodb in načrtovanju naslednjih tekov.

Vsak tek zaključi sestanek na katerem člani ekipe izpostavijo dobre in slabe dogodke (*angl. sprint retrospective*), ki so jih opazili med trajanjem teka. Sodelujoči podajo razloge za doseženo hitrost teka (*angl. sprint velocity*). Definicija hitrosti je seštevek vseh točk končanih uporabniških zgodb v teku. Ugotovljena hitrost razvojne ekipe je v pomoč pri načrtovanju naslednjih tekov. Posebna pozornost je namenjena razlogom, ki so privedli do tega, da nekatere uporabniške zgodbe niso bile zaključene. Cilj sestanka je, da se ekipa dogovori za izboljšave obstoječega procesa dela, ki jih bo izvedla do prihodnjič.

Po potrebi se ožja ekipa sestane na posebnem sestanku (*angl. backlog refi-*



Slika 2.4: Izvedbena faza metode Scrum [59].

nement / grooming) na katerem pregleda in uskladi novonastale uporabniške zgodbe, da so jasno definirane in kot take pripravljene za vključitev v enega od naslednjih tekov. Sestanek ni obvezen del Scrum cikla, vendar pripomore h kvalitetnejši opredelitvi uporabniških zgodb, kar kasneje pospeši njihovo implementacijo.

Zaključna faza (*angl. closure phase – post-game phase*) je faza predaje programske rešitve, ki se prične takrat, ko vodstvo oceni, da so vsi tehnični in vsebinski predpogoji za prehod v produkcijo pripravljeni, ko je bilo opravljeno končno (sistemsko) testiranje z vsemi delujočimi integracijami in napisana vsa tehnična in uporabniška dokumentacija. Programska rešitev je pripravljena za prehod v produkcijo.

2.3 Kakovost razvoja programske opreme

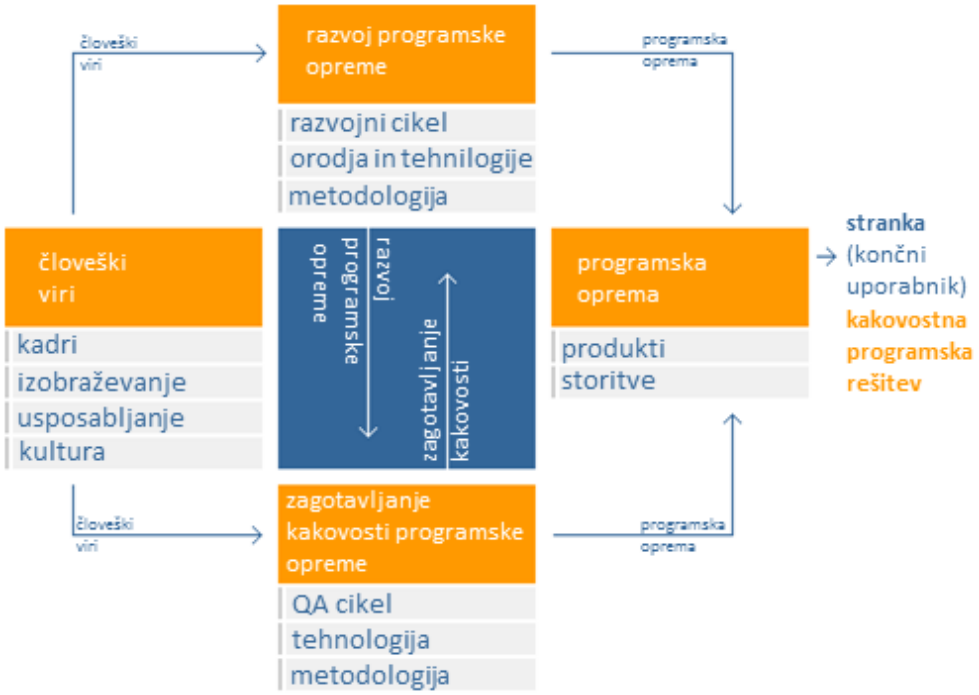
Inštitut IEEE (*angl. Institute of Electrical and Electronics Engineers*) opredeljuje kakovost programske opreme z dvema definicijama: stopnja prilagajanja sistema, komponente ali procesa podanim zahtevam in stopnja prilagajanja sistema, komponente ali procesa naročnikovim potrebam ali pričakovanjem [18].

Ob pojmu kakovosti programske opreme, se srečamo tudi z vprašanjem kako zagotoviti kakovostno programsko opremo (*angl. Software Quality Assurance - SQA*) v procesu razvoja programske opreme. IEEE opredeljuje zagotavljanje kakovosti programske opreme kot načrtovan seznam vseh potrebnih korakov, za zagotovitev zadostne stopnje zaupanja v to, da se programska oprema (produkt) prilagaja tehničnim zahtevam oz. nabor postopkov s katerimi ocenjujemo proces nastanka programske opreme [18].

Proces razvoja programske opreme lahko oplemenitimo z različnimi mehanizmi zagotavljanja kakovosti. Raven kakovosti lahko dvignemo s preprosto vpeljavo primerov dobrih praks in orodij. Še bolje pa je, če zadostimo mednarodnim standardom (npr. ISO 9001 [19]) in uveljavljenim modelom vrednotenja uspešnosti in zrelosti procesov (npr. CMMI [20]). Slika 2.5 prikazuje povezavo med vsemi službami organizacije, ki vodijo do kakovostnega končnega izdelka.

Pomemben dokument, ki ga pripravimo v okviru projekta, je načrt zagotavljanja kakovosti (*angl. Software Quality Assurance Plan - SQAP*). V njem opredelimo postopke, orodja, oblike poročil, ki jih bo ekipa za zagotavljanje kakovosti uporabljala pri svojem delu. V načrt vključimo oblike in načine obveščanja med ekipo za zagotavljanje kakovosti ter ostalimi sodelujočimi na projektu. Postopki zagotavljanja kakovosti pri razvoju programske opreme niso omejeni le na strogo tehnični del (pregledi izvorne kode, izvajanje različnih testov programske rešitve, upravljanje), ampak se vključujejo v celoten življenjski cikel projekta (programske rešitve) širijo pa se tudi v delovanje organizacije [17].

S postopki zagotavljanja kakovosti želimo zmanjšati možnosti za: napačno



Slika 2.5: Sodelovanje različnih služb znotraj organizacije za zagotavljanje kakovosti končnega izdelka [58].

(slabo) opredeljene uporabniške zahteve, nejasnosti v komunikaciji med naročnikom in razvojno ekipo, odstopanja od zahtev, napake v načrtovanju in implementaciji programskih komponent, odstopanja od dogovorjenih standardov pri implementaciji, odsotnost testiranja, napake postopkov in dokumentiranja [18].

2.3.1 Tehnični dolg

Kakovost programske rešitve pogosto opišemo s pojmom tehnični dolg (*angl. technical debt, design debt*). V osnovi ga merimo v časovnih enotah (npr. 10 dni in 9 ur), kar je preprosto pretvoriti v količino sredstev potrebnih za odpravo dolga.

Pojem opisuje čas porabljen za razvoj, ki bo namenjen temu, da nadome-

stimo vse nedomišljene (hitre) rešitve v izvorni kodi z najbolj optimalnimi in zanesljivimi. Čeprav izraz povezujemo s slabo implementirano izvorno kodo, lahko nastane tudi zaradi slabih arhitekturnih odločitev, odlašanjem s prestrukturiranjem programske kode (*angl. refactoring*), nejasnih uporabniških zahtev, pomanjkljive dokumentacije, slabega razumevanja problema ali nerealnih projektnih rokov. Izraza se je domislil Ward Cunningham, ko je poskušal upraviteljem organizacije v kateri je bil zaposlen, predstaviti smoter prestrukturiranja programske kode.

Posledice visokega tehničnega dolga (če je programska rešitev že v produkcijskem okolju) so težje vzdrževanje in (časovno) zahtevnejše nadgradnje. Nadgradnje posledično lahko pripeljejo do nestabilnosti ali izpada produkcijskega sistema, kar pripelje do finančnih izgub ali pogodbenih kazni [21, 22].

2.3.2 Ocena SQALE

Ocena SQALE (*angl. Software Quality Assessment based on Lifecycle Expectations*) predstavlja enoznačno oznako za stopnjo tehničnega dolga programske kode. Ocena je osnovana na razmerju tehničnega dolga:

*strošek odprave pomanjkljivosti / strošek razvoja vrstice pr. kode * št. vrstic*

Ocenjevalna metoda je generična in neodvisna od programskega jezika in orodij za statično analizo programske kode. Ocena se naslanja na štiri komponente:

- model kakovosti,
- analitični model,
- različni indeksi,
- indikatorji.

Indeksi pokrivajo najpogostejše pojme kakovostne izvorne kode – preverljivost (*angl. testability*), zanesljivost, zamenljivost, učinkovitost, varnost,

vzdrževalnost, prenosljivost in pouporabljivost. Spremenljiva komponenta, ki si jo vsak uporabnik metode prilagodi svojim potrebam ali potrebam organizacije so indikatorji. Za prikaz ocene SQALE se najpogosteje uporablja vrednosti med A (najboljše) in E (najslabše) [23].

Poglavje 3

Priporočene prvine razvoja programske opreme

V poglavju bomo predstavili nekatere priporočene prvine razvoja programske opreme. Izbrane prvine pozitivno vplivajo na kakovost končne programske rešitve, pripomorejo k preglednosti, stabilnosti, zmanjšujejo tveganja in omogočajo preprosto spremljanje napredka projekta - tako v metodološkem, tehničnem in poslovnem smislu.

Pomembna cilja, ki jih želimo doseči sta jasnost stanja projekta in avtomatizacija nekaterih zamudnih postopkov pri razvoju programske opreme. Izpostavljene prvine omogočajo izpolnitev obeh ciljev, kar po našem mnenju, končni programski rešitvi prinaša zrelost in kakovost.

Prvine so bile izbrane na podlagi desetletja avtorjevih delovnih izkušenj, opazovanj in spremljanja trendov IT sveta. Primerne so za manjša oz. srednje velika podjetja (ki sama nimajo dovolj sredstev, da bi uvedla kakšnega od znanih standardov), vendar dovolj prilagodljive, da rastejo in se razvijajo skupaj s podjetjem.

Predstavljene prvine lahko uvajamo posamično oz. nepopolno, vendar se najboljše rezultate dosežemo z upoštevanjem vseh. V tem pogledu je avtorjev nabor priporočenih prvin podoben modelu ekstremnega programiranja (*angl. eXtreme Programming - XP*). Omenjeni model prav tako priporoča uporabo

prvin, ki povečajo produktivnost in dvignejo raven kakovosti. Nekatere od prvin, ki ji model ekstremnega programiranja izpostavlja (npr. pisanje programskih testov, pregledi izvorne kode) so priporočene tudi v našem modelu.

3.1 Obravnava uporabniških zahtev

S pojmom uporabniška zahteva (*angl. user requirement*) opisujemo opredeljeno potrebo uporabnika po določeni funkcionalnosti v nastajajoči programski rešitvi. Zahteve so zbrane v dokumentu uporabniških zahtev (*angl. user requirements document*), ki nastane v analitični fazi projekta.

Izbrana metodologija projekta vpliva na način upravljanja uporabniških zahtev med razvojno fazo. Če programsko rešitev razvijamo po slapovnem modelu pričakujemo, da so uporabniške zahteve natančno opredeljene in usklajene z naročnikom. Agilni pristop v analitični fazi (npr. priprava seznama želenih lastnosti programske rešitve) dopušča ohlapnejše opredeljene zahteve, ki se nato natančneje opredelijo v sodelovanju z naročnikom na dnevnem nivoju. Uporabniške zahteve pretvorimo v zahtevke (naloge) v fazi načrtovanja. Pri agilnih metodah (npr. Scrum) pa se uporabniške zahteve že zapisujejo v obliki zahtevka (uporabniška zgodba).

Med razvojem programske rešitve uporabimo orodja za upravljanje zahtevkov (*angl. issue tracking system*). Z uporabo takega orodja med razvojem programske rešitve, lahko zahteve razporejamo med razvijalce. Vsak zahtevek vsebuje navodila za implementacijo, potrebno dodatno dokumentacijo, pomembnost in okvirno oceno napora (zapisano kot čas ali točke). Tip zahtevka je tudi prijava napačnega delovanja programske rešitve (*angl. bug report*). Pri prijavljenih napakah je pomembno, da prijavitelj zapiše okoliščine (datum in uro napake, aplikacijsko okolje) in korake za ponovitev napake. Vsak zahtevek sledi življenjskemu ciklu, ki je opredeljen s tipom zahtevka.

Uporaba orodij za upravljanje zahtevkov močno poenostavi pregled nad napredovanjem projekta, olajša organizacijo dela, omogoča prenos znanja in komunikacijo med razvijalci in naročnikom. Na podlagi vseh zajetih podat-

kov o spremembah na zahtevkih orodja, preko nabora metrik, omogočajo analitični pogled na napredek pri razvoju programske rešitve.

Pomembno je da z naročnikom uskladimo kanale in oblike komunikacije, v nasprotnem primeru se lahko zgodi, da nove (oz. spremenjene) uporabniške zahteve prihajajo v izvedbo slabo opredeljene in nepremišljene. Zahtevi ni moč slediti, če je prišla do razvojnika preko telefona in bila takoj izvedena. V primeru, da razvojnik zapusti razvojno ekipo, z njegovim odhodom izgubimo tudi del znanja o okoliščinah nastanka spremembe.

3.2 Programski jezik

Izbira programskega jezika lahko močno vpliva na kakovost in stabilnost končne programske rešitve. Pri izbiri moramo upoštevati domeno programske rešitve, naročnika (če je ta znan) in kakovost samega programskega jezika.

Ob izbiri programskega jezika, upoštevajoč domeno programske rešitve, je potrebno pretehtati vse vidike izvajalnega okolja (namen uporabe, hitrosti izvajanja, prenosljivost, zunanje omejitve). Če rešitev implementiramo za znanega naročnika, ki bo kasneje prevzel vzdrževanje programske rešitve, je izbira odvisna od nivoja znanja naročnikove vzdrževalne ekipe, njegove tehnološke usmeritve in s tem povezanih stroškov. Pogosto izbira programskega jezika (in izvajalnega okolja) ni v domeni izvajalca, pač pa je predmet dogovora med izvajalcem in naročnikom. Izbira programskega jezika je takrat opredeljena že v dokumentu tehnične specifikacije projekta.

Kakovost programskega jezika je širši pojem in vključuje zrelost jezika (količina dokumentacije, nabor orodij, količina skupnih programskih knjižnic, uveljavljenost v večjih podjetjih), velikost in aktivnost skupnosti uporabnikov jezika, ter tehnične lastnosti jezika (zmogljivost in stabilnost). Pri skupnih programskih knjižnicah gre opozoriti na pomembnost zmogljivih knjižnic (ogrodij) za izvajanje testiranja enot (*angl. unit test*) in imitacijo programskih konstruktov (*angl. mocking*). Če programski jezik ne ponuja ogrodij za izvedbo testov enot, bomo težko zagotovili pravilno delovanje izvorne kode.

Testi enot so potrebni za opis pravilnega delovanja izvorne kode in za preverjanje vpliva dograjenih sprememb.

Kriterij za izbiro programskega jezika, ki pomembno vpliva na kvaliteto procesa razvoja programske rešitve, je zmožnost avtomatizacije procesov (*angl. build automation*) - prevajanja izvorne kode v strojno, testiranja prevedene programske kode in grajenja datotek za namestitev (*angl. build artifacts*) v aplikacijsko okolje.

Avtomatizacijo opisanega postopka dosežemo z uporabo orodij za upravljanje projektov (*angl. build management tools*). Orodja so pogosto povezana s programskim jezikom in zagotavljajo enotno definicijo postopka in deležnike procesa. Poleg zaporedja korakov (ali izjem v korakih) postopka preko orodij upravljamo z uporabljenimi programskimi knjižnicami in vključujemo dodatne vtičnike, ki nadgradijo postopek (npr. izdelava poročil o prilagajanju izvorne kode uveljavljenim standardom). Modernejša orodja vključujejo tudi mehanizme za naslavljanje in posredovanje programskih knjižnic z medmrežja. Uporaba teh orodij standardizira izvajalni proces, omogoča projektu večjo prenosljivost, ter lažje in hitrejše vključevanje novih članov v razvojno ekipo. Ključni kriteriji ob izbiri orodja naj bodo:

- razširjenost,
- preprostost upravljanja uporabljenih programskih knjižnic,
- količina prosto dostopnih vtičnikov,
- prilagoditve postopka grajenja namestitvenih datotek.

Ob zadnji točki merimo na postopek, ki ga orodje predpisuje za izdelavo končnega izdelka. Če primerjamo orodiji Apache Ant [24] in Apache Maven [25], opazimo očitno razliko. Apache Ant omogoča popolno svobodo pri veriženju korakov ob gradnji, medtem ko Apache Maven predpisuje zaporedje korakov.

Premislek o izbiri programskega jezika običajno opravimo v fazi analize zahtev in načrtovanja projekta. Pogosto je izbira jezika odvisna od splošne tehnične usmeritve podjetja izvajalca.

3.3 Sistem za upravljanje z izvorno kodo

Sistemi za upravljanje z izvorno kodo zagotavljajo hranjenje (navadno na namenskem strežniku) in verzioniranje izvorne kode. Razvijalci implementirajo spremembe na svoji, lokalno shranjeni, verziji izvorne kode. Ko z implementacijo zaključijo, skupek sprememb pošljejo na strežnik z nameščenim sistemom za upravljanje izvorne kode. Vsaka sprememba izvorne kode je hranjena pod unikatno verzijo – revizijo (*angl. revision*). Ob tem se shranita tudi časovni žig in oznaka avtorja spremembe. Tako natančno vodenje revizijskih oznak omogoča takim sistemom primerjavo, združevanje in restavriranje točno določenega stanja izvorne kode.

Prednosti uporabe takih sistemov so:

- vzporedno delo več razvojnikov na izvorni kodi,
- vpogled v zgodovino sprememb izvorne kode,
- vejitve (*angl. branching*) in združevanje (*angl. merging*) izvorne kode,
- sledljivost spremembam.

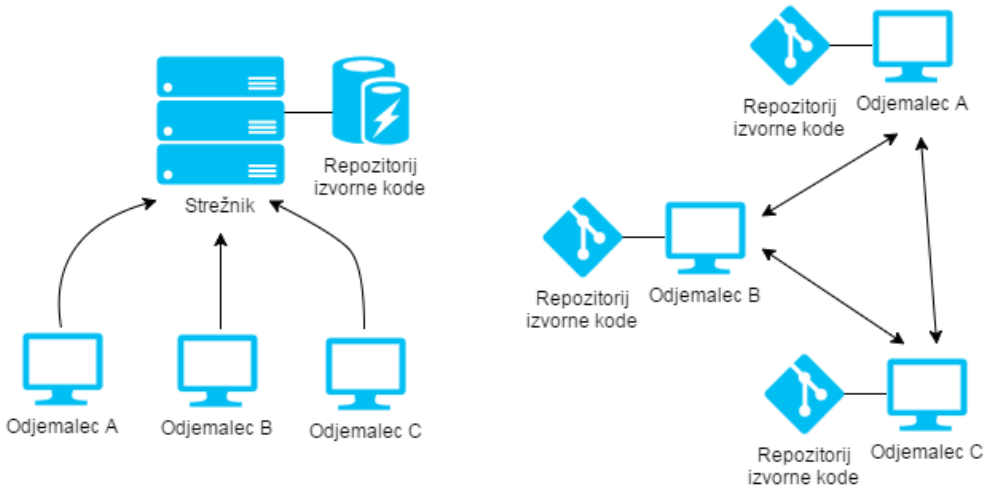
Pomembno je varno hranjenje izvorne kode na neodvisni lokaciji z varnostno kopijo. Ta ukrep je ključnega pomena, saj je izvorna koda rezultat dela celotne projektne ekipe.

Sistemi za upravljanje z izvorno kodo omogočajo razvijalcem projektne ekipe vzporedno delo na projektni izvorni kodi, brez medsebojnega oviranja. Sistemi implementirajo mehanizme za varno (pravilno zaporedje glede na časovne žige) združevanje delov izvorne kode. Če dva razvijalca implementirata spremembo na istem delu izvorne kode, ki jo sistem za upravljanje z izvorno kodo s svojimi mehanizmi ni uspel združiti pride do konflikta sprememb (*angl. merge conflict*). Reševanje takega stanja zahteva uporabnikov poseg [26].

Osnovni namen sistemov za upravljanje z izvorno kodo je od njihovega nastanka enak, močno pa so se spremenili pogoji v katerih ekipe razvijalcev

opravljajo svoje delo. Pogosto projektna ekipa ne sedi več v isti pisarni, ampak je razkropljena po več državah v različnih časovnih pasovih.

Glede na način hranjenja osrednje kopije izvorne kode se sistemi delijo v dve skupini: **centralizirani** in **porazdeljeni**. Razliko v načinu delovanja lahko vidimo na sliki 3.1.



Slika 3.1: Centralizirani (levo) in porazdeljeni (desno) sistemi za upravljanje z izvorno kodo.

Med **centralizirane sisteme** prištevamo nekoliko starejše sisteme – Concurent Versions System [27] ali Subversion [28]. Osnova dela s takimi sistemi je model strežnik – odjemalec. Uporabnik s centralnega strežnika prenese izvorno kodo, implementira spremembe in spremembo izvorne kode vrne na centralni strežnik. Spremenjena izvorna koda je nato vidna tudi ostalim uporabnikom. Osrednja kopija izvorne kode je vedno shranjena na centralnem strežniku.

Najbolj izpostavljeni predstavniki **porazdeljenih sistemov** za upravljanje izvorne kode so sistemi Git [29], Mercurial [30] in Bazaar [31]. Porazdeljenost sistemi dosežejo tako, da si uporabnik prenese (*angl. clone*) celotno zgodovino sprememb izvorne kode (tudi vse metapodatke) na svoj računalnik. Na tak način nastanejo kopije celotne zgodovine sprememb izvorne kode in

tako ni potrebe po centralnem strežniku. Zaradi lokalne kopije stanja sistema lahko uporabnik vse spremembe opravi brez omrežne povezave (omrežno povezavo potrebuje le pri sinhronizacije sprememb z izvorom) [32, 33].

Izbira med centraliziranim ali porazdeljenim sistemom za upravljanje izvorne kode je dandanes preprosta. Porazdeljeni sistemi so z mehanizmi za preprosto združevanje vej brez konfliktov, delovanja brez neprestane povezave v omrežje, hitrostjo in načinom dela, ki ne zahteva osrednjega strežnika izrinili centralizirane sisteme.

Čeprav porazdeljeni sistemi za upravljanje z izvorno kodo odpravljajo potrebo po osrednjem strežniku v praksi temu ponavadi ni tako. Ob začetku implementacije programske rešitve določimo osrednji strežnik projekta na katerega razvojniki pošiljajo svoje spremembe.

3.4 Vejitve in načini dela z izvorno kodo

Pojem veja (*angl. branch*) opisuje časovno zaporedje sprememb izvorne kode programske rešitve. Vzporedna oz. stranska veja (*angl. child branch*), ki nastane kot rezultat vejitve je dvojnik izvorne (glavne) veje (*angl. parent branch*). Spremembe se na vzporedni veji vodijo ločeno in jih lahko združimo nazaj v glavno vejo. Vzporedne veje uporabljamo za vzporedno implementacijo novih funkcionalnosti, razne optimizacije ali večje prestrukturiranje izvorne kode programske rešitve.

Če želimo kakovostno upravljati projektno izvorno kodo, moramo izbrati najprimernejši način dela (*angl. workflow*). Z načinom dela opredelimo uporabo vej in zaporedje korakov pri ravnanju z izvorno kodo. S tem omogočimo sočasno delo več razvijalcem na isti izvorni kodi. Nekateri delovni tokovi tudi preprečujejo nepotrebne konflikte pri združevanju programske kode. Izbira pristopa (načina vejitev in združevanja sprememb) je odvisna od števila razvijalcev, njihove lokacije, količine in frekvence sprememb.

V nadaljevanju si oglejmo nekaj pristopov in z njimi povezanih vejitev izvorne kode.

3.4.1 Centraliziran pristop

Pri uporabi centraliziranega pristopa (*angl. centralized workflow*) vsi razvijalci implementirajo spremembe izvorne kode na glavni veji. Ob prekrivanju sprememb izvorne kode pogosto pride do konfliktov ob združevanju. Mehanizmi sistemov za upravljanje z izvorno kodo praviloma preprečujejo, da bi razvijalec oddal svoje spremembe in pred tem ne bi posodobil svoje kopije izvorne kode z zadnjim stanjem na centralnem strežniku [34].

Tak način dela je pogost pri delu s centraliziranimi sistemi za upravljanje izvorne kode in je pogojno primeren za manjše ekipe, ki delujejo na isti lokaciji.

3.4.2 Pristop z uporabo namenskih vej

Če se odločimo za ta pristop, se vsaka nova funkcionalnost implementira v ločeni veji (*angl. feature branch*). Vejo lahko posredujemo na osrednji strežnik in tako delimo spremembe tudi z ostalimi razvijalci v ekipi. Izbira takega način dela prinese veliko svobode pri odločitvi, kdaj bomo katero od funkcionalnosti vključili v programsko rešitev. Pristop z uporabo vej (*angl. feature branch workflow*) nadgrajuje centraliziran pristop in zmanjša število nepotrebnih konfliktov ob združevanju izvorne kode [35].

3.4.3 Pristop Gitflow

Pristop poimenovan Gitflow (*angl. Gitflow workflow*) je izredno robusten, namenjen upravljanju velikih projektov, ki odlično sovpada z lahkotnostjo uporabe vej v sistemu Git [29].

Pristop natančno opredeli namen in zaporedje uporabe vej. Poleg uporabe vej za implementacijo novih funkcionalnosti opredeli tudi veje za zaključeno verzijo (*angl. release branch*), razvoj (*angl. development branch*), nujne popravke (*angl. hotfix branch*) in glavno vejo (*angl. master branch*), ki naj bi vedno odražala zadnje zaključeno stanje izvorne kode (*angl. production state*) [36]. Delovni tok je predstavil Vincent Driessen v svojem spletnem

zapisu leta 2010 [37].

3.4.4 Primerjava pristopov

Za začetek povejmo, da je predstavljene pristope moč uporabljati neodvisno od izbranega sistema za upravljanje izvorne kode. Izbiro najbolj primernega pristopa bomo osnovali na številu razvojnikov ter zahtevnosti in zrelosti programske rešitve.

Prvi, centraliziran pristop, je primeren le, če razvijamo preprosto programsko rešitev z malim številom (največ 3) razvojnikov. V izogiba konfliktom pri združevanju izvorne kode je najbolje, da razvojniki jasno uskladijo protokol oddaje sprememb.

Uporaba namenskih vej je nujna, če želimo omogočiti vzporedno delo srednje veliki skupini razvojnikov. Razvojniki si razdelijo neizvedene funkcionalnosti programske rešitve in vsako od njih razvijejo v namenski veji. Veje so združujejo v glavno vejo pod nadzorom izkušenega razvojnika, ki rešuje tudi morebitne konflikte izvorne kode.

Pristop Gitflow nadgradi namensko uporabo vej z natančno opredelitvijo uporabe in postopka združevanja vej. Omogoča vzporedno delo več razvojnikov na aktualni in prihodnji verziji programske rešitve. Po izkušnjah avtorja najboljša izbira za programske rešitve, ki so dosegle nivo zrelosti in sklope nadgradenj, združujejo v vnaprej opredeljene verzije. Na primer programske rešitve v fazi vzdrževanja. Pristop doseže najboljši učinek ob uporabi distribuiranega sistema za upravljanje izvorne kode.

3.5 Integracijski strežnik

Integracijski strežnik (*angl. build server, continuous integration server*) je osrednje orodje za avtomatizacijo razvoja programske opreme. Praviloma je nameščen na namenski in neodvisen strežnik (ali v večjih organizacijah, gručo strežnikov) z namenom prevajanja izvorne kode, izvajanja testov, gradnje programske opreme in orkestracije pomožnih procesov [39].

Z uporabo integracijskega strežnika pridobimo neodvisno okolje, ki ga lahko po potrebi nastavljamo v različna stanja, v katerem naj bi delovala naša programska rešitev (npr. prevajanje izvorne kode z več različnimi prevajalniki, testiranje delovanja z malo pomnilnika, testiranje delovanja v različnih operacijskih sistemih). Orkestracija pomožnih procesov (npr. obremenitveni test določenega aplikacijskega okolja ali statična analiza izvorne kode) je preprosta, saj orodje omogoča zagon procesov na časovno periodo, ob izpolnjenih določenih pogojih ali ročno. Za vse opisane postopke lahko opredelimo tudi različne načine obveščanja.

Brez uporabe integracijskega strežnika kot osrednjega orodja, si tudi ni moč predstavljati principa sprotne integracije (*angl.continuous integration*). Princip predpostavlja, da razvijalci pogosto (tudi večkrat na dan) oddajajo svoje spremembe izvorne kode v sistem za upravljanje izvorne kode. Na podlagi sprememb integracijski strežnik spremenjeno kodo prevede, sproži ustrezne teste in uspešno zgrajeno programsko rešitev shrani. V primeru neuspeha vpletene obvesti o napaki. Postopek združuje manjše spremembe izvorne kode pogosteje in s tem poskrbi, da se napake sproti odkrivajo in odpravljajo. Postopek se lahko nadgradi z izvedbo statične analize izvorne kode in avtomatsko namestitvijo programske rešitve [43].

3.6 Sistem za hranjenje programskih komponent

Če izvorno programsko kodo hranimo v sistemu za upravljanje izvorne kode, že prevedene in zgrajene programske komponente odlagamo v sistem za hranjenje programskih komponent (*angl. build artifacts repository*). Sistem za hranjenje programskih komponent je praviloma nameščen na samostojnem strežniku z večjo količino namenskega diskovnega prostora.

Na strežniku hranimo urejene in z verzijami označene skupne programske knjižnice in javne programske knjižnice, ki jih uporabljamo v lastnih programskih rešitvah. Na strežnike odlagamo tudi komponente naše programske

rešitve (vmesne in končne verzije). Knjižnice so tako na voljo za souporabo vsem članom razvojne ekipe ali več ekipam v večjih organizacijah. V večjih organizacijah ima strežnik tudi posredniško vlogo (*angl. proxy server*), saj se programske knjižnice (po tem ko so prvič zahtevane) na njem ohranijo in so ob ponovni zahtevi posredovane neposredno z njega in ne iz medmrežja.

Sistem nudi grajene programske komponente tudi drugim orodjem (npr. orodju za nameščanje programskih rešitev v izvajalna okolja). Osrednji strežnik nudi programske knjižnice na zahtevo orodij za upravljanje s projekti ali preko uporabniškega vmesnika.

3.7 Avtomatski pregled izvorne kode

Statična analiza programske kode je del avtomatskega pregleda programske kode (*angl. code review*). Analizo sestavlja nabor algoritmov in tehnik, ki se izvedejo na neprevedeni izvorni kodi. Rezultat analize izpostavi potencialne napake in uporabo slabih praks ob implementaciji programske rešitve, ki jih primerja z naborom vnaprej definiranih pravil [38].

Rezultate statične analize lahko v grobem razdelimo v tri skupine:

- **napake v programski kodi** (logične napake, varnostne pomanjkljivosti, uporaba slabih praks pri implementaciji),
- **oblika zapisa programske kode** (neustrezna preglednost sintakse izvorne kode),
- **metrike** (število vrstic, razredov, komponent; stopnja dokumentiranosti, zahtevnost programske kode, soodvisnosti modulov).

V prvi vrsti so izsledki analize namenjeni opozarjanju na napake in zagotavljanju boljše splošne kakovosti programske izvorne kode. Ne gre pa zanemariti tudi učne narave teh analiz za mladega neizkušenega razvojnika.

Za statično analizo uporabljamo orodja, ki jih je preprosto vključiti v avtomatiziran proces. Analiza programske kode se v našem procesu lahko

sproži na integracijskem strežniku ob spremembi izvorne kode, na določeno časovno obdobje ali ročno.

3.8 Osebni pregled programske kode

Pregled programske kode s strani drugih razvijalcev (*angl. peer code review*) je prvina, ki je pogosto zanemarjena (še posebej v majhnih projektnih ekipah). V namen pregleda se v organizaciji oblikuje posebna ekipa pregledovalcev, praviloma izkušenejših razvojnikov različnih specializacij. Če velikost organizacije tega ne dopušča, za pregled izvorne kode skrbi vodja razvojne ekipe, oz. pod določenimi pogoji zunanji svetovalec.

Na tedenskem nivoju razvojniki v pregled oddajo dele izvorne kode. Priporočljivo je, da deli niso večji kot 400 vrstic in da si pregledovalec za pregled kode ne vzame več kot 1 uro svojega delovnega časa. Med pregledom ugotavljamo ali izvorna koda ustreza standardom, priporočilom, dobrim praksam oz. ali vsebuje logične napake ali odstopanja od uporabniških zahtev, ki niso bile odkrite z statično analizo izvorne kode.

Pogosto se izkaže, da se znotraj iste ekipe ponavljajo enake napake. V ta namen pripravimo kontrolni seznam, ki vsebuje opozorila na najpogostejše pomanjkljivosti. Seznam je v pomoč pregledovalcem in razvojnikom pri implementaciji.

Pomanjkljivosti, ki jih ugotovimo tekom pregleda, je potrebno odpraviti. Za to se priporoča uporaba namenskih orodij (platfom) za pregled programske kode. Preko teh orodij pregledovalci napake označijo, avtorji vodijo dialog s pregledovalci in potrdijo popravke napak. Uporaba orodij je tudi način, da opomb pregledov ne pozabimo, saj te napake odkrijemo pred rednim testiranjem in jih kot take ponavadi nismo vključili v sistem sledenja sprememb (*angl. issue tracking*) [40].

Postopek je odličen način za uvajanje novih članov razvojne ekipe. Pri metodi Scrum je opravljen pregled pogosto tudi kriterij za zaključek uporabniške zgodbe [41].

3.9 Namestitev programske rešitve v kontrolirano okolje

Zaključni korak je predaja programske rešitve v kontrolirano okolje. Kontrolirano okolje je ločeno, neodvisno aplikacijsko okolje postavljeno in nastavljeno z določenim namenom. Če izvzamemo produkcijsko okolje, kamor nameščamo končne verzije programske rešitve in okolje na razvijalčevem računalniku, je osrednji namen ostalih okolij testiranje naše programske rešitve.

Če želimo omogočiti vzporedne testne aktivnosti, je priporočljivo imeti vsaj štiri okolja:

- **razvojno** (*angl. development environment*)
  V to okolje sproti nameščamo še netestirane funkcionalnosti. Namenjeno je hitrim pregledom delovanja funkcionalnosti, preden so predstavljene naročniku. Praviloma okolje ne ustreza vsem dogovorjenim tehničnim karakteristikam, podatki so pogosto izmišljeni oz. nastajajo sproti, povezano je na razvojno integracijsko okolje.

- **testno** (*angl. acceptance environment*)
  V testno okolje nameščamo zaključene novorazvite funkcionalnosti. Okolje je namenjeno naročniku, ki potrdi ali zavrne spremembo. Povezano je na testno integracijsko okolje.

- **predprodukcijsko** (*angl. stage environment*)
  V predprodukcijsko okolje nameščamo verzije programske rešitve, ki so testirane in potrjene s strani naročnika, podatki so identični produkcijskim (pogosto anonimizirani). Povezano je na predprodukcijska integracijska okolja.

  Ker je okolje po tehničnih karakteristikah in arhitekturni zasnovi preslikava produkcijskega okolja, se okolje uporablja za performančne (*angl. performance testing*), obremenitvene (*angl. load testing*) in penetracijske (*angl. penetration testing*) teste.

- **produkcijsko** (*angl. production environment*)
  Okolje je namenjeno namestitvi končnih verzij naše programske rešitve in končnim uporabnikom. Okolje ima dogovorjene tehnični karakteristike, vitalni znaki gradnikov okolja (strežniki, podatkovne baze, zunanji viri) so ustrezno nadzorovani (*angl. monitoring*). Okolje ustreza varnostnim zahtevam (omejeni in zabeleženi dostopi, varnostna kopiranja podatkov in redne nadgradnje).

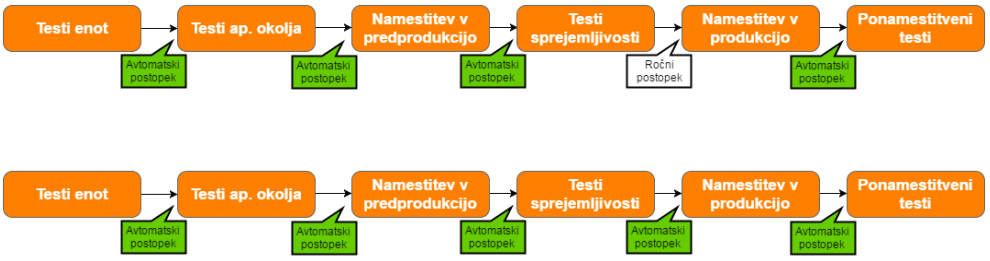
Pogosto se v ta nabor doda tudi namensko okolje za zagotavljanje kakovosti (*angl. quality assurance environment*), ki po namembnosti sledi testnemu okolju. V tem okolju preverimo ali novonastale funkcionalnosti ne spremenijo (okvarijo) delovanja že obstoječih funkcionalnosti programske rešitve. Z regresijskimi testi (*angl. regression testing*) se preveri delovanje vseh glavnih funkcionalnosti programske rešitve pred prehodom v (pred)produkcijsko okolje.

Postopek nameščanja programske rešitve je zaporedje korakov, ki jih je potrebno opraviti ob vsaki namestitvi. Če ob nameščanju ne obstajajo izredni pogoji (npr. zaradi varnosti razlogov naročnik, po pripravljenih navodilih, namešča programsko rešitev sam) ga je moč popolnoma avtomatizirati.

Monotona zaporedja korakov v povezavi s hitro prilagodljivim številom okolij z veliko strežniki, združenimi v grozde, so prinesla nov tip orodij – orodja ARA (*angl. Application Release Automation*). V teh orodjih lahko oblikujemo postopek namestitve z dodanimi preverjanji uspešnosti korakov in mehanizmi, ki se aktivirajo ob neuspehu postopka (npr. povratek na zadnje delujoče stanje in obveščanje o napakah). Orodja so postopke nameščanja naredila manj občutljiva za človeške napake, ki so zelo pogost vzrok okvarjenih namestitev. Avtomatske namestitve so lahko ponovljive, prilagodljive, ne zahtevajo tehničnega predznanja (zagon namestitvenega postopka je le potrditev v spletnem uporabniškem vmesniku) in omogočajo pogostejše namestitve [42].

Pogoste namestitve programske rešitve so ena od značilnosti projekta vodenega po agilnim metodologijah. Avtomatizacija namestitev (in korakov v razvojnem procesu) je izoblikovala dva principa: **sprotna dostava** (*angl. continuous delivery*) in **sprotna namestitev** (*angl. continuous deployment*) programske opreme.

Prvi princip narekuje avtomatsko namestitev vsake spremembe izvorne kode do predprodukcijskega okolja. Predvideva tudi, da je programska rešitev v vsakem trenutku pripravljena za namestitev v (pred)produkcijsko okolje in poudarja strogo testiranje vsakega koraka v razvojno-namestitvenem procesu. Poslovna odločitev sproži namestitev spremembe tudi v produkcijsko okolje. Drugi princip je nadgradnja prvega in predvideva nepretrgane namestitve vsake spremembe v produkcijsko okolje [44, 45]. Oba principa prikazuje slika 3.2.



Slika 3.2: Princip sprotne dostave (zgoraj) in princip sprotne namestitve (spodaj) [44].

3.10 Post-mortem analiza

Ob zaključku projekta oz. prehodu projekta v vzdrževalno fazo opravimo post-mortem analizo. Na posebnem (daljšem) sestanku se srečajo vsi vpleteni v projekt (naročnik, razvojniki in sponzorji projekta). Skupaj poskušajo izpostaviti, kaj je bilo tekom trajanja projekta narejeno dobro, kaj bi bilo lahko narejeno drugače in kje so bile narejene napake. Ocenimo vse vidike projekta: vodenje, načrtovanje (pravilnost časovne ocene), viri, razvoj,

orodja, komunikacija, itd. Rezultat sestanka je dokument, ki služi izboljšavi delovnih procesov in pristopa k naslednjim projektom [46].

Čeprav gre za tipični sestavni del projektnega vodenja, s katerim zmanjšamo tveganje v prihodnje, na analizo pogosto pozabimo oz. ta ne prinese oprijemljivih sklepov. Sklepi in ugotovitve pogosto ne spremenijo načina dela v organizaciji. Dober pristop k sprotnem analiziranju stanja je vpeljala metodologija Scrum, ki je prvine post-mortem analize zajela v retrospektivnih sestankih.

Poglavje 4

Povezana uporaba priporočenih prvin

V poglavju bomo predstavili način souporabe predlaganih prvin na hipotetičnem primeru razvoja programske rešitve. Preko primera bomo izbiro predlagane uporabe prvin in orodij tudi utemeljili.

Uporaba predlaganih prvin in orodij je dovolj splošna, da zadosti potrebam, ki se pojavljajo na majhnih, srednjih in velikih projektih. Orodja, ki so uporabljena v primeru, ne predstavljajo izdatnega stroška za organizacijo, kjer želimo tak proces vpeljati.

4.1 Primer uporabe prvin

V primeru uporabe priporočenih prvin bo prikazan način uporabe večine prvin, ki smo jih opisali na predhodnih straneh. Uporabljenih prvine je moč podpreti z različnimi orodji, vendar smo za primer izbrali odprtokodna (z nekaj izjemami) orodja in platforme temelječe na programskem jeziku Java [47].

Cilj hipotetičnega projekta je izdelati spletno trgovino za znanega naročnika. Spletna trgovina omogoča pregledovanje in nakup izdelkov, ki jih naročnik nudi. Katalog izdelkov je dosegljiv na naročnikovih zalednih siste-

mih preko spletnih storitev. Na enak način je dosegljiv tudi njegov naročilni sistem (podatki o zalogah, upravljanje z naročili, plačevanje). Podrobnosti o programski rešitvi za predstavitev našega namena niso pomembne, saj uporaba prvin ni neposredno vezana na podrobne zahteve. Naš namen je prikaz uporabe prvin na način, ki prinaša, po našem mnenju, najboljše rezultate.

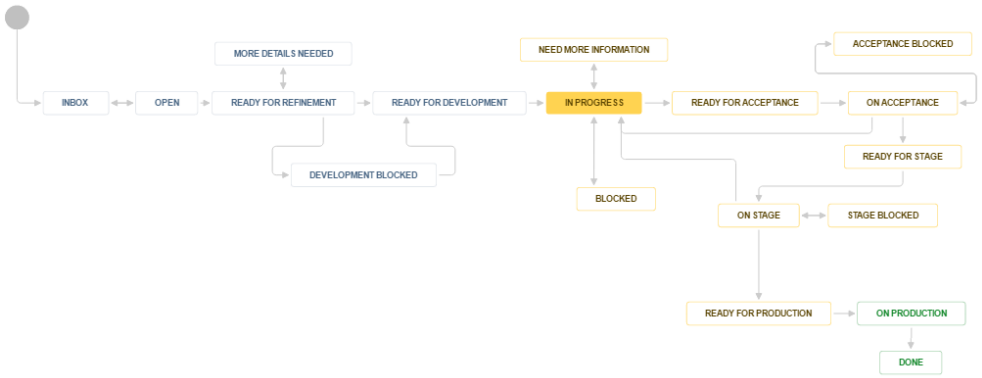
4.1.1 Projektno okolje

Značilnost spletnih trgovin so pogoste nadgradnje, ki izhajajo iz neprestanega prilagajanja kupcem in želje po konkurenčni prednosti. Zaradi zavedanja o pogostih spremembah, slabega tehničnega znanja naročnika in velike možnosti nepopolnih opredelitve zahtev, je bila izbrana agilna metodologija vodenja projekta - **Scrum**. Izbrana metodologija bo omogočala postopno nadgradnjo spletne trgovine in s tem naročniku hitro prilagajanje zahtevam na tržišču. Scrum bomo vodili preko platforme **Atlassian JIRA** [49]. Platforma nam bo v pomoč tudi pri spremljanju napredka, razdeljevanju nalog razvojnikom in komuniciranju z naročnikom.

V dokumentu strežniške arhitekture smo opredelili štiri okolja: **razvojno** (nameščamo vmesne različice programske rešitve, ki so namenjene hitremu pregledu delovanja), **testno** (naročnikova testna ekipa preverja in potrjuje delovanje razvitih nadgradenj), **predprodukcijsko** (namenjeno testiranjem v okolju, ki je primerljivo s produkcijskim, ter regresijskim in performančnim testom), ter **produkcijsko**.

Na podlagi okolij smo v platformi JIRA opredelili življenjski cikel posamezne zahteve. Status zahteve nam bo sporočal stanje zahteve (npr. v testiranju na testnem okolju - *on acceptance*, testiran v predprodukcijske okolju in pripravljen za namestitev v produkcijo - *ready for production*). Uporabljen življenjski cikel prikazuje slika 4.1. Statusi, obarvani z modro barvo opisujejo obdobje, ko je zahteva usklajevana. Statusi rumene barve označujejo izvedbo in proces testiranja. Zeleni statusi so končna stanja.

Spletno trgovino bomo razvili v okolju **Java EE** [48]. Programsko okolje smo izbrali zaradi njegove razširjenosti, zrelosti in naročnika, ki želi naknadno



Slika 4.1: Življenjski cikel zahteve v platformi Atlassian JIRA.

prevzeti vzdrževanje programske rešitve. V luči te odločitve bomo uporabili strežnik **Red Hat WildFly** [50], ki je trenutno eden najbolj razširjenih in dostopnih aplikacijskih strežnikov [51]. Projektno programsko kodo bomo upravljali z orodjem za upravljanje projektov **Apache Maven** [25]. Izvorna koda projekta bo shranjena na osrednjem strežniku z nameščenimi sistemom za upravljanje izvorne kode **Git** [29]. Kakovost in metrike izvorne kode bomo spremljali v platformah **SonarQube** [52] in **JetBrains Upsource** [53], ki ga bomo uporabljali za izvajanje ročnih pregledov izvorne kode.

Integracijski strežnik **Jenkins** [54] bo skrbel za avtomatizacijo in orkestracijo celotnega razvojnega procesa naše programske rešitve. Na strežnik **Sonatype Nexus** [55] bomo odlagali naše zgrajene programske komponente (označene z unikatno oznako verzije). Strežnik bo tudi posrednik za skupne programske knjižnice celotni razvojni ekipi.

Zadnji korak, namestitev programske rešitve, bomo opravljali z orodjem **Ansible** [56]. Z njegovo pomočjo bomo oblikovali postopek namestitive brez izpada delovanja, ki bi ga občutil obiskovalec spletne trgovine. Uporabniški vmesnik za zagon namestitev preko skript Ansible bo platforma **Semaphore** [57].

4.1.2 Način dela

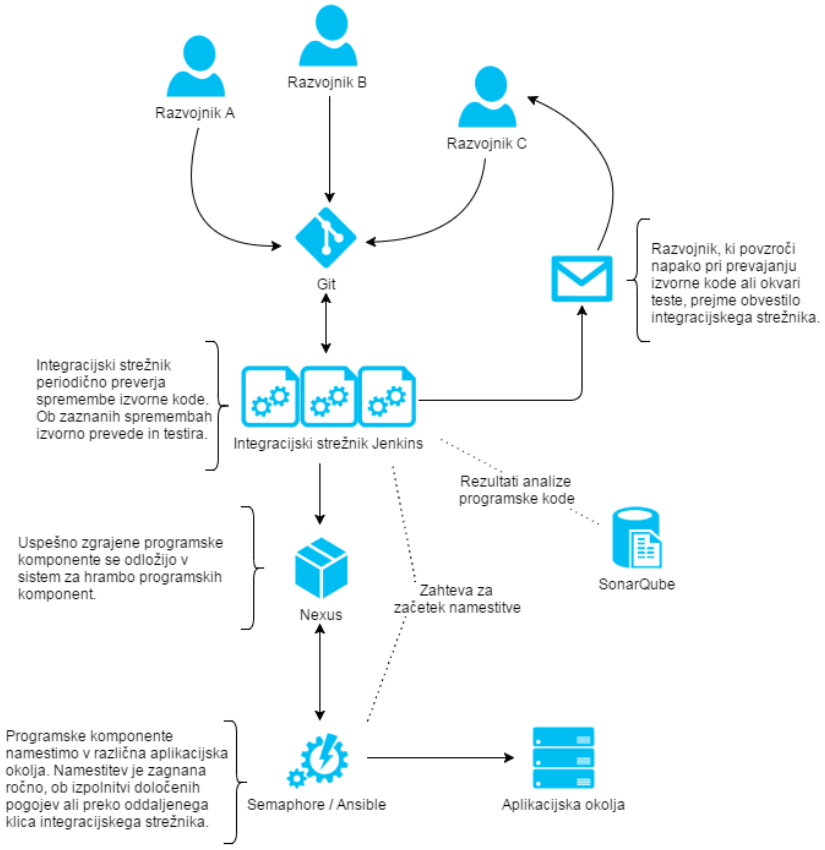
V nadaljevanju bomo opisali sled spremembe izvorne kode od njenega nastanka do namestitve v produkcijsko okolje. Hkrati bomo predstavili vsakodnevna opravila razvojnika, ki sodeluje pri nastanku naše spletne trgovine.

Razvojnik izbere uporabniških zgodbo, ki je vključena v tek. Z izbiro se uporabniški zgodbi, zahtevku (*angl. issue*), spremeni status. Spremenjeni status nakazuje na to, da se implementacija uporabniške zgodbe začela izvajati. Spremembo izvorne kode razvojnik opravi na ločeni veji (poimenovano po oznaki zahtevka) in jo ob zaključku združi z glavno razvojno vejo. Spremembe združene razvojne veje posreduje na osrednji strežnik za upravljanje z izvorno kodo. Razvojnik spremeni status zahtevka, ki sedaj nakazuje, da so zahteve uporabniške zgodbe implementirane in pripravljene na testiranje.

Integracijski strežnik periodično preverja, ali je na osrednji razvojni veji prišlo do sprememb izvorne kode. Če do spremembe pride zažene prevajanje, testiranje in grajenje programskih komponent. Če so postopki uspešni, programske komponente shrani v sistem za hranjenje programskih komponent in sproži analizo programske kode. V primeru neuspešnega poskusa pošlje sporočilo o napaki. Celoten postopek je predstavljen na diagramu 4.2.

Ko razvojna ekipa izvede vse dogovorjene uporabniške zgodbe, ki sodijo v okvir verzije, se začne postopek priprave verzije naših programskih komponent. Sledimo pristopu **Gitflow** in iz zadnjega stanja razvojne veje pripravimo novo vejo. Nova verzija zajema stanje izvorne kode, ki je primerno za namestitev v testno okolje. Postopek izgradnje programske rešitve je identičen izgradnji programske rešitve na razvojni veji. Razvoj novih funkcionalnosti se nemoteno nadaljuje na razvojni veji.

Verzijo programske rešitve namestimo na testno okolje. Med avtomatiziranim postopkom s sistema za hranjene programskih komponent pridobimo komponente programske rešitve in jih prenesemo na aplikacijske strežnike. Strežnike posodabljamo z novo verzijo postopoma, po vsaki namestitvi preverimo, ali je bila namestitve uspešna. Testiranje uspešnosti namestitve je



Slika 4.2: Prikaz (delno) avtomatiziranega postopka razvoja programske opreme.

del postopka namestitve. Namestitev programske rešitve v razvojno okolje povežemo v enoten postopek, če preko integracijskega strežnika po končani gradnji programskih komponent sprožimo tudi postopek namestitve. Za ostala okolja postopek prožimo ročno, saj čakamo naročnikovo (poslovno) potrditev.

Naročnikova testna ekipa na testnem okolju preverja pravilnost delovanja in ustreznost nameščene verzije programske rešitve. Ob odkriti napaki popravek odpravimo na veji v katerih vodimo verzijo, ter ga združimo tudi z razvojno vejo. Ko so vse uporabniške zgodbe preverjene, ponovimo namestitev na predprodukcijsko okolje (ustrezno se spremenijo statusi zahtevkov).

Na predprodukcijskem okolju se izvedejo regresijski testi in specifična testiranja povezana s pravimi podatki. Če je naročnik s testiranim zadovoljen se zgradijo končne programske komponente. Princip Gitflow predvideva prenos stanja izvorne kode v veji verzije v glavno vejo. Končne programske komponente namestimo v produkcijsko okolje po dogovorjenem postopku.

Razvojna ekipa tedensko pregleduje novonastalo izvorno kodo na razvojni veji. Skupaj s podatki analize programske kode in metrikami lahko enostavno oceni kakovost izvorne kode.

4.2 Prednosti uporabe prvin

Predstavljene prvine smo izpostavili kot model procesa razvoja programske opreme. Opisane prvine lahko (v večji ali manjši meri) vpeljemo v vsako tehnološko podjetje, ki se ukvarja z razvojem programske opreme. Izbrane so bile na podlagi:

- trendov razvoja programske opreme,
- želje po avtomatizaciji postopkov,
- zagotavljanja kakovosti programske opreme,
- avtorjevih izkušenj z delom v tehnoloških podjetjih.

Izbrane prvine so najprimernejše za uporabo v malih in srednje velikih podjetjih, ki želijo zagotoviti kakovost dela in programskih rešitev, a hkrati nimajo sredstev, ki bi jih namenjale za vpeljavo standardov ali najem zunanjih svetovalcev. Prvine lahko vpeljemo postopoma (ali le izbrane) in sproti spremljamo spremembe v kulturi podjetja in odnos zaposlenih do kakovosti. Po naših izkušnjah uporaba prvin zagotavlja poceni in kakovostno programsko rešitev, izvedeno v predvidenih časovnih rokih.

Vse omenjene prvine lahko razdelimo v tri večje sklope: **preglednost**, **avtomatizacija** in **kakovost**. V nadaljevanju si podrobneje oglejmo vsak sklop.

Uporaba sistema v sistem sledenja zahtevam v podjetju pogosto prinese negodovanje, saj so razvojniki morda vajeni prejemati navodila za izvedbo določene funkcionalnosti preko elektronske pošte ali, še slabše, preko neposrednega ustnega dogovora. Uporabniška zahteva z zapisanimi navodili, ki so premišljena in postavljena v kontekst, olajša delo vsem vpletenim. Razvojnik je natančno seznanjen z vsem potrebnimi informacijami za izvedbo spremembe, testna ekipa po opisanem razume opravljeno spremembo in projektni vodja lažje spremlja napredek projekta. Pozitivno se spremeni tudi delovna kultura, saj razvojniki zavračajo slabo opredeljene naloge, kar vodi do bolje opredeljenih zahtev v prihodnje. Ker so vse zahteve zbrane na enem mestu, je enostavno spremljati napredek projekta in ujemanje s predvideno časovnico.

Po daljši uporabi sistem nehote postane tudi baza znanja za lažje razumevanje vzrokov in odločitev, ki so vodili do novo odkrite napake. Uporaba takega sistema je za podjetje pozitivna, saj preko njega lahko spremlja obremenjenost posameznikov, pravilnost časovnih ocen in morebitna tveganja.

Integracijski strežnik in princip sprotne integracije sta drugi dve prvini, ki močno posežeta v način dela razvojne ekipe. Z uporabo principa sprotne integracije razvojnike spodbujamo, k pogostejšim oddajanjem izvorne kode. S tem pristopom zgodaj odkrijemo napake združevanja in jih tudi takoj odpravimo. Preprečimo stanje, ki ga opišemo z izrazom "pekel združevanja"(*angl. merge hell*). Izraz opisuje stanje, ko več razvojnikov (praviloma dan pred predajo programske rešitve naročniku) želi vso novonastalo izvorno kodo združiti v delujočo programsko rešitev. Ponavadi neuspešno.

S pomočjo integracijskega strežnika lahko avtomatiziramo monotone postopke (npr. namestitev programske rešitve) in orkestriramo zaporedje dogodkov (npr. analiza programske kode). Z avtomatizacijo iz postopkov odstranimo t.i. človeški faktor, ki je pogost vzrok napak. Postopki postanejo lahko ponovljivi in nadzorovani. Zmanjšamo tudi količino napak v postopkih, kar pomeni, da se ekipa osredotoči na izvedbo novih funkcionalnosti.

Glavna vloga integracijskega strežnika je preverjanje delovanja združene izvorne kode. Pravilnost delovanja programske rešitve se preveri s programskimi testi (test enot, integracijski test, test uporabniškega vmesnika). Sodeč po izkušnjah je pisanje testov zelo nepriljubljeno opravilo in razvojnike je zanj težko navdušiti. Razlogi za odsotnost testov so: kratki projektni roki, preobsežna in napačno strukturirana izvorna koda ali neznanje. Tudi vodstvo podjetja pogosto ne vidi prednosti testov, saj pisanje programskih testov podaljša čas izvedbe.

Vendar pozitivnih vplivov programskih testov ne gre zanemariti. Z uporabo testov enot pravzaprav potrdimo, da je naša programska koda pravilno deluje. Nabor testov služi kot varovalka ob spremembah izvorne kode in prestreže napake, ki bi kasneje povzročile nejevoljo uporabnikov. Veliko lažje je tudi večje prestrukturiranje programske kode, saj s testi potrdimo enako delovanje kot pred posegom.

Dober način, ki spodbudi zavedanje o kakovosti programskih rešitev in pisanje programskih testov v podjetju, je uporaba platforme za analizo kakovosti programske kode.

Preko omenjene platforme spremljamo analize projektov, ki jih izvajamo v podjetju. Rezultati analiz posameznega projekta se osvežijo ob vsaki spremembi njegove programske kode. Rezultati so dostopni vsem, zato se pogosto dogodi, da projektne ekipe začnejo tekmovati, čigav projekt bo dosegel boljši rezultat. Vodstvu podjetja je informacija o kakovosti posameznega projekta v pomoč pri oceni tveganj napredka projekta. Če želimo naročnika navdušiti, mu lahko tedensko pošiljamo poročila o kakovosti oz. mu omogočimo dostop do platforme.

4.3 Učinki uporabe prvin

Potencialne učinke uporabe predstavljenih prvin bi (sodeč po osebnih izkušnjah) lahko razdelili v dve skupini.

Neposredni učinki, ki se kažejo s samo preglednostjo in sledljivostjo procesa razvoja programske opreme. Z rednim izvajanjem testov enot zagotavljamo, da programska rešitev pravilno deluje ne glede na spremembe. V tem postopku odkrijemo potencialne napake, ki bi lahko povzročile nepravilno delovanje v produkcijskem okolju. Odprava napak, ki smo jih odkrili tekom razvoja je za podjetje cenejša, kot naknadna analiza in odprava napake v produkcijskem okolju. Ker je dobršen del izgradnje in namestitve programskih rešitev avtomatski se projektne ekipe posvečajo rednemu projektnemu delu in ne monotonim opravilom. To pomeni, da projekt napreduje hitreje in v predvidenih finančnih okvirjih.

Med **posredne učinke** prištevamo spremembo kulture, ki na prvo mesto postavlja kakovost programske rešitve in procesa razvoja. Lažje obvladovanje potencialno tveganih projektov, saj predstavljen proces vsebuje nekaj kazalnikov, ki predčasno opozorijo na morebitne zaplete. Posredni učinek je tudi rast ugleda podjetja, ki postane prepoznavno kot podjetje s kakovostnim in jasnim pristopom k razvoju programske opreme. Pridobljen ugled predstavlja za podjetje tržno prednost.

Poglavje 5

Sklepne ugotovitve

V diplomskem delu smo poskušali predstaviti nekatere prvine kakovostnega procesa razvoja programske opreme. Izbrane prvine niso le tehnične (uporaba določenih orodij), ampak posegajo tudi v organizacijo dela (način vodenja uporabniških zahtev) pri razvoju programske rešitve.

Predstavili smo prelomne modele življenjskega cikla programske opreme, ki so navdahnili agilne metodologije. Nekaj pozornosti smo namenili tudi kakovosti programske opreme in predstavitvi pojmov, ki se vedno bolj pojavljata tudi v upraviteljskih krogih - tehnični dolg in SQALE.

Predstavljene prvine smo izbrali na podlagi lastnih izkušenj in opazovanj. Predstavitev prvin logično sledi vsakodnevnem delovnem toku razvojnika. Od vrstice zapisane v programskem jeziku, do namestitve spremembe, ki jo ta vrstica predstavlja v aplikacijskem okolju. Veliko poudarka smo namenili avtomatizaciji postopkov in jo predstavili preko hipotetičnega primera. Uporabljena orodja in povezava prvin zadostijo potrebam po zagotavljanju kakovosti v majhnem ali srednje velikem podjetju, ki se ukvarja z razvojem programske opreme.

Predstavili smo prednosti uporabe prvin in učinke, ki jih uporaba prvin prinaša. Največji dosežek, ki ga pri uvajanju omenjenih prvin lahko dosežemo je samoumevnost. Samoumevnost uporabe izpostavljenih prvin za projektne ekipe nakazuje na to, da so prvine popolnoma prešle v kulturo podjetja.

Predstavljen model prvin je neodvisen od tehnološke usmeritve in dovolj prilagodljiv, da je primeren za uporabo v široki paleti malih in srednje velikih podjetij. Če bi ga želeli prilagoditi tudi uporabi v velikih podjetjih, bi na seznam dodali tudi prvine, ki opredeljujejo upravljanje dokumentacije in komunikacijo med več ločenimi skupinami razvojnikov.

V vsaki fazi razvoja programske opreme, kot glavni ali stranski izdelek, nastane tudi dokumentacija. Dokumentacijo vseh faz je smotrno shranjevati na osrednjem mestu, ki omogoča preprosto pregledovanje, iskanje in kategorizacijo vsebin. Upravljanje dokumentacije je v velikih podjetjih pomembno, saj se razvojne skupine pogosto spreminjajo. S pametnim upravljanjem dokumentacije olajšamo prenos in zajezimo odtekanja znanja.

V delo na velikih projektih je običajno vključenih več manjših razvojnih skupin. Manjše razvojne skupine so bolj učinkovite in lažje za upravljanje. Težave se pogosto pojavijo, ko je potrebno aktivnosti več manjših skupin uskladiti za dosego skupnega cilja. Preproste aktivnosti, kot je na primer usklajena namestitev funkcionalnosti, se izkažejo za veliko logistično operacijo. Težava se pojavi tudi pri pretoku vsakodnevnih projektnih informacij med skupinami. Temu se najlažje izognemo s protokolom komunikacije med skupinami in vpeljavo vodij, ki skrbijo za usklajeno izvajanje zapletenejših postopkov.

Literatura

[1] What are the Software Development Life Cycle (SDLC) phases? [Online]. Dosegljivo: http://istqbexamcertification.com/what-are-the-software-development-life-cycle-sdlc-phases [Dostopano 23. 7. 2016].

[2] M. Fowler (2005) The New Methodology [Online]. Dosegljivo: http://www.martinfowler.com/articles/newMethodology.html [Dostopano 23. 7. 2016].

[3] What are the Software Development Models? [Online]. Dosegljivo: http://istqbexamcertification.com/what-are-the-software-development-models [Dostopano 23. 7. 2016].

[4] A. B. Cremers, S. Alda Software Development Process Models and their Impacts on Requirements Engineering [Online]. Dosegljivo: http://www.iai.uni-bonn.de/III/lehre/vorlesungen/SWT/RE05/slides/04_Software%20Development%20Process%20Models.pdf [Dostopano 24. 7. 2016].

[5] What is Waterfall model- advantages, disadvantages and when to use it? [Online]. Dosegljivo: http://istqbexamcertification.com/what-is-waterfall-model-advantages-disadvantages-and-when-to-use-it [Dostopano 23. 7. 2016].

[6] A. Kampuš (2002) Projektni management pri razvoju programskih rešitev [Online]. Dosegljivo:

http://www.cek.ef.uni-lj.si/magister/kampus60.pdf [Dostopano 23. 7. 2016].

[7] J. Strmec (2009) Prenova metode razvoja programskih rešitev v podjetju Hermes Softlab [Online]. Dosegljivo: http://www.cek.ef.uni-lj.si/magister/strmec336-B.pdf [Dostopano 23. 7. 2016].

[8] Rapid application development (The James Martin RAD method) [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Rapid_application_development [Dostopano 23. 7. 2016].

[9] What is RAD model- advantages, disadvantages and when to use it? [Online]. Dosegljivo: http://istqbexamcertification.com/what-is-rad-model-advantages-disadvantages-and-when-to-use-it [Dostopano 23. 7. 2016].

[10] K. S. Rubin *Essential Scrum: A Practical Guide to the Most Popular Agile Process*, Addison Wesley, 2013

[11] K. Waters (2007) What Is Agile? (10 Key Principles of Agile) [Online]. Dosegljivo: http://www.allaboutagile.com/what-is-agile-10-key-principles [Dostopano 23. 7. 2016].

[12] K. Waters (2007) 10 Good Reasons To Do Agile Development [Online]. Dosegljivo: http://www.allaboutagile.com/10-good-reasons-to-do-agile-development [Dostopano 23. 7. 2016].

[13] A brief introduction to Scrum [Online]. Dosegljivo: https://www.atlassian.com/agile/scrum [Dostopano 23. 7. 2016].

[14] A brief introduction to Kanban [Online]. Dosegljivo: https://www.atlassian.com/agile/kanban [Dostopano 23. 7. 2016].

[15] Scrum in agilne metode managementa [Online]. Dosegljivo: http://www.blazkos.com/scrum-in-agilne-metode-managementa.php [Dostopano 23. 7. 2016].

[16] N. Štrukelj (2016) Uvajanje metodologije Scrum na večjem projektu: študija primera in analiza faktorjev sprejemljivosti [Online]. Dosegljivo: https://repozitorij.uni-lj.si/IzpisGradiva.php?id=83775&lang=slv [Dostopano 23. 7. 2016].

[17] SQA Planning [Online]. Dosegljivo: http://www.exforsys.com/tutorials/sqa/sqa-planning.html [Dostopano 27. 7. 2016].

[18] D. Galin, *Software Quality Assurance: From theory to implementation*, str. 24–26, Addison Wesley, 2004

[19] ISO 9000 [Online]. Dosegljivo: https://en.wikipedia.org/wiki/ISO_9000 [Dostopano 23. 7. 2016].

[20] CMMI [Online]. Dosegljivo: http://cmmiinstitute.com [Dostopano 23. 7. 2016].

[21] S. Ramakrishnan (2013) Managing Technical Debt [Online]. Dosegljivo: https://www.scrumalliance.org/community/articles/2013/july/managing-technical-debt [Dostopano 24. 7. 2016].

[22] P. Kruchten, R. L. Nord, I. Ozkaya (2012) Technical Debt: From Metaphor to Theory and Practice [Online]. Dosegljivo: http://harkas.net/docs/Technical%20Debt-From%20Metaphor%20to%20Theory%20and%20Practice.pdf [Dostopano 24. 7. 2016].

[23] J.L. Letouzey (2011) The SQALE Method: Definition Document [Online]. Dosegljivo: http://www.sqale.org/wp-content/uploads/2011/05/SQALE-Method-EN-V0-09.pdf [Dostopano 24. 7. 2016].

[24] Apache Ant [Online]. Dosegljivo:
http://ant.apache.org [Dostopano 24. 7. 2016].

[25] Apache Maven [Online]. Dosegljivo:
https://maven.apache.org [Dostopano 24. 7. 2016].

[26] What is version control? [Online]. Dosegljivo:
https://www.atlassian.com/git/tutorials/what-is-version-control [Dostopano 24. 7. 2016].

[27] Concurrent Versions System - CVS [Online]. Dosegljivo:
http://savannah.nongnu.org/projects/cvs [Dostopano 24. 7. 2016].

[28] Subversion - SVN [Online]. Dosegljivo:
https://subversion.apache.org [Dostopano 24. 7. 2016].

[29] Git [Online]. Dosegljivo:
https://git-scm.com [Dostopano 24. 7. 2016].

[30] Mercurial [Online]. Dosegljivo:
https://www.mercurial-scm.org [Dostopano 24. 7. 2016].

[31] Bazzar [Online]. Dosegljivo:
http://bazaar.canonical.com/en [Dostopano 24. 7. 2016].

[32] G. Lionetti (2012) What is Version Control: Centralized vs. DVCS [Online]. Dosegljivo:
http://blogs.atlassian.com/2012/02/version-control-centralized-dvcs [Dostopano 24. 7. 2016].

[33] S. Chacon, B. Straub, *Pro Git: Everthing you need to know about Git*, str. 27–30, Apress, 2014 [Online]. Dosegljivo:
https://progit2.s3.amazonaws.com/en/2016-03-22-f3531/progit-en.1084.pdf [Dostopano 24. 7. 2016].

[34] Centralized Workflow [Online]. Dosegljivo: https://www.atlassian.com/git/tutorials/comparing-workflows/centralized-workflow [Dostopano 24. 7. 2016].

[35] Feature Branch Workflow [Online]. Dosegljivo: https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow [Dostopano 24. 7. 2016].

[36] Gitflow Workflow [Online]. Dosegljivo: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow [Dostopano 24. 7. 2016].

[37] V. Driessen (2010) A successful Git branching model [Online]. Dosegljivo: http://nvie.com/posts/a-successful-git-branching-model [Dostopano 24. 7. 2016].

[38] Static code analysis [Online]. http://www.viva64.com/en/t/0046 [Dostopano 24. 7. 2016].

[39] The Build Server: Your Project's Heart Monitor [Online] https://blog.codinghorror.com/the-build-server-your-projects-heart-monitor/ [Dostopano 24. 7. 2016].

[40] Best Practices for Code Review [Online]. Dosegljivo: https://smartbear.com/learn/code-review/best-practices-for-peer-code-review [Dostopano 24. 7. 2016].

[41] J. Merčun (2014) Uvedba agilne metodologije razvoja na sistemu za vodenje podjetja [Online]. Dosegljivo: http://eprints.fri.uni-lj.si/2786/1/63020106-JURE_MER%C4%8CUN-Primer_prehoda_na_agilno_metodologijo_razvoja_programske_opreme.pdf [Dostopano 24. 7. 2016].

[42] C. Smith (2015) The 5 Big Benefits of Automated Deployment [Online]. Dosegljivo:

http://www.red-gate.com/blog/database-lifecycle-management/5-big-benefits-automated-deployment [Dostopano 24. 7. 2016].

[43] M. Fowler (2006) Continuous Integration [Online]. Dosegljivo: http://martinfowler.com/articles/continuousIntegration.html [Dostopano 24. 7. 2016].

[44] C. Caum (2013) Continuous Delivery Vs. Continuous Deployment: What's the Diff? [Online]. Dosegljivo: https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff [Dostopano 24. 7. 2016].

[45] M. Fowler (2013) ContinuousDelivery [Online]. Dosegljivo: http://martinfowler.com/bliki/ContinuousDelivery.html [Dostopano 24. 7. 2016].

[46] M. Kampe What is a post-mortem? [Online]. Dosegljivo: http://www.cs.pomona.edu/classes/cs181f/supp/postmortem.html [Dostopano 24. 7. 2016].

[47] Java Software [Online]. Dosegljivo: https://www.oracle.com/java/index.html [Dostopano 25. 7. 2016].

[48] Java Platform, Enterprise Edition (Java EE) [Online]. Dosegljivo: http://www.oracle.com/technetwork/java/javaee/overview/index.html [Dostopano 25. 7. 2016].

[49] Atlassian JIRA [Online]. https://www.atlassian.com/software/jira [Dostopano 25. 7. 2016].

[50] Red Hat WildFly [Online]. http://wildfly.org [Dostopano 25. 7. 2016].

[51] N. Salnikov-Tarnovski (2016) Most popular Java application servers: 2016 edition [Online]. Dosegljivo: https://plumbr.eu/uncategorized/most-popular-java-ee-servers-2016-edition [Dostopano 25. 7. 2016].

[52] SonarSource SonarQube [Online]. Dosegljivo: http://www.sonarqube.org [Dostopano 25. 7. 2016].

[53] JetBrains Upsource [Online]. Dosegljivo: https://www.jetbrains.com/upsource [Dostopano 25. 7. 2016].

[54] Jenkins [Online]. Dosegljivo: https://jenkins.io [Dostopano 25. 7. 2016].

[55] Sonatype Nexus [Online]. Dosegljivo: http://www.sonatype.org/nexus [Dostopano 25. 7. 2016].

[56] Ansible [Online]. Dosegljivo: https://www.ansible.com [Dostopano 25. 7. 2016].

[57] Semaphore [Online]. Dosegljivo: https://github.com/ansible-semaphore/semaphore [Dostopano 25. 7. 2016].

[58] Slika povzeta po Web Gain Soft Corporation [Online]. Dosegljivo: http://www.wg-soft.net/outsourcing.html [Dostopano 31. 7. 2016].

[59] Slika povzeta po Profit Labs - Development Process [Online]. Dosegljivo: http://www.profit-labs.com/development-process [Dostopano 28. 7. 2016].