Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Valneja Stojčič

Vpeljava agilnega in vitkega razvoja programske opreme v majhnem podjetju

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Viljan Mahnič

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo: Vpeljava agilnega in vitkega razvoja programske opreme v majhnem podjetju

Tematika naloge:

Proučite značilnosti agilnega in vitkega razvoja programske opreme s poudarkom na metodah Scrum in Kanban, ki se najpogosteje uporabljata. Na podlagi tega predlagajte najprimernejši način za vpeljavo omenjenih metod v razvojni proces manjšega podjetja za razvoj programske opreme. Pri tem upoštevajte značilnosti projekta, v okviru katerega bo potekalo uvajanje. V dogovoru s sodelavci vpeljite predlagani način dela in ovrednotite dosežene rezultate, še zlasti potrebni čas (angl. *lead time*).

Zahvaljujem se mentorju prof. dr. Viljanu Mahniču za izjemno odzivnost, pomoč in vodenje pri izdelavi diplomske naloge. Zahvala gre tudi sodelavcema Aleksandru in Vanji. Na koncu se zahvaljujem svoji družini, Robertu in njegovi družini za potrpežljivost in spodbudo med izdelovanjem diplomske naloge ter skozi celoten študij.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
WIP	work in progress	delo v teku
BI	business intelligence	poslovna inteligenca

Povzetek

Naslov: Vpeljava agilnega in vitkega razvoja programske opreme v majhnem podjetju

Avtor: Valneja Stojčič

Povzetek: V diplomskem delu je predstavljena vpeljava agilnih in vitkih metod v proces podjetja, ki je nekatere prakse uporabljalo že prej, vendar nedosledno. Za uspešno vpeljavo je bilo najprej potrebno preučiti značilnosti agilnega in vitkega pristopa s poudarkom na Scrumu in Kanbanu. Upoštevajoč značilnosti projekta se je izkazalo, da bi bila najprimernejša kombinacija obeh - Scrumban. Na podlagi tega je bil izdelan načrt vpeljave, s točno opredelitvijo uporabljenih metod, izbranih pravil in predvidenega poteka dela. Po vpeljavi je bila opravljena analiza uspešnosti, kjer je uspešnost ocenjena s pomočjo grafov in spremljanja doslednosti uporabe izbranih praks. Rezultati meritev so pokazali, da je bila vpeljava uspešna in posledično dostava projektnega izdelka hitrejša.

Ključne besede: Kanban, Scrum, Scrumban, agilne metode, vitki razvoj, razvojni proces.

Abstract

Title: Implementing agile and lean software development in a small company

Author: Valneja Stojčič

Abstract: This Bachelor's thesis introduces the implementation of agile and lean methods into the company process, which used some practices previously, but inconsistently. For successful implementation, the characteristics of agile and lean approach, with an emphasis on Scrum and Kanban, had to be considered first. Considering the characteristics of the project, it became apparent that the combination of both would be best – Scrumban. Based on this, an implementation plan with an exact characterization of methods used, selected rules and estimated work course was formed. After the implementation, a performance analysis was carried out, which assessed the success rate with the help of graphs and monitoring the consistency of using specific practices. The results proved that the implementation was successful and, consequently, the project product delivery was faster.

Keywords: Kanban, Scrum, Scrumban, agile development, lean development, working process.

Poglavje 1

Uvod

V veliko podjetjih, ki se ukvarjajo z razvojem programske opreme, že uporabljajo agilne metodologije. Sprva so v podjetjih uporabljali slapovni model, ki je temeljil na zaporednem izvajanju faz (analiza, načrtovanje, izvedba, testiranje, uvedba) [14]. Takšen razvoj je bil počasen, neprilagojen na spremembe, izdelek pa ni bil takšen, kot je bil naročen. Kljub temu, da je bila vnaprej narejena vsa dokumentacija in so bile zahteve zelo podrobno opredeljene, proces ni dal zaželenih rezultatov. Preveč časa se je porabilo za načrtovanje, zato ga je premalo ostalo za dejansko implementacijo. Kasneje so se začele uvajati agilne metodologije, ki so odpravile težave. Proces je bil prilagojen na spremembe, dokumentacija ni bila več tako pomembna, naročnik pa je pogosteje dobival delujoče kose izdelka.

Kanban se je najprej začel uporabljati v proizvodnji, kasneje so ga zaradi široke uporabnosti prenesli na razvoj programske opreme. Pogosteje se je začel uporabljati šele v zadnjih letih in se zaradi široke uporabe Scruma počasi prebija na trg. Prednost Kanbana pred drugimi metodologijami je preprostost in majhno število pravil. Kljub temu da agilne metodologije v splošnem izboljšajo kvaliteto izdelka in pospešijo proces, njihova uporaba pogosto ni pravilna, rezultati pa niso takšni, kot bi si jih želeli. Pravilna uporaba agilnih metod ne pomeni, da procesa ne smemo prilagoditi svojim potrebam.

Podjetje, v katerem sem zaposlena, se ukvarja z načrtovanjem in razvojem kompleksnih informacijskih sistemov, poslovno inteligenco in analitiko. Projekt, ki ga analiziram, se izvaja že več let in je pred prenovo procesa že uporabljal nekatere prakse Kanbana in Scruma. Vendar procesi niso bili dobro načrtovani in prilagojeni izvajanju. Problem je predstavljala tudi pogosta menjava razvijalcev na projektu, ki niso upoštevali pravil, definiranih na začetku izvajanja. Vse to je vplivalo na kvaliteto in potreben čas (angl. *lead time*) za izdelavo posameznih funkcionalnosti.

Namen diplomske naloge je opisati vpeljavo agilnega in vitkega razvoja v obstoječ projekt ter analizirati uspešnost uporabljene metode, ob tem pa iskati odpad, ki se v razvoju programske opreme pojavlja. Predvsem nas je zanimalo, koliko lahko zmanjšamo potreben čas in si olajšamo ter bolje organiziramo delo.

V drugem poglavju so predstavljene agilne metodologije in prakse, ki so bile uporabljene. V tretjem sta predstavljena načrt vpeljave in dogovorjena pravila. V četrtem pa rezultati opažanj skozi proces, podkrepljeni z meritvami, ki so narejene s pomočjo orodja za spremljanje procesov Jira. Opisano je, kako smo se spopadali s spremembami, ki so se med razvojem pojavljale. Dodane so tudi primerjave uspešnosti s prejšnjim načinom dela.

Poglavje 2

Agilne in vitke metodologije

2.1 Agilne metodologije

Agilne metodologije [9] so skupek metod za razvoj programske opreme. Besedna zveza je bila prvič uporabljena leta 2001, čeprav so se nekatere agilne metodologije uprabljale že prej. Leta 2001 so zagovorniki agilnega razvoja objavili Manifest agilnega razvoja programske opreme [6], ki poudarja naslednje vrednote:

- posamezniki in interakcije pred procesi in orodji,
- delujoča programska oprema pred vseobsežno dokumentacijo,
- sodelovanje s stranko pred pogodbenimi pogajanji,
- odziv na spremembe pred togim sledenjem načrtom.

Vsakega uporabnika na koncu najbolj zanima delujoča programska oprema, torej z obsežno dokumentacijo ne bo zadovoljen, če končni izdelek ne bo takšen, kot si ga je želel. Prav tako uporabnika ne zanima prvoten načrt in kako dobro so se ga razvijalci držali, če končni izdelek ni prilagojen spremembam, ki so se pojavile med razvojem. Marsikaterih zahtev na začetku še ne bomo mogli zajeti in naivno bi bilo verjeti, da lahko že takrat sestavimo popoln načrt. Predvsem pa izdelava delujoče programske opreme zahteva

dobro komunikacijo, tako znotraj razvojne skupine, kot z naročnikom. Le tako je dosežena prilagodljivost spremembam, ki so v vsakem koraku razvoja dobrodošle. Več o agilnih metodah v informacijskih sistemih je opisano v članku [7].

Vitki razvoj se je najprej pojavil v proizvodnji, prvič pa sta ga na razvoj programske opreme prenesla Mary in Tom Poppendieck v knjigi Lean Software Development [13]. Cilj vitkega razvoja je prepoznati in odstraniti vse elemente, ki ne ustvarjajo dodane vrednosti. Elementi (programska oprema, aktivnosti itd.) z dodano vrednostjo so le tisti, ki jih je končni kupec pripravljen plačati.

Scrum prav tako deli zapletene naloge na manjše dele oziroma uporabniške zgodbe (angl. *user story*). Skupine se zavežejo k dostavi delujočega kosa končnega produkta na koncu vsakega vnaprej določenega intervala. [3]

Kanban je metodologija, kjer je pomemben nenehen razvoj in dostava funckionalnosti. Skupine uporabljajo kanban tablo za vizualizacijo dela, ki je razdeljeno na manjše kose, prikazane na karticah. Te so pritrjene na tablo in horizontalno potujejo po stolpcih, ki predstavljajo faze izdelovanja funkcionalnosti do dokončanja.

Scrumban je agilna metodologija, ki združuje lastnosti Scruma in Kanbana, zato je za njegovo uporabo potrebno poznati pravila in lastnosti obeh metodologij [11, 12].

2.2 Scrum

2.2.1 Definicija

Metodologija Scrum [15, 16] je na področju vodenja in razvoja programske opreme trenutno najbolj priljubljena in največkrat uporabljena [1]. Temelji na iterativnem in inkrementalnem pristopu razvoja kompleksnih rešitev. Razvoj poteka v iteracijah (angl. *sprint*), delo pa je razdeljeno na več manjših in lažje obvladljivih kosov. Vsak, ki je posredno ali neposredno vključen v razvoj, ima v procesu točno določeno vlogo.

2.2.2 Vloge

V metodologiji Scrum so predpisane tri vloge: skrbnik izdelka (angl. *product owner*), skrbnik metodologije (angl. *scrum master*) in razvojna skupina (angl. *development team*).

Naloga skrbnika izdelka je predstaviti naročnikove želje razvojni skupini. Želene funkcionalnosti so razdeljene na uporabniške zgodbe in opisane na karticah, ki so umeščene v seznam zahtev (angl. *product backlog*). Za opise in umestitev v seznam skrbi skrbnik izdelka. Na začetku projekta mu ni potrebno razdelati in opisati vseh zgodb, ampak le tiste, ki so najbolj prioritetne in se bodo v iteraciji izvedle najprej. Tekom projekta se o končnem produktu razkriva vedno več podrobnosti glede na to, katere informacije razvijalci trenutno potrebujejo za uspešen razvoj. Skrbnik izdelka lahko skozi celoten razvoj dodaja uporabniške zgodbe v seznam zahtev in jim dinamično spreminja prioriteto glede na trenutne potrebe. Poleg tega mora biti razvojni skupini na razpolago za dodatna vprašanja, da razvoj poteka nemoteno. Ko je uporabniška zgodba končana, jo mora skrbnik izdelka tudi potrditi, da gre lahko nova funkcionalnost v produkcijo. Če uporabniško zgodbo zavrne, se ta vrne spet v razvoj.

Skrbnik metodologije skrbi, da je Scrum jasen vsem članom skupine ter da razvoj poteka tekoče. Opaziti mora nepravilno izvajanje procesa in na to tudi opozoriti. Njegova naloga je spodbujati razvojno skupino k čimvečjemu napredku, hkrati pa jo varovati pred preobremenjenostjo, ki bi jo lahko povzročil skrbnik izdelka.

Razvojna skupina običajno obsega od 3 do 9 ljudi in skrbi za realizacijo zahtev. Med seboj si člani razdelijo delo ter se sami odločajo, kako bodo prišli do rešitev zahtevanih funkcionalnosti. Ob nejasnostih je njihova naloga, da pri skrbniku izdelka pridobijo potrebne informacije.

2.2.3 Uporabniške zgodbe

Uporabniška zgodba opisuje funkcionalnost, ki je bila zahtevana s strani naročnika [8]. Sestavljene so iz pisnega opisa funkcionalnosti (za planiranje in kot opomnik), pogovorov o njej (za pojasnjevanje podrobnosti) in sprejemnih testov (za določanje, kdaj je zgodba končana).

Vsaka zgodba mora biti pred implementacijo ocenjena z ustreznim številom točk (angl. *story points*), ki nam povejo, kako zahtevna je za implementacijo v primerjavi z drugimi zgodbami. Ena točka predstavlja približno tri do šest ur dela, idealen delovni dan pa je ocenjen s šestimi urami neprekinjenega dela. Običajno je ocena število Fibonaccijevega zaporedja (1, 2, 3, 5, 8, 13, 20), saj v primerjavi z lažjimi, kjer so razlike po eno točko, težjih zgodb ni mogoče tako točno oceniti. Na primer; obsežno zgodbo je težko oceniti s točno 11 točkami, zato izberemo najbližjo iz zaporedja (torej 13), zelo enostavno zgodbo pa brez problema lahko ocenimo s točno 2 točkama, kar pomeni da bo funkcionalnost končana v enem ali dveh dneh. Ocene potrebujemo v času načrtovanja iteracij, ko določamo, koliko zgodb lahko uspešno realiziramo znotraj določene iteracije.

2.2.4 Iteracije in sestanki

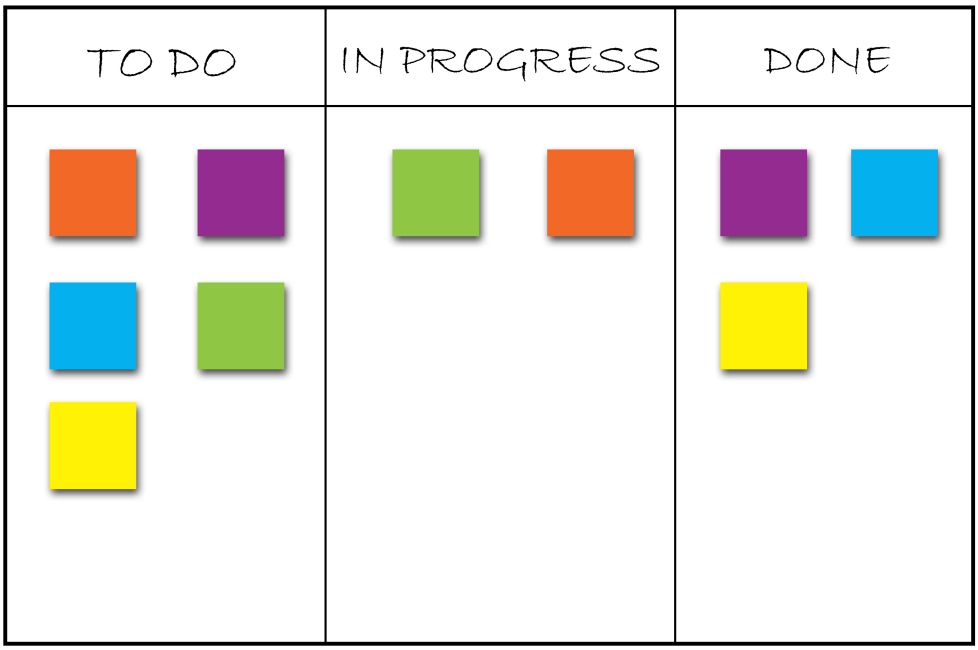
Dolžina iteracij je vnaprej določena in je ponavadi od enega tedna do enega meseca, kjer je dolžina dveh tednov najbolj pogosta. Vsaka iteracija se začne s sestankom načrtovanja iteracije (angl. *sprint planing*), kjer analiziramo hitrost prejšnjih iteracij (angl. *sprint velocity*). Glede na hitrost iteracije določimo, koliko in katere zgodbe (z že določenimi ocenami) so lahko končane v tej iteraciji. Sestanka se udeležijo skrbnik izdelka, skrbnik metodologije in razvojna skupina, lahko pa tudi naročnik in ostali zainteresirani za izdelek.

Razvojna skupina skozi trajanje iteracije razvija funkcionalnosti iz zgodb, ki so v seznamu zahtev. Vsak dan ob isti uri se člani skupine sestanejo in pogovorijo o delu, ki ga trenutno opravljajo. Vsak razvijalec odgovori na tri vprašanja: "Kaj sem počel včeraj? Kaj bom počel danes? Kakšne težave

imam pri izpolnjevanju naloge?" Sestanki običajno trajajo 15 minut.

Na koncu iteracije potekata dva sestanka. Retrospektiva (angl. *sprint retrospective*) je sestanek, kjer naredi razvojna skupina pregled, kaj je bilo v tej iteraciji dobrega in kaj se lahko izboljša. Pregled iteracije (angl. *sprint review meeting*) je sestanek o produktu, izdelanem v tej iteraciji, kjer je lahko produkt tudi demonstriran.

2.3 Kanban



Slika 2.1: Osnovna kanban tabla

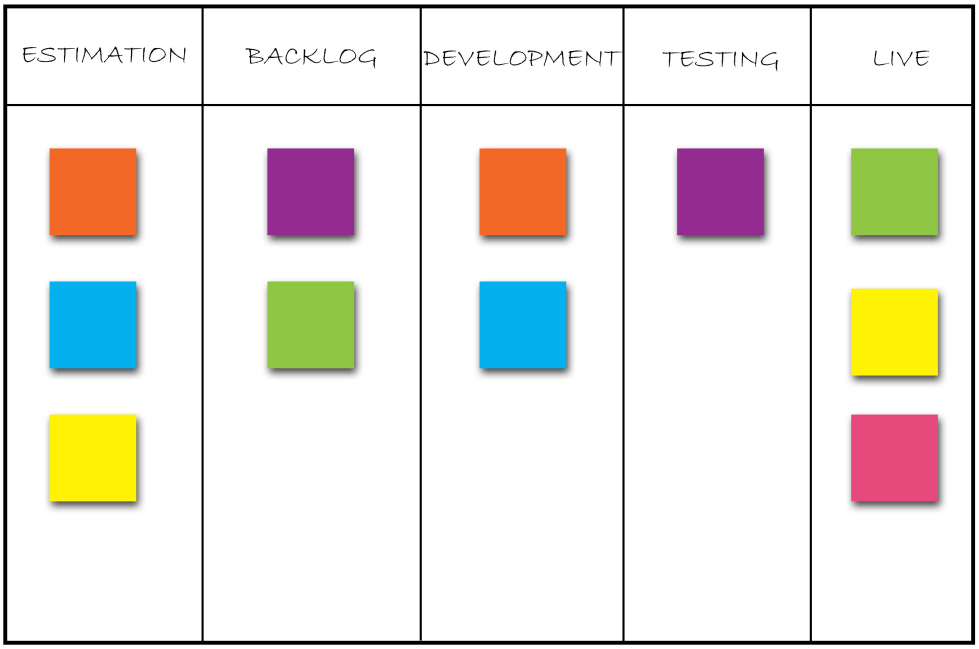
2.3.1 Definicija

Kanban je vitka metodologija za upravljanje in izboljševanje dela. Metodologija ni tako stroga kot Scrum, saj na primer iteracije niso časovno določene in v skupini ni nujno točno določenih vlog. Kanban je zelo prilagodljiv in temelji na kakovosti ter čimprejšnji dostavi programske opreme.

2.3.2 Kanban tabla

Kanban tabla je glavno orodje za vizualizacijo in spremljanje dela v teku. Razdeljena je na poljubno število stolpcev, ki prikazujejo posamezna stanja

v razvojnem procesu. Z njeno pomočjo spremljamo delovni tok (angl. *workflow*) od leve proti desni. Najbolj preprosta kanban tabla na sliki 2.1 prikazuje tri glavne stolpce: "to-do", "in progres", "done". Seveda pa je možno in zaželjeno, da vsaka skupina prilagodi stolpce svojim potrebam (primer na sliki 2.2).



Slika 2.2: Primer personalizirane, projektu prilagojene kanban table

Glavna naloga table je, da v vsakem trenutku prikazuje trenutno stanje dela in je ves čas posodobljena. Delo je razdeljeno na več nalog, ki so opisane na karticah. Razvijalec, ki konča nalogo, si iz stolpca "to-do" izbere naslednjo kartico in jo premakne v "in progress". Želja je, da kartica kar se da hitro pride iz skrajno levega dela table do skrajno desnega dela, kjer je delo končano. Čas, ki ga kartica porabi na tej poti, je potrebni čas in je glavno merilo uspešnosti. Nekateri stolpci na tabli imajo poleg imena določeno tudi maksimalno število kartic, ki jih lahko vsebujejo (omejitev WIP).

Najprej se je tabla uporabljala v fizični obliki na beli tabli ali pa listu papirja. S pisalom so bili oblikovani stolpci, zahteve pa so bile opisane na listkih, ki so bili pripeti v stolpce. Danes najdemo veliko spletnih orodij, ki nam nudijo takšne table v elektronski obliki, zahteve pa so lahko veliko bolj natančno opisane. Zelo uporabna so tudi obvestila o spremembah na tabli, da smo lahko ves čas na tekočem z delom.

2.3.3 Uporaba Kanbana

Bistveni elementi Kanbana so:

- **vizualizacija delovnega toka** (angl. *visualize the workflow*),

  Delo spremljamo s pomočjo kanban table, kjer je razdeljeno na manjše obvladljive kose, opisane na karticah, ki se pomikajo po tabli s poimenovanimi stolpci. Nove kartice se lahko kadarkoli v procesu dodajajo v skrajno levi stolpec na tabli. V vsakem trenutku mora biti vsakemu, ki pogleda na tablo jasno, kaj dela vsak član razvijalske skupine, kar pomeni, da je naloga razvijalcev, da skrbijo za stalno posodobljenost table.

- **omejitev dela v teku** (angl. *limit Work In Progress - WIP*),

  Omejitev dela v teku pomeni določitev maksimalnega število nalog (kartic), ki se lahko v nekem trenutku nahajajo na določeni stopnji razvoja oziroma v enem stolpcu kanban table. Prav tako je omejitev dela v teku prisotna tudi pri Scrumu, le da je ta glede na število točk vseh zgodb v eni iteraciji. Ker pri Kanbanu nimamo iteracij, lahko omejimo le število nalog, ki so lahko v nekem trenutku v izvajanju. Kanban ne predpisuje kakšna, mora biti ta omejitev in jo mora skupina izbrati sama.

  Če bi bila omejitev preveč ohlapna, bi se lahko zgodilo, da bi v procesu nastal zastoj, ki ga dolgo ne bi opazili. Za primer si predstavljajmo dva

razvijalca, ki uporabljata tablo s štirimi stolpci: "to-do", "in development", "testing", "done", kjer imata "in development" ter "testing" omejitev največ 4 kartice. Razvijalca najprej vzameta vsak svojo kartico in jo končano premakneta v "testing", kjer sta sedaj dve kartici. To ponovita še enkrat in stolpec "testing" doseže omejitev. Ko ponovno vzameta vsak svojo kartico, ju ob končanju ne moreta več premakniti naprej, vendar ima stolpec "in development" prostora za štiri kartice, zato si vzameta še dve. Tako je v stolpcu "testing" nastalo ozko grlo in povzročilo zastoj tudi v stolpcu "in development", kar se je opazilo relativno pozno. Sedaj se morata oba razvijalca posvetiti testiranju in izprazniti ta stolpec, nato pa testirati še vse kar je ostalo v "in development". Če bi že na začetku definirali manjšo omejitev, se v stolpcih zgodbe ne bi nabirale in bi končane funkcionalnosti prihajale v stolpec "done" prej, naročnik pa bi bolj redno dobival delujoče kose programske opreme.

Obraten primer bi bila prestroga omejitev, kjer razvijalci ostajajo brez dela. V tem primeru trije razvijalci uporabljajo tablo z enakimi stolpci kot prej, vendar je sedaj omejitev na obeh največ 1 kartica. Prvi razvijalec vzame eno kartico, kar pomeni da je stolpec "in development" zapolnjen in ostala dva razvijalca ostaneta brez dela. Šele ko prvi razvijalec konča z delom na kartici, jo lahko eden od treh testira in drugi začne delo na novi, kjer eden še vedno ostaja brez dela.

Na začetku je zelo težko izbrati primerno omejitev in se jo lahko tekom procesa prilagaja tako, da je proces čimbolj optimiziran. Razlogi za potrebo po spremembi so:

- sprememba števila razvijalcev v skupini,
- prvotna omejitev je prenizka in razvijalci ostajajo brez dela,
- prvotna omejitev je preveč ohlapna in razvijalci izvajajo preveč nalog na enkrat.

- **merjenje potrebnega časa** (angl. *measure the lead time*).

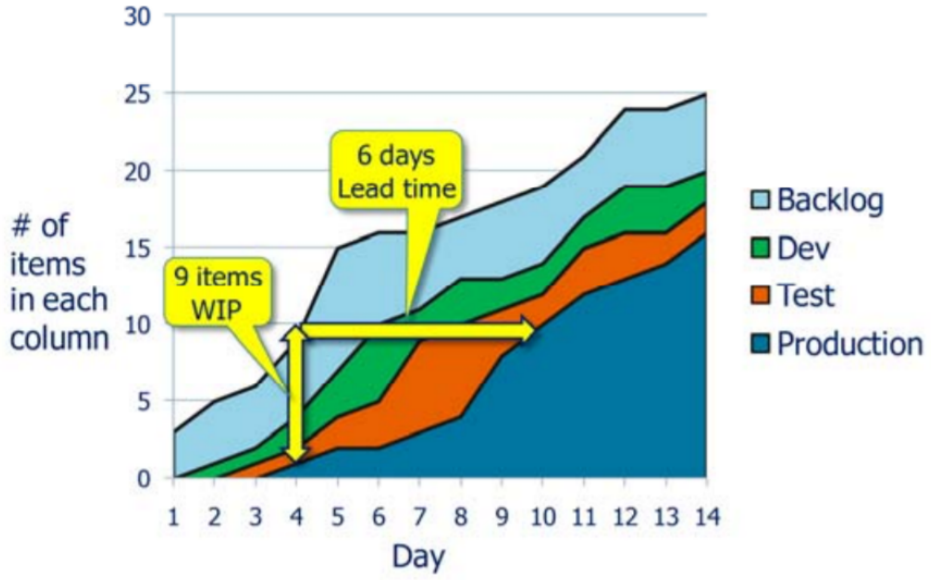
Potrebni čas je čas, ki ga kartica porabi za potovanje od vstopa v skrajno levi stolpec do skrajno desnega. Z merjenjem potrebnega časa spremljamo napredek ter iščemo nepravilnosti, ki se tekom procesa dogajajo. Naloga razvijalcev je da, proces optimizirajo tako, da je potrebni čas čim manjši in čim bolj enakomeren. Kako s pomočjo kumulativnega diagrama delovnega toka merimo potrebni čas, je prikazano na sliki 2.3.

Predstavljenih je nekaj prednosti Kanbana pred drugimi metodologijami:

- Ozka grla so hitro vidna, kar pomeni, da je vidno, kje v procesu je nastal zastoj in je delo ustavljeno. S tem, ko se takšne nepravilnosti pojavijo in vplivajo na vse vpletene, ljudje bolj pogosto sodelujejo in skupaj poskušajo optimizirati celoten proces, in ne samo svojega dela.
- Nudi postopnen razvoj delovnega procesa od slapovnega (waterfall) do agilnega, kar je pomembno predvsem za podjetja, ki pred tem niso nikoli uporabljala agilnih pristopov.
- Nudi možnost agilnega razvoja, brez potrebe po časovno določenih iteracijah, sestankih, vlogah in ostalih pravilih kot pri Scrumu in je zaradi tega predvsem lažje razumljiv.

2.4 Eliminacija odpada

V vitkem razvoju [10] je odpad nekaj, kar kupcem ne predstavlja vrednosti. Izraz se je prvič pojavil v poznih štiridesetih letih 20. stoletja, ko je majhno podjetje Toyota začelo s proizvodnjo vozil. Ker ljudje niso imeli veliko denarja, so avtomobili morali biti poceni. Nižjo ceno bi lahko dosegli z množično izdelavo, kar je pomenilo izdelavo več tisoč enakih avtomobilov. Vendar pa je bil japonski trg takrat premajhen, da bi potreboval toliko vozil. Tukaj se je pojavilo vprašanje, kako bi lahko Toyota naredila manj vozil z



Slika 2.3: Kumulativni diagram delovnega toka [11]

enako ceno, kot pri množični proizvodnji. Odgovor na to vprašanje je bil čisto enostaven: eliminacija odpada. Karkoli kar v prozvodnem procesu ne predstavlja vrednosti za kupca, je odpad. Enako velja za razvoj programske opreme.

Prakse agilnih pristopov stremijo k zmanjševanju odpada. Vendar pa je za izvedbo eleminacije potrebno naprej odpad poiskati med vsemi procesi, ki so za razvoj produkta uporabljeni. Pri razvoju programske opreme se lahko za vsak proces razen analize zahtev in pisanja kode vprašamo, če ga zares potrebujemo.

Sedem vrst odpada pri razvoju programske opreme je [13]:

Delno dokončano delo pogosto postane zastarelo, preden je dokončano. Poleg tega je v napoto drugemu delu, ki ga je potrebno dokončati. Delno dokončano delo porablja sredstva in dokler ni v produkciji, niti ne vemo, ali reši problem, ki je bil predstavljen v zahtevah.

Dodatni procesi porabljajo dragocen čas, ampak ne prinašajo nobene vrednosti. Dober primer je papirologija, ki je nihče ne bere, kljub temu pa porabi veliko virov. Kadar je ta res potrebna, naj bo kratka in jedrnata. Najbolj se vidi, ali je papirologija potrebna, če vidimo, da nekdo čaka na njo. Tudi v takšnem primeru naj bo napisana kar se da učinkovito.

Dodatne funkcionalnosti, ki so dodane za vsak slučaj, če jih bo kdo potreboval, se sprva zdijo neškodljive in dobra ideja, ampak so v resnici velik odpad. Vsak del kode, ki je napisan, mora biti integriran, preveden in testiran ter kasneje vzdrževan cel življenski cikel. Zato se je potrebno držati pravila, da kode ne dodamo, če je trenutno ne potrebujemo, saj predstavlja odpad.

Preklapljanje med nalogami vzame veliko časa, saj potrebuje razvijalec vsakič nekaj časa, da preklopi misli in se prilagodi delu na drugi nalogi. Najhitrejši način, da končamo dva projekta je, da delamo enega na enkrat. Če traja en projekt dva tedna, to pomeni, da imamo recimo en projekt končan v dveh tednih in dva projekta v štirih. Kadar izdelujemo oba istočasno, ne bomo imeli po dveh tednih končanega nobenega in zelo težko dva končana projekta v štirih tednih. Najverjetneje bomo s preklapljanjem med nalogami potrebovali za oba projekta skupaj pet tednov, kar je en teden več, kot če delamo vsakega neodvisno od drugega.

Čakanje je eden izmed navečjih odpadov v razvoju programske opreme, saj je prisotno pri skoraj vsakem projektu. Pogosto se pri razvoju zdi, da je čakanje naravno. A moramo se zavedati, da so zakasnitve začetka projekta, preobsežne dokumentacije zahtev, potrditev in testiranja odpad ter o tem razmišljati že pred začetkom projekta. Točno moramo vedeti, kdo kaj dela in kako bomo med izvedbo komunicirali, če pride do nejasnosti, da nam ne bo potrebno čakati nekaj dni na osebo, ki ima odgovore na določena vprašanja.

Premikanje v razvoju je širok pojem. Lahko pomeni premikanje informacij, zahtev, kode, razvijalcev. Kako dolgo potujejo podatki, da razvijalec dobi vse potrebne informacije? Ali so ljudje na dosegu roke za tehnično pomoč? Ali je potrebno po odgovor na drug konec hodnika? Kakšno pot potuje koda ob predaji projetka drugemu razvijalcu? Premikanje vseh teh stvari porabi veliko časa.

Napake so lahko različno velik odpad, odvisno od tega, koliko časa potrebujemo, da jih odkrijemo. Kritična napaka, ki je odkrita v treh minutah, ni tako velik odpad, kot majhna napaka, ki ostane neodkrita več tednov. Način za zmanjševanje napak je sprotno odkrivanje s testiranjem in integracijo, takoj ko je to mogoče.

Poglavje 3

Predstavitev podjetja in projekta

3.1 Predstavitev podjetja

Podjetje, v katerem sem zaposlena, je bilo ustanovljeno leta 1989 in danes šteje preko 50 zaposlenih. Ukvarja se z načrtovanjem in razvojem kompleksnih informacijskih sistemov ter poslovno inteligenco in analitiko.

3.2 Predstavitev projekta

Projekt je naročilo podjetje iz Nemčije in se ukvarja z izposojo vozil (rent-a-car). Ker so locirani v tujini, večina naše komunikacije poteka preko spleta, občasno pa sestanki potekajo tudi pri njih v Nemčiji. Načrtovano je bilo, da naj bi se projekt zaključil spomladi 2018, ampak se to ni zgodilo. Zaradi menjave vpletenih ljudi in naročnikovih želja o dodatnih funkcionalnostih se je izvajanje podaljšalo in trenutno rok za dokončanje ni določen.

Projekt se izvaja že nekaj let in je v zadnjih mesecih potreboval prenovo zaradi menjave zaposlenih na projektu ter drugih sprememb znotraj razvoja produktov. Ob začetku dela na diplomski nalogi maja 2018, smo posodobili način dela in metode, ki jih uporabljamo. Pred tem se je uporabljal Scrum,

nato pa se je začel počasi uvajati Kanban. Zaradi pomanjkljivega načrta in nedosledne uporabe pristopov niso bile izkoriščene vse prednosti agilnih metodologij. Skupaj smo se kot nova skupina dogovorili, da bomo uporabljali Kanban ter naš napredek skrbno beležili. Hkrati pa smo ugotovili, da nam vsem ustrezajo nekateri elementi Scruma ter v naš proces dodali še nekaj teh. Vsi smo se tako strinjali, da bomo za našo metodologijo izbrali Scrumban. Za dosledno in pravilno uporabo Scrumbana, morajo imeti vsi člani dovolj znanja o metodologiji in takrat sem se ponudila, da Scrumban raziščem in pomagam pri pravilni izvedbi ter vsak mesec opravim analizo uspešnosti.

Naša skupina je trenutno sestavljena iz treh članov in razvijamo dve aplikaciji, ki sta medsebojno povezani. Naše delo obsega vzdrževanje stare kode ter razvijanje novih funkcionalnosti. Obe aplikaciji sta namenjeni upravljanju podatkov in preračunavanju stroškov na pogodbah za nakup vozil. Prva aplikacija vsebuje poslovno logiko za izračun stroškov, druga pa pogoje nakupa vozil.

Poglavje 4

Načrt vpeljave

Kanban ima zelo malo pravil, pa še ta so zelo ohlapna in jih je potrebno definirati za vsak projekt posebej. Scrum je bolj strikten glede pravil, vendar jih, je prav tako kot pri Kanbanu, potrebno podrobneje definirati. Pred začetkom izvajanja Scrumbana smo določili strukturo kanban table in omejitve WIP na stolpcih, ki sta pravili Kanbana ter uvedli vloge, iteracije, dnevne sestanke, sestanke, povezane z iteracijami, pravila in predviden potek dela, ki so pravila Scruma.

4.1 Določitev vlog

Vloge smo uporabili, ker smo s sistemom vlog že vsi vpleteni seznanjeni. Poleg tega se vsi strinjamo, da takšna delitev vlog pripomore k lažjemu reševanju zahtev in boljši komunikaciji.

Skrbnik izdelka je oseba iz podjetja, ki je naročilo produkt. Ta oseba najbolje pozna zahteve in je na voljo za vprašanja. Ker se ta oseba nahaja v Nemčiji, izvajamo večino komunikacije preko videoklicev ali elektronske pošte.

Skrbnik metodologije je oseba iz naše skupine, ki ima največ izkušenj z vodenjem projektov ter najbolje pozna ta produkt, saj ga razvija že nekaj let. Kot je bilo že omenejo, delamo na tem projektu trije člani, zato je skrbnik

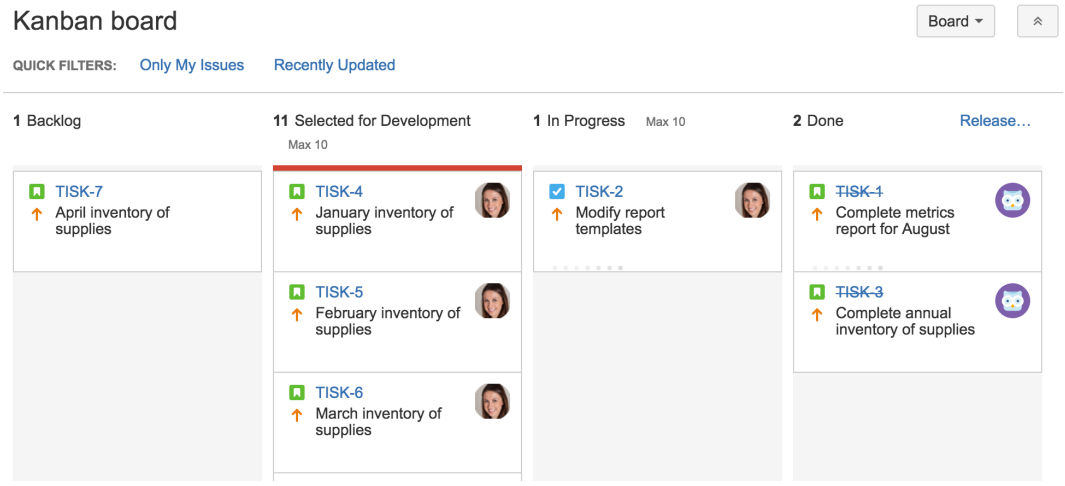
metodologije hkrati tudi član razvojne skupine.

Razvojna skupina smo trije programerji, ki razvijamo tako čelni del (angl. *frontend*) kot zaledni del (angl. *backend*) in delamo na obeh aplikacijah.

Naloga vseh vpletenih je, da se vsak zaveda svoje vloge in je seznanjen z dolžnostmi, ki jih ta prinaša. Vsakršne nejasnosti vedno najprej rešujemo znotraj skupine (razvojna skupina in skrbnik metodologije) in nato šele kontaktiramo skrbnika izdelka.

4.2 Uporaba orodja za vodenje projekta

Potrebovali smo orodje, ki nudi grafični prikaz za spremljanje napredka, pregled trenutnega stanja, upravljanje zahtev ter nudi možnost osnovne komunikacije pri vsaki zgodbi (možnost komentiranja ter podajanja dodatnih informacij). Poleg tega pa je najpomembneje, da podpira delo z obema metodologijama in vsebuje kanban tablo za spremljanje dela.



Slika 4.1: Primer kanban table v Jiri [4]

Odločili smo se za uporabo Jire [2], ker nudi ravno te funkcionalnosti, ki jih potrebujemo.

Kanban tabla v Jiri na sliki 4.1 je razdeljena na stolpce, ki jih določi skupina, poleg imena pa prikazuje število kartic, ki so trenutno v vsakem stolpcu in omejitev WIP, ki je na stolpcu morebiti določena. Uporabnik lahko premika kartice po stolpcih glede na delo, ki ga trenutno izvaja. Vsaka kartica je na tabli na kratko opisana, prikazuje pa tudi nekaj ostalih ključnih informacij. Prikazuje, kdo se z delom na njej ukvarja, kakšnega tipa je (zgodba, hrošč), njena prioriteta (nizka, srednja, visoka) ter planirana izdaja. Vsako kartico se lahko podrobneje pogleda s klikom na njo. Poleg informacij, ki so vidne že iz table, so tam še komentarji, posnetki zaslonov, načrti in podrobnejši opis naloge z vsemi informacijami, ki so za razvoj potrebne. Prav tako se pod vsako nalogo beleži zgodovina vseh aktivnosti na njej, na primer vsi premiki med stolpci, spreminjanje informacij na kartici, komentarji itd.

4.3 Predstavitev zahtev

Skrbnik izdelka je zadolžen za opis zahtev na uporabniških zgodbah in njihovo umestitev v seznam zahtev v Jiri v obliki kartice. V okviru ene zgodbe je lahko več zahtev. Primer je zgodba: "Uporabnik lahko pogleda podrobnosti o izbrani pogodbi.", ki ima zahteve: "Prikazani morajo biti osnovni podatki o pogodbi.", "Podatke se lahko spreminja s klikom na polje.", "Uporabnik lahko zbriše pogodbo.". Na vsaki kartici je poleg opisa problema določen tip kartice (zgodba, hrošč), njena prioriteta in oseba, ki se takrat z njo ukvarja. Prav tako so na kartici označene tudi ostale osebe, ki se jih zgodba tiče, v našem primeru vsi člani razvijalske skupine. To pomeni, da te osebe dobivajo obvestila o spremembah na kartici. Ko se oseba odloči za delo na kartici, jo dodeli sebi in premakne v ustrezen stolpec. Ob ugotovitvi, da je neka zgodba preobsežna, je razvijalcu skupaj s skrbnikom izdelka dovoljena delitev na več manjših uporabniških zgodb.

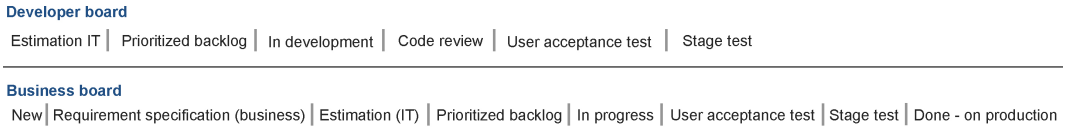
Uporabniške zgodbe morajo vsebovati opise zahtev, ki so jasni tako naročniku ali uporabniku kot razvijalcem. Razvijalec lahko dopolni opise s tehničnimi podrobnostmi, ki bi bile v pomoč pri razumevanju naloge drugim razvi-

jalcem ali pa njemu kasneje. Prav tako lahko pod vsako zgodbo v komentarjih poteka komunikacija o morebitnih nejasnostih in dodatnih informacijah. S tem lahko o dogajanju informiramo ostale člane skupine ali pa shranimo informacije za kasneje.

Kadar je tip kartice hrošč, je na kartico dobro pripeti posnetke zaslona, iz katerih je razvidno, kje v aplikaciji se je pojavil problem in kaj je pričakovan rezultat. S tem preprečimo predolge opise zgodb in bolj jasno prikažemo problem.

Vsaka kartica vsebuje tudi sprejemne teste, ki jih določi skrbnik izdelka. To pomeni, da mora implementacija zadoščati pogojem v sprejemnih testih, če jo želimo objaviti v produkcijo.

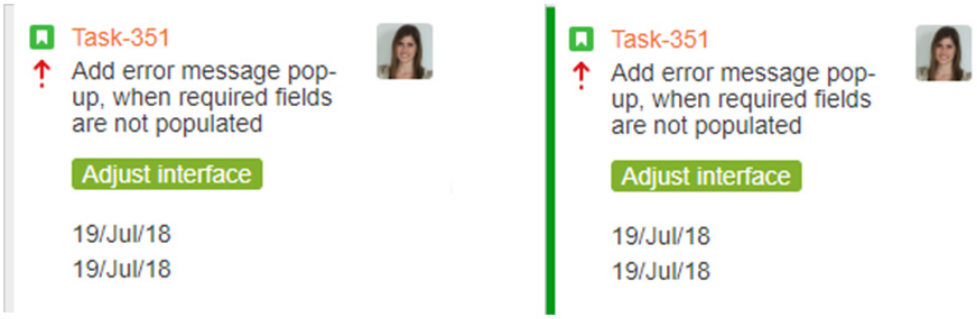
4.4 Struktura kanban table



Slika 4.2: Stolpci obeh kanban tabel na našem projektu

V projektu imamo dve kanban tabli: razvijalsko in poslovno. Prva je namenjena razvijalcem in jim je prilagojena. Druga pa je namenjena skrbniku izdelka, naročniku in ostalim, ki se za produkt zanimajo. Obe tabli sta skladni in vsebujeta iste kartice. Na sliki 4.2 je primerjava stolpcev v obeh. Tabli sta med seboj posodobljeni in se premik kartice na eni vidi tudi na drugi v ujemajočih stolpcih. Na razvijalski tabli manjkajo stolpci "new", "requirement specification (business)", "done - on production" v primerjavi s poslovno tablo. Pri tem je za razliko od poslovne, na razvijalski tabli dodan stolpec "code review". Vse kartice iz razvijalskih stolpcev "in development" in "code review" so na poslovni tabli prikazane v stolpcu "in progress", ostali stolpci z ujemajočimi imeni pa vsebujejo iste kartice.

Nekateri stolpci so navidezno razdeljeni še na dve stanji, kar ni razvidno iz imena stoplca. Stanji prikazujeta, ali je kartica še v izvajanju ali pa je že končana in čaka na prehod v naslednji stolpec. V katerem od teh dveh stanj v stolpcu je trenutno kartica, je vidno iz obarvanosti levega roba kartice, kot je prikazano na sliki 4.3.



Slika 4.3: Primer kartice, ki je še v delu (siva) in ko je že končana (zelena)

Podrobneje je opisana razvijalska tabla, saj je za nas pomembnejša in se direktno nanaša na naše delo.

Estimation IT vsebuje zahteve, ki so potencialni kandidati za naslednjo iteracijo. V času planranja iteracije se zgodbam v tem stoplcu določajo ocene. Planiranje poteka skupaj s skrbnikom izdelka, saj s karticami v tem stolpcu upravlja on. Kartice imajo dve možni stanji, sivo "estimation" in zeleno "estimation done". Ko jih je dovolj ocenjenih in obarvanih zeleno, jih nekaj, glede na hitrost iteracije, skrbnik izdelka premakne v naslednji stolpec.

Prioritized backlog vsebuje zahteve, ki so planirane v tej iteraciji in jih je sem umestil skrbnik izdelka v času planiranja. Lahko pa se zgodi, da v ta stolpec umestimo kartico tudi med iteracijo, če je to potrebno. Pogosto se to zgodi s hrošči, ki jih je nujno čimprej odpraviti.

In development drži kartice, ki so trenutno v delu. Vsa implementacija se dogaja v tem stolpcu. Tudi tukaj je delo razdeljeno na dva dela, "development" in "development done".

Code review je stolpec, ki je bil dodan posebej za razvijalce in ga poslovna stran ne vidi ločeno, ampak združenega s prejšnjim. Razvijalci v tem stolpcu pregledujejo kodo drugih razvijalcev, komentirajo napake in možne izboljšave ter jo na koncu sprejmejo ali zavrnejo. Od tukaj gre lahko kartica v naslednji stolpec ali pa nazaj v razvoj.

User acceptance test vsebuje kartice, ki so končane in so pripravljene za preverjanje. Ko je kartica tukaj, to v praksi pomeni, da se bodo najprej izvedli vsi avtomatski testi s pomočjo orodja Jenkins, nato pa bo skrbnik izdelka preveril, ali implementacija zadošča sprejemnim testom. Če je testiranje uspešno, kartico spremenimo iz sive v zeleno, drugače pa se vrne nazaj v razvoj.

Stage test za nas ni več pomemben. Ko kartica konča fazo testiranja v "User acceptance test" in je bila uspešno sprejeta, je za nas delo na njej končano. Od tukaj naprej so zaposleni drugi ljudje, ki končane funkcionalnosti objavijo v produkcijo.

Na naši tabli imamo tudi pasove, ki poleg vertikalne delitve stolpcev, horizontalno delijo delo glede na to, v kateri sklop spada. Imamo štiri pasove: refaktoriranje obeh aplikacij, priprava podatkov naših aplikacij za oddelek BI (angl. Business intelligence), prva aplikacija, druga aplikacija. Kartice lahko prehajajo med stolpci, vendar je prepovedano prehajanje med pasovi, zato ostanejo v istem pasu od začetka razvoja do zaključka. Pasove smo določili le zaradi preglednosti in nimajo nobene posebne funkcije.

4.5 Omejitve WIP

Po posvetu smo v naši skupini izbrali skupno omejitev 5 za stolpca "in progress" ter "code review", saj smo razvijalci trije in bi bila lahko nižja omejitev

3 prenizka. Ko bi bila naloga v "code review" pri drugih razvijalcih, ta razvijalec ne bi mogel pričeti dela na novi. Ob višji omejitvi, recimo 7 nalog, bi se lahko prepogosto dogajalo, da bi razvijalec začel na eni nalogi in zaradi ovire pri razvijanju v razvoj vzel še dve drugi. Če pa to stori tudi drug razvijalec se potrebni čas ene zgodbe bistveno poveča.

V skupini se držimo ohlapnega pravila omejitev WIP = (število razvijalcev + 2), ponavadi pa se število razvijalcev na takšnih projektih giblje med tri in štiri. Torej bi, v primeru pridobitve novega sodelavca, omejitev na stolpcih "in progress" ter "code review" povečali na šest.

4.6 Osnovna ravila

Pred začetkom dela na projektu je potrebno določiti osnovna pravila (angl. *ground rules*), ki lahko veljajo za vse vpletene ali so vezana na vloge v procesu. Pravila smo določili skupaj s skrbnikom izdelka.

4.6.1 Obveznosti posamezne vloge

- Skrbnik izdelka skrbi za načrtovanje funkcionalnosti ter testiranje in potrjevanje končanih nalog. Prav tako mora biti na voljo razvijalcem za podrobnejšo razlago zahtev, če je to potrebno. Zahtevam določi prioriteto, poskrbi za primeren opis in jih nato umesti na kanban tablo. Na kanban tabli lahko ureja kartice v stolpcih "Estimation IT" in "Prioritized backlog". Prav tako mora, ko je zgodba v stolpcu "User acceptance test", testirati pravilno delovanje aplikacije in jih glede na to sprejeti ali zavrniti.

- Skrbnik metodologije nadzoruje uporabo metodologije na projektu. Seznanjen mora biti z elementi Scruma, ki jih uporabljamo in poznati pravila Kanbana. Skrbeti mora za pravilno izvajanje in dokončanje projekta. V našem primeru je skrbnik metodologije tudi razvijalec, kar pomeni, da ima glede table enake pravice kot razvojna skupina.

- Razvojna skupina skrbi za izpolnjevanje zahtev. Na kanban tabli iz stolpca "Prioritized Backlog" vzame kartico po želji oziroma tisto z najvišjo prioriteto in jo prestavi v "In Development". Ko konča z delom na njej, jo označi kot končano "Development Done" in sporoči ostalima dvema razvijalcema, da je pripravljena za premik v stolpec "Code Review" ter s pomočjo orodja Gitlab odda prošnjo za združitev (angl. *merge request*). Vsak razvijalec mora spremljati obvestila o prošnjah za združitev drugih razvijalcev in jim kodo pregledati. Ko je koda pregledana in sprejeta s strani vseh razvijalcev, mora tisti, ki je kodo napisal, naredit združitev (angl. *merge*) na vejo "Acceptance" in kartico premakniti v stolpec "User acceptance test". Ko ima kartica tudi v stolpcu "Acceptance" status, da je končana, isto naredi še z vejo "Stage". Tukaj je njegovo delo na kartici končano.

4.6.2 Obravnavanje izrednih nujnih zahtev

Kartice iz stolpca "Estimation IT" v stolpec "Prioritized Backlog" premešča skrbnik izdelka po sestanku planiranja, pred začetkom izvajanja iteracije. Torej je skupina dogovorjena, da je v naslednjih dveh tednih (kot je dolga iteracija), potrebno dokončati zgodbe iz "Prioritized Backlog", če je to mogoče. V stolpec "Prioritized Backlog" lahko pride izredna nujna zahteva tudi kadarkoli med iteracijo, zato mora biti skupina že v fazi načrtovanja pozorna na možnost takšnih dogodkov in v tem stolpcu pustiti nekaj prostora za nujne zahteve.

4.6.3 Kako postopati v primeru prekoračitve omejitve WIP

Kljub temu, da smo omejitev znotraj skupine že določili, jo lahko tekom projekta spreminjamo, ampak le če je to res potrebno. Na začetku projekta je eksperimentiranje z omejitvijo dobrodošlo, saj moramo poiskati optimalno mejo, da delo nemoteno teče. Dlje kot delamo s projektom, manjša je potreba

po spreminjanju. V primeru sprememb, ki bi pomenile potrebo po dvigu ali spustu omejitve, se moramo posvetovati s celotno skupino. Skrbnik metodologije oceni, ali je sprememba omejitve potrebna, potem pa mora skupina argumentirano določiti novo.

V primeru, ko je v razvoju že 5 zgodb, lahko razvijalec v razvoj vseeno potegne novo kartico. Ko je omejitev prekoračena, se stolpec pobarva rdeče. Orodje samo torej dopušča prekoračitve, ampak jih opazi. Dogovorjeni smo, da v takšnih primerih razvijalec argumentira svojo odločitev drugim članom skupine in to zapiše v komentar pod zgodbo.

4.7 Potek dela

Naše delo poteka v dvotedenskih iteracijah. Na začetku vsake iteracije (sprinta), skupaj s skrbnikom izdelka, načrtujemo vsebino in količino dela, ki ga lahko opravimo v naslednjih dveh tednih (angl. *sprint planning*), glede na količino dela v preteklih iteracijah.

Zgodbe ocenjujemo s točkami iz Fibonaccijevega zaporedja (1, 2, 3, 5, 8, 13, ...), pri čemer je ena točka ocenjena s tremi urami dela. Iz seznama zgodb po vrsti najprej preberemo opis in zahteve iz vsake zgodbe ter skupaj poskušamo razumeti, kaj je potrebno storiti, ali imamo dovolj informacij, na koga oziroma kaj bo naša rešitev vplivala in kako se zahteve iz tehničnega vidika lotiti. V tej fazi okvirno določimo, kateri od razvijalcev bi lahko nalogo prevzel, glede na znanje in izkušnje programerja, kar pa ne pomeni, da bo to zgodbo nujno prevzel točno določen progamer.

En razvijalec, ponavadi tisti, ki bo okvirno prevzel zgodbo, poda zgodbi točkovno oceno in jo utemelji. Ostali razvijalci se z oceno strinjamo ali pa nestrinjanje argumentiramo. Ocenjevanje zgodbe se konča, ko se vsi razvijalci strinjamo z oceno zahtevnosti. Ta postopek se nadaljuje, dokler vse zgodbe niso ocenjene, oziroma so ocenjene vse, ki so dovolj natančno definirane in imajo razvijalci dovolj informacij za začetek razvoja. Nato se o ocenah pogovorimo tudi s skrbnikom izdelka in argumentiramo svoje odločitve. Prav

tako se lahko skrbnik izdelka strinja ali ne strinja in skupaj z njim, če je to potrebno, določimo novo oceno. Za zgodbe, ki niso bile dovolj jasne, pridobimo potrebne podatke ali pa prestavimo zgodbo v naslednjo iteracijo. Glede na točke in morebitno odsotnost razvijalcev v tej iteraciji, določimo, katere zgodbe lahko za to iteracijo umestimo v seznam zahtev (angl. *sprint backlog*) in hkrati pustimo nekaj prostora za nujne zahteve, ki bi jih lahko v seznam zahtev z visoko prioriteto umestil skrbnik izdelka med iteracijo.

Enkrat na mesec (po dveh sprintih) poteka retrospektiva (angl. *sprint retrospective*), na kateri v razvojni skupini ocenimo preteklo delo in poiščemo ovire, ki so se v preteklem obdobju pojavljale. Skupaj ugotovimo, katere vrste odpada so se pojavljale (če so se) in kako jih v prihodnjih obdobjih odpraviti. Prav tako s pomočjo orodja za ugotavljanje kvalitete kode Teamscale [5] raziščemo, kaj bi bilo v prihodnje potrebno popraviti v že napisani kodi in odpravljanje nepravilnosti umestimo kot zgodbo v seznam zahtev. Orodje nam pomaga analizirati različne aspekte kvalitete kode, kot so dolžina funkcij, ponavljanje kode, globina gnezdenja kode, itd. Ko dokončamo zgodbe, ki smo jih s skrbnikom izdelka določili kot prioritetne, se lotimo kartic refaktoriranja kode, ki smo jih umestili v to iteracijo.

Vsak dan se razvijalci udeležimo dnevnega sestanka (angl. *daily standup*), kjer ostale seznanimo z delom, ki ga trenutno opravljamo. Sestanki potekajo vsako jutro ob istem času in sicer v našem primeru ob 9:30 in trajajo približno 15 minut. Vsak član tekom sestanka odgovori na naslednja vprašanja "Kaj sem počel včeraj? Kaj bom počel danes? Kakšne težave imam pri izpolnjevanju naloge?" Pogosto se našemu dnevnemu sestanku doda od 15 do 30 minut, kjer skupaj poskušamo odpraviti težave, na katere so razvijalci naleteli.

Približno enkrat do dvakrat na mesec naša skupina obišče naročnikovo podjetje v Nemčiji, kjer imamo sestanke s skrbnikom izdelka v živo. Termine obiskov poskušamo prilagoditi na čas pred novo iteracijo, da lahko takrat opravimo planiranje. Obiski trajajo od dva do tri dni, odvisno od količine obveznosti in sestankov. Kadar se planira veliko novih funkcionalnosti, po-

trebujemo več časa, da opredelimo vse zahteve.

Preglejevanje kode med razvijalci poteka vsakič, ko kateri iz med razvijalcev konča implementacijo zahtev iz uporabniške zgodbe. Ko odda zahtevo za združitev, vsi ostali člani dobimo obvestilo. Ko kodo pregledamo vsi člani in jo potrdimo, gre lahko kartica v testiranje. S tem dosežemo, da smo v vsako implementacijo vpleteni vsi člani, posredno ali neposredno. Tudi, če oseba zgodbe ni razvijala, je seznanjena s kodo, kar pomeni, da ima celotna skupina približno enako znanje o obeh aplikacijah. Če določen razvijalec zaradi različnih razlogov ne more dokončati naloge, lahko to prevzame drug. V tem projektu je to pomembno predvsem zato, ker tri osebe razvijamo dve aplikaciji in bi lahko ob odsotnosti enega razvijalca, razvoj ene od teh, zaradi pomanjkanja znanja, obstal.

Razvijalec mora vsak dan spremljati kanban tablo in jo glede na svoje delo posodabljati, prav tako pa mora biti ves čas pozoren na prekoračitev omejitve WIP.

4.8 Spremljanje uspešnosti

Uspešnost lahko merimo na več načinov: glede na porabljen čas, kvaliteto ter pogostost pojava različnih vrst odpada.

4.8.1 Merjenje potrebnega časa

Potrebni čas je čas, ki ga kartica potrebuje od prihoda v določen stolpec do zaključka izvajanja v enem iz med naslednjih stolpcev. Izbor stolpcev, na katerih izvajamo meritve, je odvisen od informacij, ki jih želimo izvedeti.

Potrebni čas bomo merili na tri načine z analizo različnih stolpcev na poslovni tabli, saj ta vsebuje vse stolpce, od takrat ko je kartica kreirana, do takrat ko je končana. Pri tem moramo imeti v mislih, da sta stolpca "in development" ter "code review" na poslovni tabli združena v stolpec "in progress".

- Analiza dostavnega časa - vključeni so vsi stolpci. To je povprečen čas potovanja kartice od prispetja na tablo, do zaključka. Dostavni čas zanima predvsem naročnika, saj je to čas od trenutka, ko je podal zahtevo, do trenutka, ko je prejel delujočo funkcionalnost.

- Analiza časa razvoja - vključeni so stolpci "in progress", "user acceptance test", "stage test", "done - on production". To je povprečen čas potovanja kartice od začetka razvoja do zaključka. Ta čas zanima vpletene v proces, saj je to čas, ki je bil porabljen za implementacijo in izvedbo do trenutka, ko se je delo na kartici zaključilo, brez časa, ko je kartica samo čakala na začetek razvoja.

- Analiza časa programiranja - stolpec "in progress". To je povprečen čas programiranja in pregledovanja kode. Ta čas zanima programerje, saj lahko spremljajo svojo hitrost programiranja v določenem obdobju.

S pomočjo teh različnih meritev ugotavljamo, kje na tabli se kartice zadržujejo najdlje in analiziramo, kako to pospešiti.

Jira nam ponuja dve vrsti grafov za spremljanje uspešnosti, diagram nadzora (angl. control chart) in kumulativni diagram delovnega toka (angl. cumulative flow diagram). S pomočjo obeh bomo analizirali uspešnost in primerjali meritve od začetka vpeljave in po treh mesecih uporabe.

4.8.2 Pozornost na pojav odpada

Naloga vsakega člana skupine je, da druge člane opozarja na napake in priporoča rešitve za izboljšanje tako kode kot procesa. S tem, ko drug drugemu preglejujemo kodo, je večja možnost za odkritje napak. Dogovorjeni smo tudi, da pišemo teste za vsako na novo dodano kodo. Prav tako so tudi dnevni sestanki namenjeni iskanju ovir (čakanje, premikanje), in ne samo pregledu trenutnega stanja. Vsak dan imamo v skupini na voljo 15 minut za pogovor, kjer smo dolžni opozoriti na nepravilnosti v kodi ali procesu in s tem preprečiti odpad. Poleg dnevnih sestankov poteka enkrat na mesec

retrospektiva, kjer povzamemo zadnja dva sprinta, poiščemo, kateri odpad se je pojavljal ter podamo predloge za v prihodnje.

Z analizo pojava odpada, lahko ocenimo našo uspešnost. Redkeje kot se odpad pojavlja, boljši je proces. Naš cilj je s pomočjo sestankov odpad zmanjševati in ugotavljati, če se količina odpada čez čas zmanjšuje.

Poglavje 5

Analiza uspešnosti vpeljave

5.1 Sestanki in vloge

Vsi trije člani smo se v preteklosti že srečali s Scrumom, zato so bili nam Scrum sestanki in vloge blizu. Kljub temu pa ni bilo vedno vse po pričakovanjih in bi lahko marsikaj še izboljšali.

5.1.1 Dnevni sestanki

Dnevni sestanki so nam od vsega predstavljali največje probleme. Sestanki naj bi potekali vsak dan ob enakem času na istem mestu, kar je nam predstavljalo prvi problem. Prvi mesec smo imeli vsak delavnik ob 9:30 rezervirano sobo za sestanke. Ker se je pogosto dogajalo, da je kdo od nas manjkal ali zamudil, sta ostala dva člana opravila sestanek kar v pisarni. Nato so rezervacijo prevzele druge skupine, kar je pomenilo, da smo ostali brez sobe za sestanke. Poleg tega so poleti časi dopustov in smo zadnja dva meseca opravljali dnevne sestanke še redkeje. Na srečo smo se vsi zavedali prihajajočih odsotnosti in smo bili med tem časom vsi dovolj seznanjeni s svojimi dolžnostmi in imeli večinoma dovolj informacij za nemoteno delo.

Ugotovili pa smo, da bi se nam na dolgi rok lahko poznala neorganiziranost in bomo morali to v prihodnje izboljšati. V tednih, ko smo cel teden vestno opravljali dnevne sestanke je bila komunikacija med nami boljša, smo

se lažje skupaj soočili s problemi in jih reševali takoj ko so nastali.

5.1.2 Planiranje iteracije

Planiranje iteracije je bilo najuspešnejše, kadar smo to opravili na skupni lokaciji s skrbnikom izdelka, saj planiranje preko video klicev vzame veliko več časa zaradi prekinitev in motenj. S pomočjo teh sestankov smo si, sploh v poletnih mesecih, lažje razdelili delo in razčlenili zahteve, da je bilo kasneje čimmanj zapletov. Potekali so po pričakovanjih in nam olajšali delo čez celo iteracijo, ker smo imeli delo dobro načrtovano in razdeljeno.

5.1.3 Retrospektiva

Sestanki retrospektive so potekali uspešno in nam pomagali pri izboljšanju nekaterih delov procesa. S pogovorom in analizo smo odkrivali odpad, ki se je med delom pojavljal ter ga uspešno zmanjševali, kot je opisano kasneje. Izboljšala se je predvsem kvaliteta naše kode, saj smo vsak mesec pripravili nekaj zahtev za izboljšave kode iz prejšnjega meseca. Tako smo se bolj redno že sproti držali pravil kvalitetne kode in nam ni bilo potrebno stvari popravljati kasneje.

5.2 Zmanjševanje odpada

Delno dokončano delo za nas ni predstavljajo problema. S pomočjo planiranja iteracij smo ves čas reševali le zahteve, za katere smo vedeli, da morajo biti končane. Prav tako smo predčasno ugotovili, katerih zahtev v tistem trenutku ni mogoče rešiti in jih prestavili v naslednje iteracije. Redko smo začeli delo na zgodbah, ki niso bile dovolj dobro definirane ali pa smo, kar se je dalo hitro, opozorili skrbnika izdelka na manjkajoče informacije in začasno preložili zgodbo.

Dodatni procesi so problem predvsem pri posameznikih ali v skupinah, ki se še učijo. Kot skupina smo ugotovili, da smo na začetku porabili

veliko več časa in energije za sestanke, kot kasneje. Prav tako sem jaz kot posameznik porabila več časa za reševanje zahteve na začetku, saj sem se morala pogosto vračati na popravljanje kode iz prejšnje zahteve. Včasih se je zgodilo, da nisem sledila kodnim standardom ali nisem dobro razumela vseh zahtev in sem morala kodo popravljati kasneje. Šele, ko so mi bila pravila in proces blizu, sem lahko učinkovito izvrševala naloge. Prav tako smo vsi v skupini porabili nekaj dodatnega časa za pravilno uporabo Jire in smo morali nekatere postopke izvajati dvakrat, kar pa so dodatni procesi, torej odpad. S pomočjo sestankov smo se redno opozarjali na pravilno uporabo in pravila ter v treh mesecih optimizirali proces.

Na dodatne funkcionalnosti smo morali opozarjati predvsem skrbnika izdelka, saj nam je večkrat v sklopu zgodbe želel dodati funkcionalnosti, ki jih uporabnik mogoče sploh ne bi uporabljal, nam pa bi vzele relativno veliko časa za implementacijo. Po pogovoru se je skoraj vedno strinjal, da nekatere funkcionalnosti niso potrebne in smo implementacijo preskočili. Čez čas se je to izboljševalo in sedaj se takšni problemi skoraj ne ponavljajo več.

Preklapljanje med nalogami smo na stolpcu "in progress" preprečili že od začetka z omejitvijo WIP. Ves čas smo bili pozorni na omejitve ter poskušali delati eno nalogo na enkrat. Preklapljanje je predstavljalo problem le kadar se je nabralo več nalog v stolpcu "code review" in nismo vestno spremljali obvestil. Tako smo morali ustaviti delo, ki ga opravljamo in najprej pregledati zgodbe, ki so ustvarjale ozko grlo v stolpcu "code review". Problem bi lahko mogoče odpravili z drugačnim sistemom opozoril o čakajočih zgodbah, ki bi jim lažje sledili.

Čakanje je večinoma problem v času dopustov. V različnih stolpcih so se dogajali zastoji zaradi pomanjkanja prisotnosti zaposlenih. Kljub vnaprejšnjemu planiranju in nadomestnim ljudem, se je zgodilo, da so se zaradi pomanjkanja ljudi ali pomanjkanja znanja nadomestnih de-

lavcev stvari ustavile. Predvsem pade velika odgovornost na prisotno osebo, ki ima trenutno največ znanja o procesu ali poslovnih zahtevah. To rešujemo na tak način, da predajamo znanje med seboj v skupini, kar pomeni, da je vedno prisotna vsaj ena oseba, ki ima nekaj znanja za reševanje problemov. V primeru vprašanj o naših dveh aplikacijah lahko katerikoli razvijalec odgovori na vprašanja, in ne samo tisti, ki jo primarno razvija, s čimer zmanjšujemo čas čakanja s strani naše skupine. Za prihodnje podobne situacije bo potrebna boljša organizacija in planiranje dopustov vnaprej tako, da bo vedno prisotna vsaj ena oseba, ki je odgovorna za določen proces. S takšnimi ukrepi se bo zmanjšal obseg čakanja.

Premikanje je velik problem predvsem z vožnjo v Nemčijo. Povprečna vožnja traja 4 ure v eno smer, kar je 8 ur skupaj. Prav tako je podjetje pri njih veliko večje in lahko traja samo premikanje med sejnimi sobami po 10 minut. Sliši se relativno malo, ampak ob petih sestankih na dan ta številka zelo naraste. Srečanja na skupnih lokacijah so zelo pomembna zaradi lažje komunikacije in osebnega stika. Problem smo začeli reševati tako, da smo obiske iz dvakrat mesečno zmanjšali na enkrat mesečno.

Napake odpravljamo s sprotnim testiranjem in medsebojnim pregledovanjem kode. Ker včasih ni mogoče napisati testov za vsak del kode, se napake težje odkrije. Z medsebojnim pregledovanjem tako odkrivamo napake, ki jih testi niso odkrili ali pa odkrijemo celo napake v testih, ki niso mogle biti odkrite na drugačen način.

Odpad poskušamo odkrivati vsak dan tako na dnevnih sestankih kot med delom. Na vsaki retrospektivi pa z analizo potrebnega časa ugotavljamo, kako bi delo še pohitrili in katere neodkrite vrste odpada nam morebiti povečujejo dostavni čas. Kljub stalnemu zmanjševanju odpada, se moramo zavedati, da ga ni mogoče povsem odstraniti.

Prvi mesec se je pojavljalo največ odpada, ki smo ga lahko uspešno odstranili. Predvsem smo, z boljšim poznavanjem metodologije, zmanjšali število dodatnih procesov. Prav tako je upadlo število napak zaradi vedno bolj učinkovitega pregledovanja kode. Sedaj se od vseh vrst odpada najpogosteje pojavlja premikanje, na katerega nimamo direktnega vpliva.

5.3 Kanban tabla

Ker pred tem še nisem uporabljala Kanbana, sem potrebovala dva tedna, da sem osvojila vse funkcionalnosti. Ker sta druga dva sodelavca tako Jiro kot Kanban že uporabljala, jima je bilo to veliko lažje.

Kljub temu, da sedaj vsak od nas že dobro pozna orodje in idejo kanban table, se dogaja, da občasno pozabimo posodabljati tablo. Predvsem je to prisotno pri manjših popravkih kode na zgodbah, kjer se skoraj "ne splača" premikati kartice v določen stolpec. Čeprav to izgleda povsem nedolžno, predstavlja problem, saj ostali člani skupine ne morejo vedeti, kaj razvijalec v tistem trenutnku počne. Prav tako Jira ne more pravilno beležiti časa, porabljenega na kartici, in so lahko grafi za analizo netočni.

Ločeni tabli za razvijalsko in poslovno stran sta nam v veliko pomoč. Pri načrtovanju iteracije, skupaj s skrbnikom izdelka, uporabljamo poslovno tablo, kjer si lažje predstavljamo celoten proces od prihoda kartice na tablo do zaključka. Razvijalska tabla nam je v pomoč pri razvoju, saj prikazuje le kartice s katerimi operiramo med iteracijo.

Pasovi na tabli nam pomagajo pri večji preglednosti, saj je hitro vidno katero vrsto dela opravlja kateri razvijalec. Prav tako lahko določen pas skrijemo, če nas v tistem trenutku ne zanima. Takšni primeri se pojavijo na začetku iteracije, ko nas zanimajo samo uporabniške zgodbe iz obeh aplikacij in ne zgodbe refaktoriranja kode. Kartice refaktoriranja lahko prikažemo, ko smo končali ostale prioritetne zgodbe.

Razen izjem, redno in pravilno uporabljamo tablo in nam je v veliko pomoč skozi celoten proces, tako na dnevni bazi kot v času sestankov.

5.4 Omejitev WIP

Na začetku smo se spraševali, ali omejitev sploh potrebujemo, saj se je vsak od nas zavedal, da je idealno, če en razvijalec dela največ na eni zahtevi naenkrat. Ker pa jo Kanban predpisuje, smo se jo vseeno odločili dodati. Izbrali smo omejitev 5, ki se je izkazala za ustrezno in je sploh ni bilo potrebno spreminjati. Število kartic v teku je ponavadi 3 do 5, kar kaže, da je omejitev 5 primerna, saj nihče ne ostaja brez dela in zahteve od prevzema v razvoj do produkcije, potujejo kar se da hitro. Če se slučajno zgodi, da je maksimalna omejitev dosežena, skupaj preverimo, kje je nastal zastoj ter ga čim hitreje poskušamo rešit.

Prekoračitev se je v tem času zgodila trikrat.

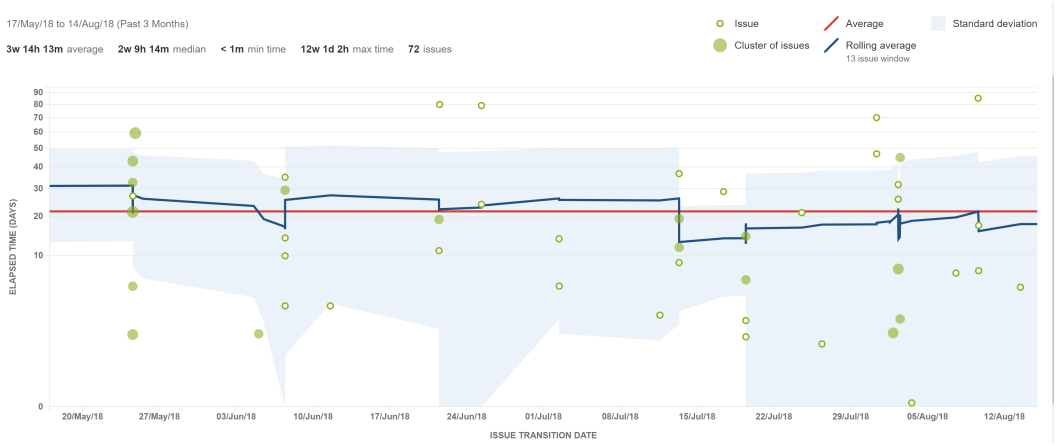
- Vsi trije razvijalci smo ob približno istem času končali delo na svoji kartici in jo premaknili v stolpec "code review". Takoj za tem je vsak vzel novo kartico iz stolpca "prioritized backlog". Pri zadnjemu, ki je to naredil, se je stolpec obarval rdeče, saj je prekoračil omejitev. Ta razvijalec je obvestil ostala dva člana, da so sedaj tri kartice v stolpcu "in development" ter tri kartice v stolpcu "code review", kar je skupno šest kartic. Dodal je tudi komentar o dogodku pod zadnjo prevzeto zgodbo. Po pogovoru je vsak član, takoj ko je lahko, pregledal kodo ostalih razvijalcev, kartice pa smo lahko kmalu premaknili v naslednji stolpec. Ker je to zelo redek dogodek, ni bilo potrebe po dvigu omejitve.

- V tem primeru, ki se je do sedaj zgodil dvakrat, je stolpec "code review" pred prekoračitvijo vseboval eno kartico. Dva razvijalca sta delala vsak na eni kartici, tretji pa je naenkrat delal na dveh povezanih karticah (zaledni del in čelni del) ter obe naenkrat končal. Po premiku kartic v stolpec "code review" in prevzetju nove kartice iz "prioritized backlog", je bila omejitev presežena. Postopek reševanja prekoračitve je bil enak kot v prejšnjem primeru. Tudi v tem primeru smo se dogovorili, da omejitve ni potrebno povečati na 6, saj razen v teh izrednih primerih, delo nemoteno teče.

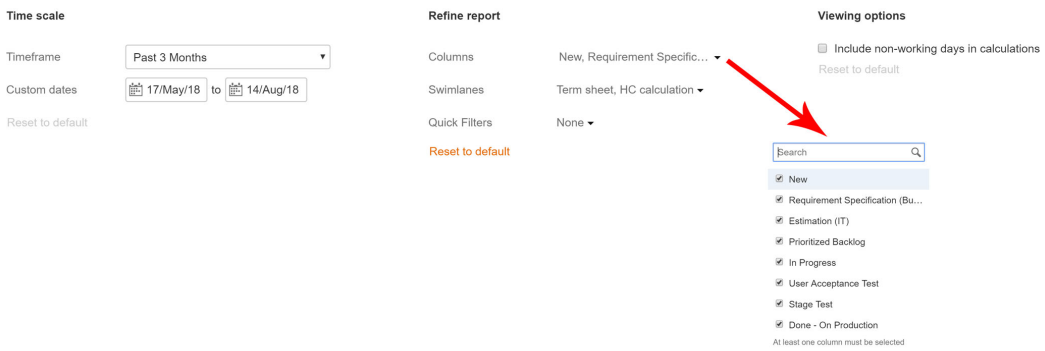
5.5 Analiza potrebnega časa in uspešnosti s pomočjo diagramov

Jira nam ponuja dva grafa analize: diagram nadzora in kumulativni diagram delovnega toka.

5.5.1 Diagram nadzora



Slika 5.1: Analiza dostavnega časa od 1. maja do 14. avgusta

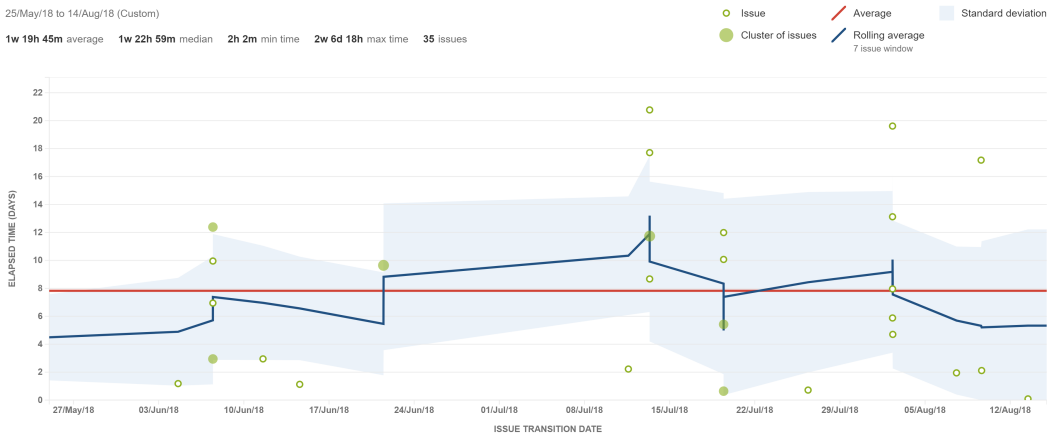


Slika 5.2: Nastavitve parametrov diagrama

Diagram nadzora je namenjen analizi potrebnega časa. S pomočjo tega diagrama lahko napovedujemo potrebni čas tudi v naslednjih iteracijah. Diagram dostavnega časa v obdobju opravljanja analize je prikazan na sliki 5.1.

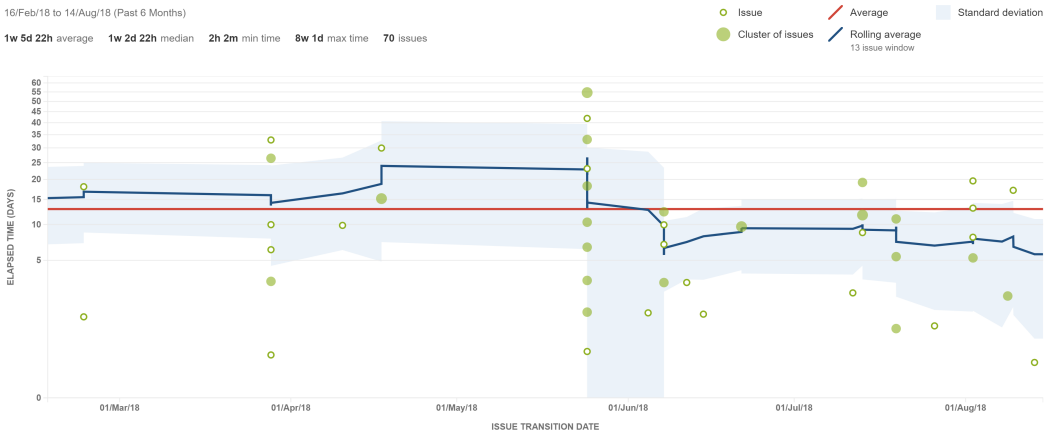
S pomočjo nastavitev, ki so prikazane na sliki 5.2, izberemo obdobje, v katerem želimo analizirati podatke, stolpce iz kanban table, pasove iz table in filter po ključnih besedah iz zgodb. Za nas sta najpomembnejša parametra čas in stolpci, saj ostali nimajo tako ključne fukncije. Pasovi, na primer, samo delijo zgodbe v sklope dela (prva aplikacija, druga aplikacija, refakturiranje, priprava podatkov za BI) in jih uporabljamo le zaradi preglednosti.

Za analizo diagramov so izbrana različna obdobja. Kot je bilo omenjeno že na začetku, uporabljamo za delo dve kanban tabli, ki imata med seboj nekoliko različne stolpce z istimi karticami. Za analizo sem vzela poslovno tablo, saj so na njej vsi stolpci od nastanka kartice do končanega dela in lahko na njej merimo potrebni čas. Pozorni moramo biti le na stolpec "in progress", ki je na razvijalski tabli razdeljen na dva: "in progress" in "code review", kar pa za analizo potrebnega časa ni pomembno.



Slika 5.3: Analiza časa razvoja za obdobje 25. maj–14. avgust

V projekt se je ta metodologija začela uvajati 1. maja 2018, ampak so se pravilne meritve potrebnih časov začele pojavljati šele po 24. maju. Projekt se namreč izvaja že nekaj let in se je za vodenje že nekaj časa uporabljala Jira,



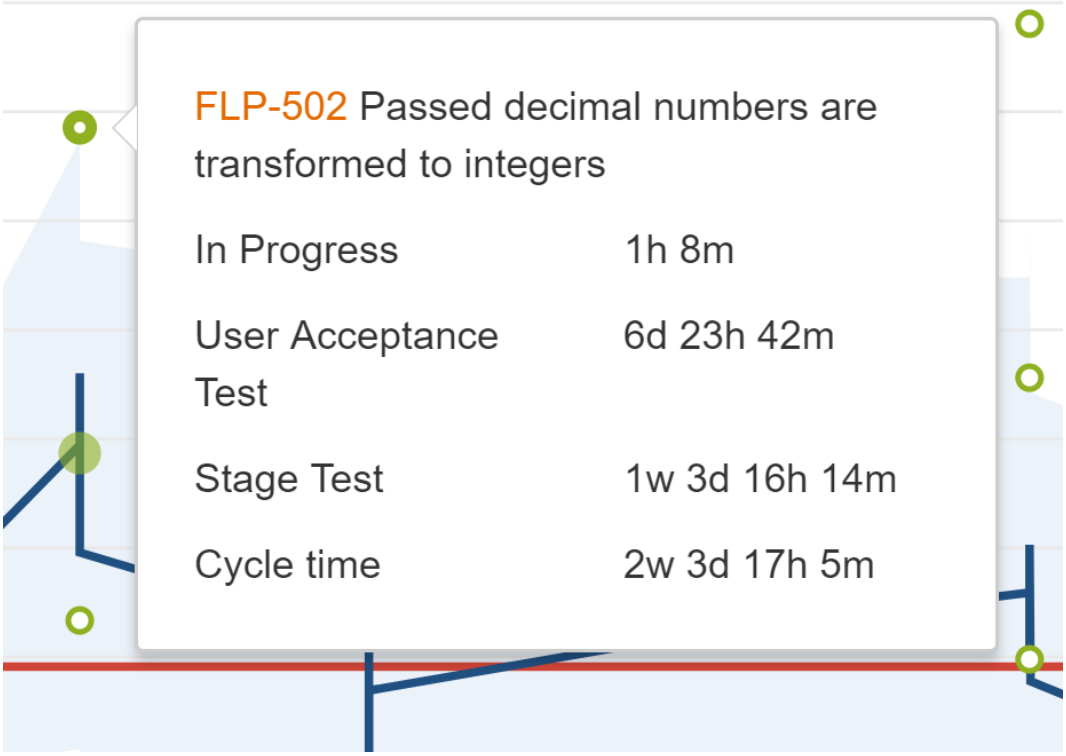
Slika 5.4: Analiza časa razvoja za obdobje 16. februar–14. avgust

ampak z drugačnimi metodologijami in pristopi. Zadnja iteracija prejšnjega načina dela je potekala od 12. aprila do 24. maja, med katero je potekala menjava zaposlenih in metodologije na projektu. V tem obdobju sem v stari skupini zamenjala prejšnjega člana, dva sta ostala ista. V začetku maja smo pričeli z načrtovanjem novega načina dela in mojim uvajanjem na projekt. Prva nova iteracija se je tako začela 25. maja in od takrat lahko merimo našo uspešnost na grafu 5.3. Za primerjavo je dodan tudi graf za zadnje pol leta na sliki 5.4, kjer je vidna očitna izboljšava po uvedbi novega načina dela, saj se je čas kartic, v stolpcih od prevzema v razvoj do objave v produkcijo, izboljšal.

Modra črta prikazuje spremembe povprečnega časa razvoja zgodb, kjer je svetlo moder pas standardna deviacija. Tanjši kot je pas, manjše so razlike in nihanja povprečnih časov ter bolj natančno lahko predvidevamo hitrost v prihodnje. Rdeča črta je povprečje celotnega grafa. Majhni zeleni krogi predstavljajo zgodbe, ki imajo večje odstopanje, večji zeleni krogi pa prikazujejo skupino izstopajočih zgodb. Z analizo iztopajočih lahko v prihodnje takšne dogodke preprečimo.

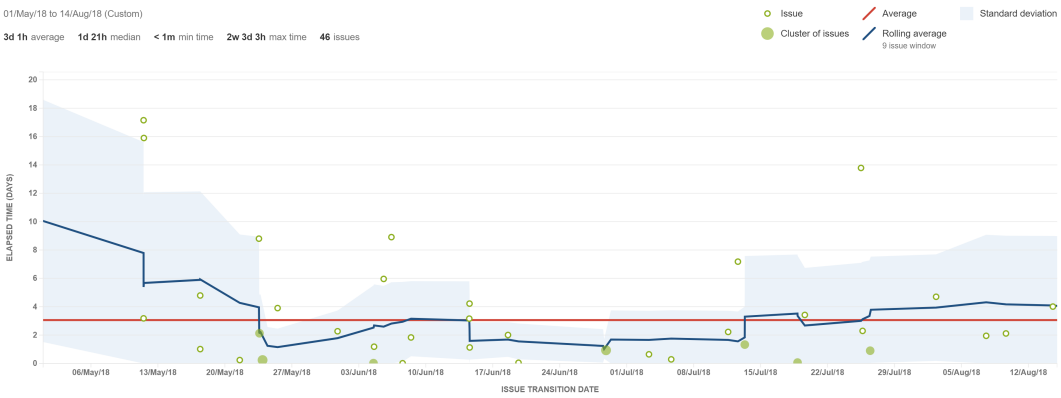
Graf na sliki 5.3 prikazuje, da je bil povprečen čas razvoja zgodb v stolpcih od ”in progress” do ”done - on production” v celotnem obdobju približno 8

dni (rdeča črta) in je bil največji v juliju. Za več informacij o izstopajočih zgodbah lahko kliknemo na zeleno piko na sliki 5.5, kjer je vidno, da problem ni bil na strani razvijalcev, vendar na strani ljudi, ki testirajo, saj je bila zgodba samo v testiranju dva tedna. V tem primeru vemo, da so bili takrat zaposleni zaradi dopustov odsotni. Prikazan je tudi graf časa programiranja, kjer lahko vidimo povprečne čase programiranja naše skupine, kar se vidi na sliki 5.6.



Slika 5.5: Analiza izstopajočih zgodb

V začetku maja, kjer smo se šele uvajali, je že viden postopen padec časa. V mesecu juniju in juliju je bila naša hitrost bolj ali manj konstantna. Hitrost se je poslabšala v zadnjem obdobju, saj se je zaradi odsotnosti ljudi od katerih dobivamo informacije o zahtevah, nekoliko povečala količina čakanja. Prav tako je na vsebino grafa vplivalo premikanje, saj smo bili v obdobju 31.7–2.8 v Nemčiji).



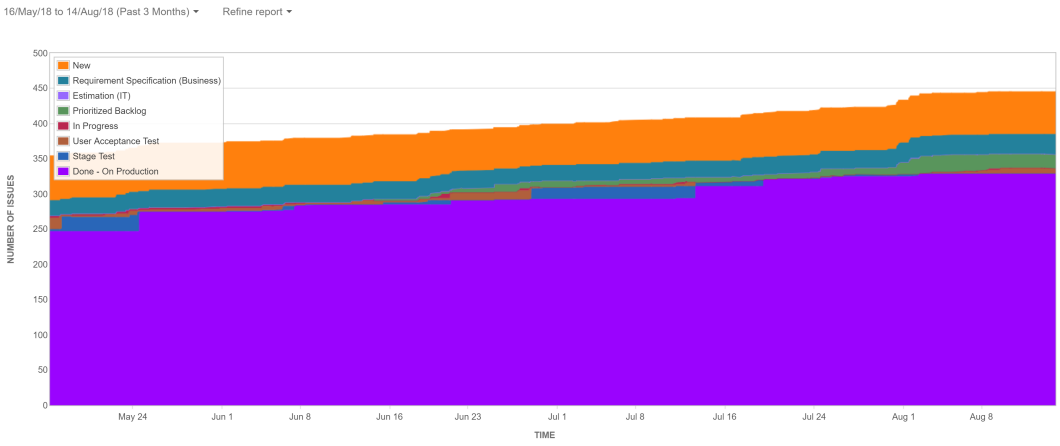
Slika 5.6: Analiza časa programiranja za obdobje 1. maj–14. avgust

Če pogledmo graf na sliki 5.6 natančneje, sta na levi strani vidni dve zgodbi, ki izstopata in sta bili v delu približno dva tedna. Obe sta bili povezani z eno zahtevnejših implementacij pošiljanja podatkov med obema aplikacijama in njihov prikaz. Čas za izdelavo bi tukaj težko zmanjšali.

Na desni je vidna še ena izstopajoča zgodba, kjer smo jo znotraj skupine vzeli v razvoj in med delom ugotovili, da nimamo dovolj informacij. Med čakanjem bi morali zgodbo prestaviti nazaj v stolpec "backlog", saj "in progress" pomeni, da razvijalec trenutno dela na tej zgodbi, kar pa tukaj ni držalo. Torej lahko sklepamo, da sta se v tem primeru pojavila dva problema. Prvi je bil pomanjkanje informacij, kjer je lahko kriv razvijalec, ker tega ni ugotovil ali pa skrbnik izdelka, ki ni dovolj natančno definiral zahtev. Drugi problem je bila nenatančna uporaba table. Eden iz med razlogov, da napak nismo pravočasno odkrili, bi lahko bilo pomanjkanje dnevnih sestankov, kjer naj bi se vsak dan seznanili s stanjem.

Kljub nepravilnostim in nihanjem na zgornih grafih, lahko na grafu dostavnega časa na sliki 5.1 vidimo, da je se v zadnjih treh mesecih dostavni čas izboljšal in je v obdobju zadnjega meseca ves čas pod povprečjem. To pomeni, da celoten proces počasi izboljšujemo in zgodbe od nastanka hitreje preidejo na produkcijo.

5.5.2 Kumulativni diagram delovnega toka



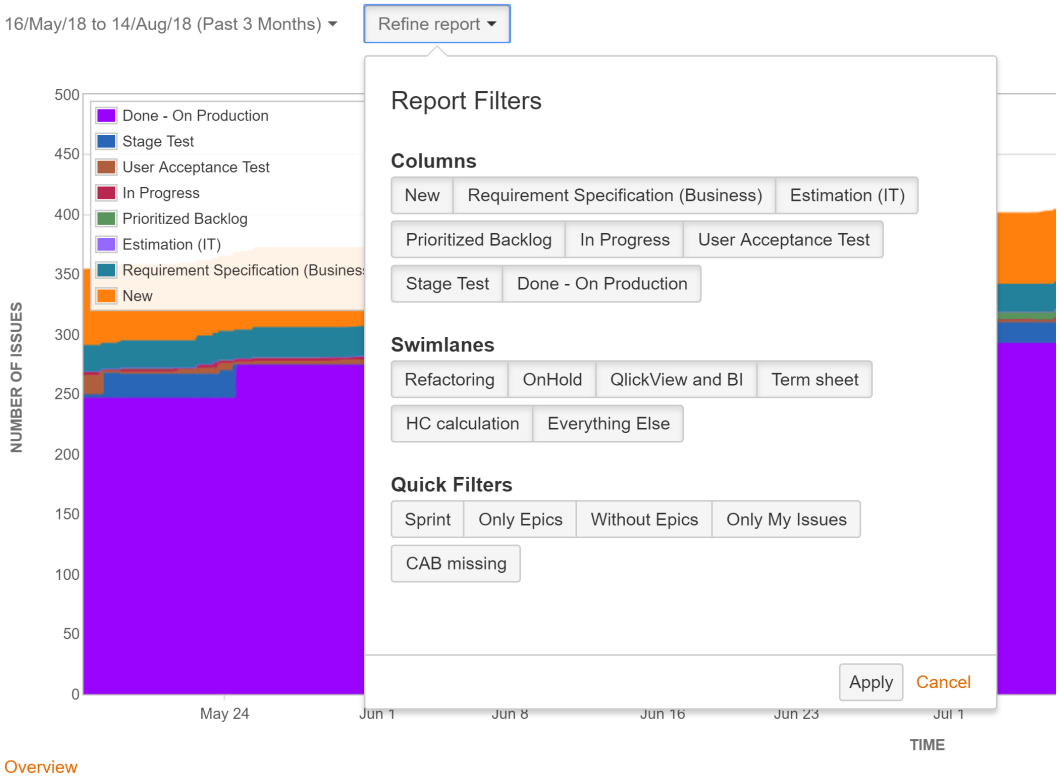
Slika 5.7: Analiza števila zgodb v stolpcih v zadnjih treh mesecih

V tem diagramu ne analiziramo časa izdelave funckionalnosti, ampak število zgodb, ki so bile v določenem trenutku v posameznih stolpcih.

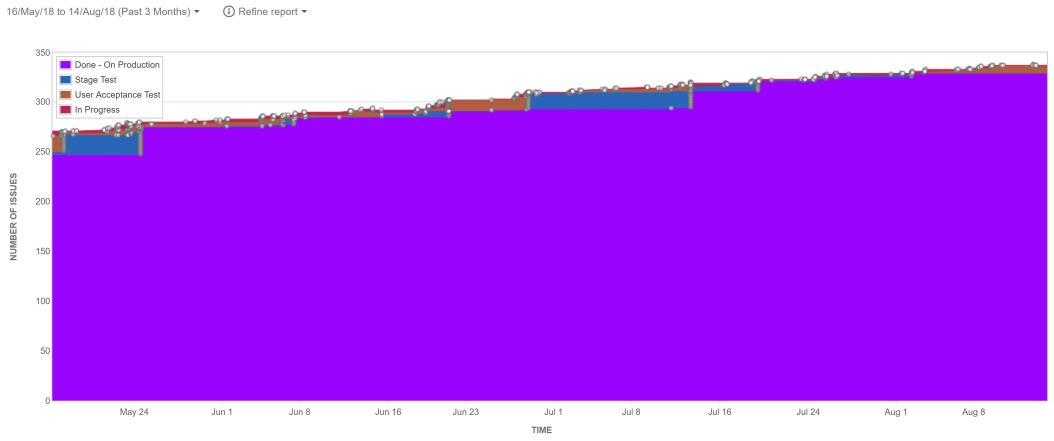
Graf na sliki 5.7 prikazuje stanje zadnjih treh mesecev. Najbolj opazni sta dve stvari: število zgodb v celotnem grafu in njegovo naraščanje. Število zgodb se skrajno levo ne začne na 0, saj so tukaj že zgodbe iz projekta, ki so se izvajale, preden smo začeli z našo metodologijo. Stalno naraščanje grafa pomeni napredek, torej vedno več zgodb prihaja na tablo in vedno več se jih končuje.

Z uporabo filtrov na sliki 5.8 lahko izluščimo podatke, ki nas zanimajo. Za zadnje tri mesece lahko na grafu slike 5.9 vidimo, da je število končanih zgodb naraslo približno z 250 na 325, kar je povprečno 30 zgodb na mesec in 10 zgodb na razvijalca v enem mesecu, oziroma 15 zgodb na iteracijo.

Te informacije lahko uporabljamo za primerjavo števila končanih zgodb vsak mesec. Eden iz med problemov je, da za natančno analizo manjka v Jiri nekaj funkcionalnosti. Prva uporabna funkcionalnost bi bila možnost nastavitve števila zgodb na 0, kjer bi lažje videli napredek skozi izbrano obdobje. Druga funkcionalnost bi lahko omogočala prikaz večih podrobnosti za določen datum, ko bi se tja postavili s kurzorjem.



Slika 5.8: Filtri prikazovanja informacij na grafu



Slika 5.9: Graf napredka v stolpcih "in progress", "user acceptance test", "stage test", "done - on production"

Poglavje 6

Sklepne ugotovitve

Začetek dela na projektu, ki se je pred tem že nekaj časa izvajal, se je sprva zdel naporen. Ampak to je bila priložnost za uporabo Scrumbana in analizo uspešnosti procesa z novo metodologijo. Sedaj bolje razumemo prednosti in pasti uporabe različnih agilnih metodologij. Poleg novega osvojenega znanja, semo pripomogli k izboljšanju procesa in hitrejši dostavi produkta. Poleg analize uporabe metodologije, smo bolje spoznali tudi orodje Jira in katere funckionalnosti vsebuje. S svojim znanjem lahko sedaj pomagam drugim skupinam izboljšati proces, razumeti Jiro in grafe, ki jih redko katera skupina uporablja.

Ni pomembno, katero metodologijo uporabljaš, dokler veš, kako, zakaj in kdaj jo uporabljati. Sami smo uporabljali Scrumban, ki je hibrid med Kanbanom in Scrumom in se je za nas izkazal za uspešnega. Vsaka metodologija ni primerna za vsak projekt, zato je potrebno pred začetkom narediti nekaj raziskovanja in nato malo eksperimentirati, če si želimo uspešnega procesa. Tako Scrum, kot Kanban sta prilagodljiva in namenjena prilagajanju glede na zahteve projekta.

Opazno je, da uporabniki agilnih matodologij niso pogosto pozorni na odpad, ki je definiran v okviru vitkega razvoja. Tako sem tudi jaz ostala dva člana moje skupine, ki imata že več let izkušenj z uporabo agilnih metodologij, pogosto opozarjala na pojav odpada. Marsikdo namreč misli, da so

nekatere vrste odpada v razvoju normalne, ker se pogosto pojavljajo, vendar lahko veliko povzročajo škodo na dolgi rok. Ravno zato je potrebno proces redno analizirati.

Naš proces bi se dalo še izboljšati in pohitriti, z upoštevanjem nepravilnosti, ki smo jih odkrili tekom izvajanja. Kljub temu je vidno v poglavju analize, da nam je dosledna uporaba agilnih metodologij izboljšala čas dostave funckionalnosti izdelka. Čeprav so tudi pred tem uporabljali agilne pristope, je nepravila uporaba nekaterih praks vplivala na čas dostave funkcionalnosti, ki je bil pred našo uvedbo, glede na analizo diagramov, slabši.

Literatura

[1] 12th annual state of agile survey. Dosegljivo: https://stateofagile.versionone.com, 2018. [Dostopano: 21. 8. 2018].

[2] Jira. Dosegljivo: https://www.atlassian.com/software/jira, 2018. [Dostopano: 21. 8. 2018].

[3] Kanban vs. scrum. Dosegljivo: https://www.atlassian.com/agile/kanban/kanban-vs-scrum, 2018. [Dostopano: 21. 8. 2018].

[4] Learn kanban with jira software. Dosegljivo: https://www.atlassian.com/agile/tutorials/how-to-do-kanban-with-jira-software, 2018. [Dostopano: 21. 8. 2018].

[5] Teamscale. Dosegljivo: https://www.cqse.eu/en/products/teamscale/overview/, 2018. [Dostopano: 21. 8. 2018].

[6] Agile Alliance. Agile manifesto. Dosegljivo: http://www.agilemanifesto.org, 2001. [Dostopano: 21. 8. 2018].

[7] Marko Bajec and Marjan Krisper. Agilne metodologije razvoja informacijskih sistemov. *Uporabna informatika, Ljubljana, april, maj, junij*, 2003.

[8] Mike Cohn. *User stories applied: For agile software development.* Addison-Wesley Professional, 2004.

[9] Torgeir Dingsøyr, Sridhar Nerur, VenuGopal Balijepally, and Nils Brede Moe. A decade of agile methodologies: Towards explaining agile software

development. *Journal of Systems and Software*, 85(6):1213–1221, June 2012.

[10] Christof Ebert, Pekka Abrahamsson, and Nilay Oza. Lean software development. *IEEE Software*, 29(5):22–25, Sept.-Oct. 2012.

[11] Henrik Kniberg and Mattias Skarin. *Kanban and Scrum-making the most of both.* C4Media Inc., 2010.

[12] Viljan Mahnic. Improving software development through combination of scrum and kanban. *Recent Advances in Computer Engineering, Communications and Information Technology*, pages 281–288, 2014.

[13] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit: An Agile Toolkit.* Addison-Wesley, 2003.

[14] Winston Royce. Managing the development of large software systems. *Proceedings of IEEE WESCON*, pages 328–338, 1970.

[15] Ken Schwaber. *Agile project management with Scrum.* Microsoft press, 2004.

[16] Jeff Sutherland. *V gruču do uspeha–umetnost vodenja projektov z metodo SCRUM.* Pasadena, 2016.