UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Simon Pihler

Uvajanje konceptov vitkosti v proces razvoja programske opreme

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2015

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Simon Pihler

Uvajanje konceptov vitkosti v proces razvoja programske opreme

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: izr. prof. dr. Viljan Mahnič

Ljubljana, 2015

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo: Uvajanje konceptov vitkosti v proces razvoja programske opreme.

Tematika naloge:

Proučite značilnosti vitkega razvoja programske opreme in jih primerjajte z že znanimi agilnimi metodologijami. Podrobneje opišite metodo Kanban in njen pristop k skrajševanju potrebnega časa z omejevanjem količine dela v teku. Na podlagi pridobljenega znanja realizirajte preprosto orodje za vodenje projektov po metodi Kanban s poudarkom na vzdrževanju table z delovnimi nalogami.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Simon Pihler, z vpisno številko 63090136, sem avtor diplomskega dela z naslovom:

Uvajanje konceptov vitkosti v proces razvoja programske opreme

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Viljana Mahniča;
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela;
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 2. februar, 2015 Podpis avtorja:

Zahvaljujem se izr. prof. dr. Viljanu Mahniču za mentorstvo, vodenje, nasvete in pomoč pri izdelavi diplomskega dela. Prav tako se iskreno zahvaljujem svojim staršem Branki in Ivanu za neprecenljivo podporo, posluh in pomoč med študijem. Hvala tudi Maši, sorodnikom in prijateljem, ker so ves ta čas verjeli vame in me pri študiju vseskozi spodbujali.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
TPS	Toyota production system	Toyotin sistem proizvodnje
JIT	just in time	ravno ob pravem času
CFD	cumulative flow diagram	kumulativni diagram poteka
WIP	work in progress	delo v teku
VSM	value stream map	načrt toka dodane vrednosti
OOP	object oriented programming	objektno usmerjeno programiranje
MVC	model-view-controller	model-pogled-krmilnik
CSS	cascading style sheet	prekrivni slogi
HTML	hypertext markup language	označevalni jezik za oblikovanje večpredstavnostnih dokumentov
DBMS	database management system	sistem za upravljanje podatkovne zbirke
ORM	object relational mapping	objektno relacijsko preslikovanje

Povzetek

V diplomskem delu so predstavljeni in opisani principi vitke proizvodnje, ideje skrajšanega časa cikla razvoja izdelkov ter prenos principa eliminacije odpadkov v sam razvoj programske opreme. Definirane so tradicionalne in agilne metode razvoja, sledi primerjava agilnih in vitkih principov. Sama vpeljava konceptov vitkega razvoja je opisana z uporabo metode kanban. V kontekstu omejevanja dela v teku je opredeljen kumulativni diagram poteka, prav tako je prikazan vpliv spremenljivk diagrama na sam razvoj projekta. V diplomskem delu je pojasnjeno načrtovanje toka dodane vrednosti ter njegov vpliv na izključevanje odpadkov v procesu razvoja programske opreme. Sledi predstavitev spletne aplikacije, ki predstavlja sistem kanban. Pred opisom aplikacije so na kratko predstavljene uporabljene spletne tehnologije, na koncu pa prikazani nekateri zanimivejši deli kode, ki so bili pri izdelavi spletne aplikacije uporabljeni.

Ključne besede: vitki principi, razvoj programske opreme, kanban, agilne metode, koncepti vitkosti, kumulativni diagram poteka, načrt toka dodane vrednosti, spletne tehnologije, PHP ogrodje Laravel

Abstract

This BSc thesis presents and descibes the principles of lean manufacturing, ideas of shortened cycle time in product development and the transfer of principles regarding elimination of waste in the process of software development. It covers traditional plan driven and agile development methods, followed by a comparison of agile and lean principles. The introduction of the lean concepts into software development process is described using the Kanban method. In the context of limiting work in progress is defined cumulative flow diagram as well there is shown the impact of variables on development project. It also covers value stream process flow and its effect on the exclusion of waste in the process of software development. By the end there is a presentation of developed web application, which represents a simple Kanban system. There is also a brief presentation of used web technologies, followed by some interesting parts of code that was used in the development of Kanban web application.

Keyword: lean principles, software development, kanban, agile methods, lean concepts, cumulative flow diagram, value stream map, web technologies, Laravel PHP framework

Poglavje 1 Uvod

Ideje razvoja izdelkov in storitev po konceptih vitke proizvodnje se razvijajo že od konca 18. stoletja, pri oblikovanju proizvodnje linije pa je te ideje v začetku 20. stoletja Henry Ford prenesel v avtomobilsko industrijo [*1*]. Pred pričetkom izdelave Fordovega Modela T so za sestavo avtomobila potrebovali 12 ur, z izidom pa so z vpeljavo nekaterih konceptov proizvodnje ta čas zmanjšali na zgolj 90 minut. S tem je lahko Ford delavcem povečal plače z 2,4 dolarja za 9-urni delavnik na 5 dolarjev za 8-urni delavnik, kljub temu pa je podjetje še vedno od vsakega prodanega avtomobila imelo ogromen dobiček [*2*].

Za pravi začetek proizvodnje po konceptih vitkosti pa pravzaprav štejemo Toyotin sistem proizvodnje (angl. Toyota Production System, v nadaljevanju TPS), ki ga je v svoj sistem vpeljal japonski proizvajalec vozil Toyota Corporation z začetki nekje od leta 1960 naprej [*1*] [*2*]. Od takratnih proizvodnih sistemov se je ideja vitke proizvodnje razlikovala po tem, da je bila podrejena dejanski prodaji in ne ciljem proizvodnje [*3*]. To pomeni, da so izdelke proizvajali glede na trenutna naročila in ne na zalogo ter predvidevanja o številkah prodaje v prihodnosti. Slabost tega principa je v tem, da je potrebni čas (angl. lead time) izdelka daljši, kar lahko odvrne bodočega kupca od nakupa. Podjetje Toyota je rešitev iskalo v času cikla[1] (angl. cycle time), ki predstavlja čas, ki preteče od naročila do končnega izdelka. Optimiziralo ga je tako, da je iz proizvodnje vozil odstranilo odpadke (angl. waste). Ko razumemo, katere aktivnosti prinašajo dodano vrednost izdelka ali storitve, lahko določimo odpadke, ki predstavljajo vse ostalo [*4*].

Taiichi Ohno, nekdanji predsednik podjetja Toyota, v svoji knjigi [*5*] omenja zmanjšanje odpadkov kot osrednji princip, po katerem TPS deluje. Slednji princip je tisti, iz katerega v osnovi izvira večina ostalih, kot npr. proizvodnja ravno ob pravem času (angl. Just in Time, v nadaljevanju JIT), inteligentna avtomatizacija (angl. autonomation), odločitve dolgoročnih koristi na račun kratkoročnih finančnih ciljev, standardizacija opravil kot temelj za nadaljnje izpopolnjevanje in spodbujanje zaposlenih pri tem [*2*] [*6*].

[1] Pri uporabi izraza čas cikla lahko v literaturi srečamo različne razlage oz. definicije. Prva označuje čas, ki preteče od začetka do konca delovnega procesa. Druga definicija označuje povprečni čas, ki preteče od izhoda dveh zaporednih elementov v delovnem procesu [*37*].

1.1 Proizvodnja ravno ob pravem času

Proizvodnja JIT se za razliko od starejšega prepričanja, da bomo z optimizacijo učinkovitosti orodja ali mehanizacijo dosegli pozitiven učinek na produktivnosti, bolj nagiba k idejam o dvigu produktivnosti, le če proizvodnja deluje bolje kot celota [*2*]. Pravila TPS zahtevajo strojno opremo le takrat, ko je to potrebno, in ne narekujejo dejstva, da je za uspeh v proizvodnji nujno dvigniti storilnost strojev [*7*].

Polna izkoriščenost strojnih kapacitet lahko pomeni le lokalno povečevanje zaloge in ne nujno boljše produktivnosti ter hitrejše dobave izdelka do končnega kupca. V TPS previsoka lokalna zaloga praktično predstavlja odpadek.

1.2 Uspeh podjetja Dell s pomočjo konceptov vitkega razvoja

Podjetje Dell Computers se je pri proizvodnji računalniške opreme zavedalo hitro-spreminjajočega se trga ter visokega upada vrednosti računalniških komponent skozi čas na trgu, zato je koncepte vitkosti uporabilo v svojo korist. Z uvedbo sistemov JIT je podjetje Dell naredilo korak pred konkurenco in v 90. letih 20. stoletja postalo eno izmed najhitreje rastočih podjetij na področju računalniške opreme [*8*].

Treba je omeniti, da je možnih dejavnikov za uspeh veliko več kot zgolj vpeljava JIT in konceptov vitkosti, a vendar je podjetje Dell Computers eno izmed podjetij, ki so koncepte poleg podjetja Toyota uspešno prenesla v poslovni svet. Čas cikla je bil v obdobju 1996–2001 občutno nižji v primerjavi s konkurenčnimi podjetji, kot sta Compaq in IBM, zaloga komponent je bila neprimerljivo manjša, tržni delež pa se je povečal s 3,4 % na kar 24,9 % (Tabela 1.1 in Tabela 1.2).

	Dell	Compaq	IBM
Zaloga komponent	15 min	7–10 dni	10 dni
Zaloga rač. sistemov	3 dni	30 dni	25 dni
Čas izdelave rač. sistema	4 ure	15 dni	12 dni
Dobava do končnega kupca	3 dni	30 dni	25 dni

Tabela 1.1: Prikaz podatkov o zalogah med vodilnimi proizvajalci računalniške opreme v obdobju 1996–2001 [*8*].

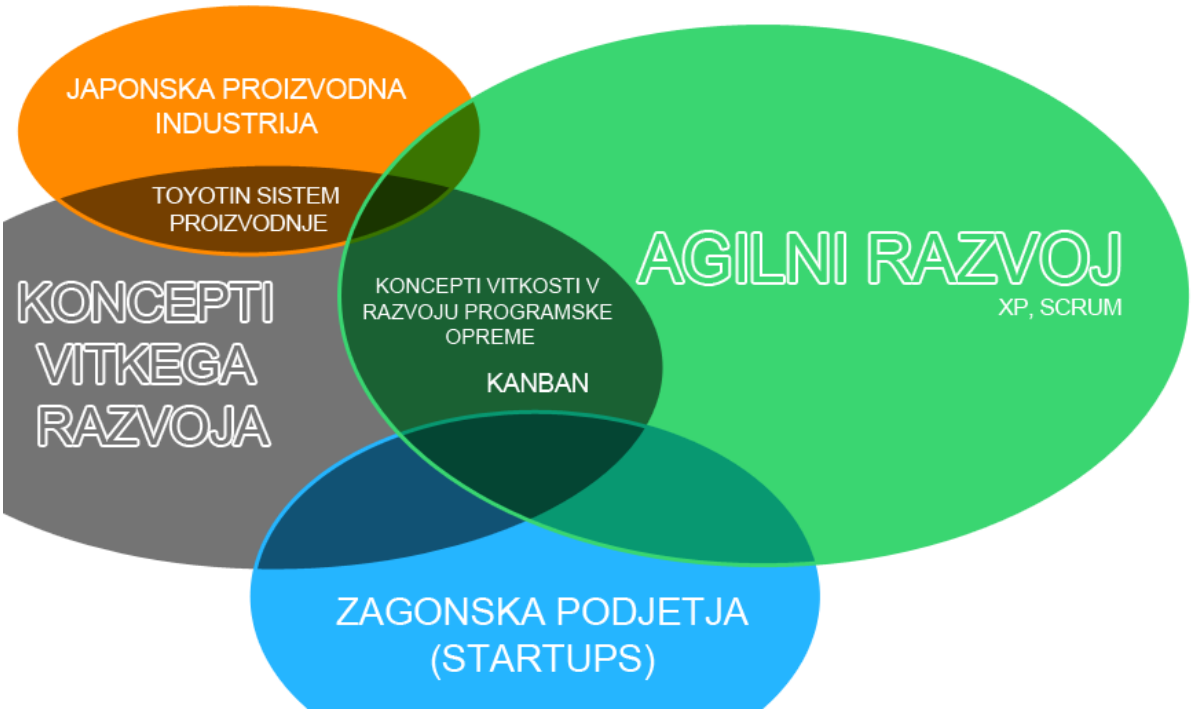
	1. četrtletje 1996	1. četrtletje 1997	1. četrtletje 1998	1. četrtletje 1999	1. četrtletje 2000	1. četrtletje 2001
Compaq	10,0 %	11,5 %	13,0 %	14,0 %	13,8 %	13.3 %
IBM	7,2 %	7,3 %	7,5 %	7,0 %	6,5 %	6,0 %
Dell	3,4 %	5,3 %	11,8 %	15,0 %	19,0 %	24,9 %

Tabela 2.1: Primerjava tržnih deležev rač. opreme med vodilnimi proizvajalci v obdobju 1996–2001 [*8*].

Poglavje 2 Koncepti vitkosti v razvoju programske opreme

2.1 Izvor in začetki

V sredini 90. let sta James P. Womack in Daniel T. Jones v svoji knjigi [*9*] predstavila pet (5) glavnih stebrov idej vitkosti, in sicer (1) vrednost (angl. value), (2) tok vrednosti (angl. value stream), (3) pretok (angl. flow), (4) izvlečni sistem (angl. pull) in (5) popolnost (angl. perfection).

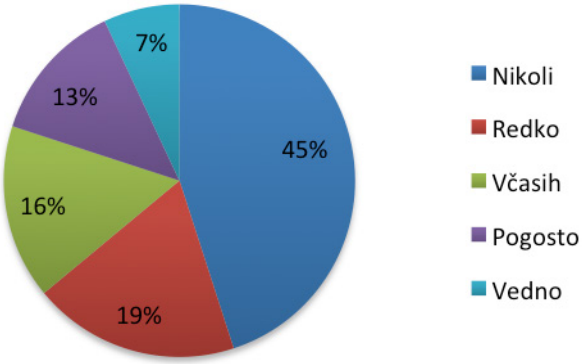


Slika 2.1: Prikaz odvisnosti različnih paradigem na področju proizvodnje po konceptih vitkosti.[2]

[2] Avtor diagrama je Yasunobu Kawaguchi, dosegljivo 16. decembra 2014 na spletni strani http://agilelion.com/agile-kanban-cafe/agile-and-lean-influences-where-did-kanban-scrum-scrumban-come-from.

Ključen poudarek pri konceptih vitkosti je torej pozornost na dodajanje vrednosti izdelku ali storitvi, proizvajalec izdelka pa želi biti pri opredelitvi dodane vrednosti objektiven. Dodatne funkcije izdelka so nezaželene, saj prinašajo dodatne stroške pri razvoju, predvsem v smislu kompleksnosti sistema in nadaljnjega vzdrževanja, kar v sam razvoj prinaša manjšo produktivnost, seveda v odvisnosti od tega, kako se s kompleksnostjo sistema spopadamo. Kot poslovni model koncepti vitkosti omogočajo hitrejše investicijske cikle, posledično višjo donosnost naložbe (angl. return on investment), prav tako tudi hitrejši odziv na dogajanje na trgu [*10*].

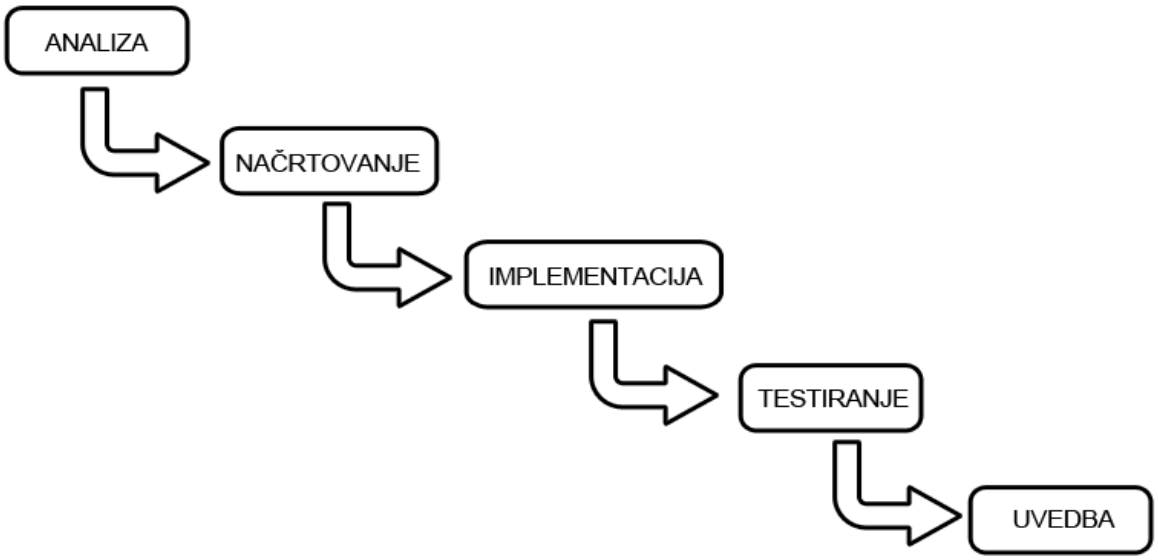
Ideje so se kmalu razširile na različna področja gospodarstva. V začetku 90. let so se kot alternativa tradicionalnim plansko vodenim metodam v domeni programske opreme začele razvijati agilne, ki so v tesnem sorodstvu z razvojem programske opreme po konceptih vitkosti. Pri uporabi tradicionalno plansko vodenih metod je razvoj temeljil na podlagi predvidevanj, da so zahteve stabilne in se tekom razvoja ne spreminjajo. Kljub temu se je hitrost sprememb v samem razvoju od 70. let, ko je Winston W. Royce v svojem članku [*11*] predstavil slapovni model (angl. waterfall model), izredno spremenila, kar je posledično pripeljalo do razvoja agilnih metod, ki omogočajo več svobode, dinamičnosti, prilagodljivosti, boljše sodelovanje z naročnikom, manj dokumentacije, boljše osredotočanje razvijalcev na produktivno delo, boljše ocenjevanje nadaljnjega razvoja, bolj očitno poslovno vrednost izdelka, večje zadovoljstvo razvijalcev pri delu, prav tako lahko tradicionalen plansko voden razvoj predstavlja večje tveganje pri stroških izdelave ob napačnem načrtovanju [*12*] [*13*]. Ena izmed študij je prišla do zaključka, da je pri programski opremi, izdelani s pomočjo tradicionalno plansko vodenih metodologij, kar 45 % funkcij, ki niso nikoli uporabljene, 19 % pa zelo redko (Slika 2.1) [*14*].



Slika 2.1: Prikaz deleža neuporabljenih funkcij v projektih, ki uporabljajo plansko vodene metode razvoja programske opreme [*14*].

2.2 Tradicionalno plansko vodene in agilne metode razvoja programske opreme

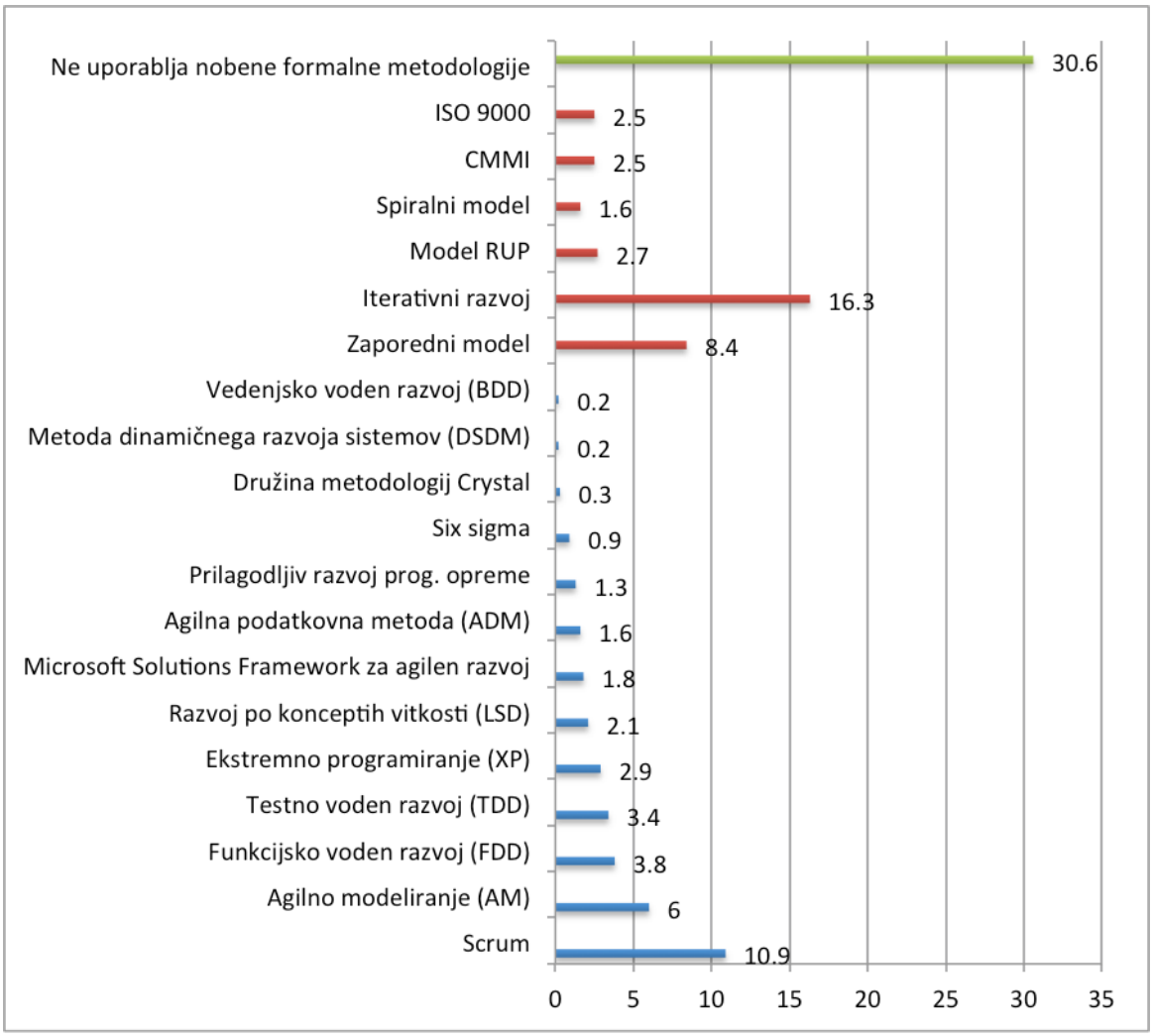
Tradicionalno plansko voden razvoj (Slika 2.2) namenja velik del aktivnosti planiranju, strukturi in dokumentaciji ter analizi, iz katere sledi podrobno načrtovanje. Zahteve naročnika so največkrat zapisane v dokumentih in odstopanja od začetnih načrtov so največkrat zelo nezaželena. Napredek razvoja se meri na osnovi plana. Faze, po katerih razvoj poteka, so po navadi analiza zahtev, načrtovanje na visokem nivoju, načrtovanje na nižjem nivoju, implementacija in kodiranje, testiranje enot, integracijsko testiranje, sistemsko testiranje in izdaja [*15*]. Modeli, ki so se razvijali od začetka 70. let, so (1) inkrementalni (angl. incremental model), (2) slapovni (angl. waterfall model), (3) spiralni (angl. spiral model) in (4) v-model (angl. v-model) [*16*].



Slika 2.2: Prikaz zaporednega modela razvoja programske opreme.

Agilne metode so svojo obliko dobile z izidom Manifesta agilnega razvoja programske opreme (angl. Manifesto for Agile Software Development). Glavni principi agilnega razvoja so (1) posamezniki in interakcije pred procesi in orodji, (2) delujoča programska oprema pred vseobsežno dokumentacijo, (3) sodelovanje s stranko pred pogodbenimi pogajanji in (4) odziv na spremembe pred togim sledenjem načrtom [*17*]. Dandanes sta najbolj prepoznavni agilni metodologiji razvoja scrum [*18*] in ekstremno programiranje (angl. extreme programming) [*19*]. Anketa o uporabi metodologij leta 2009 je pokazala, da 35 % podjetij (med njimi tudi IBM, HP in Microsoft) v domeni uporablja agilne metode razvoja, med katere zajema tudi

koncepte vitkega razvoja [*20*]. Pri tem je treba poudariti, da je 30,6 % anketirancev odgovorilo, da ne uporabljajo nobene formalne metodologije (Slika 2.3). S Slike 2.3 lahko razberemo, da je število razvijalcev, ki uporabljajo agilne metode, preseglo tiste, ki uporabljajo tradicionalno plansko vodene metode.



Slika 2.3: Prikaz deleža uporabe (v procentih) posameznih metodologij, kjer je sodelovalo 1298 anketirancev[3] [*21*].

Za tradicionalno plansko vodene metode razvoja je značilno, da razvoj projektov temelji na vnaprej pripravljenih načrtih, medtem ko agilne metode razvoj sproti prilagajajo. Čeprav se zdi, da so agilne metode in z njimi tudi nekateri koncepti vitkega razvoja dobile zagon, da postanejo vodilne metode razvoja programske opreme, niso popolnoma brezhibne. Negativen vpliv agilnosti lahko najdemo v premajhni pozornosti pri načrtovanju in arhitekturi, prav tako lahko odnos z uporabnikom oz. naročnikom v nekaterih primerih zmanjša produktivnost ekipe

[3] Forrester/Dr. Dobb's Global Developer Technographics. Anketa je bila opravljena v letu 2009.

[*12*]. Na drugi strani je tradicionalno plansko voden razvoj stabilen in predvidljiv, ki konsistentno sledi začetnemu načrtu s standardi, meritvami in kontrolo, vendar neprilagodljiv na spremembe [*22*]. Agilne metode razvoja poudarjajo pogostejšo izdajo, medtem ko pri tradicionalno plansko vodenih metodah programsko opremo izdajamo ob koncu projekta oz. redkeje, v daljših časovnih presledkih. Pri tem je treba poudariti, da agilne metode praviloma niso namenjene velikim razvojnim skupinam. Kljub temu imamo primere, kjer velike skupine delujejo po principih agilnih metodologij. Eno izmed njih je podjetje Spotify [*23*].

Zmotno je prepričanje, da agilne metode razvoja ne zahtevajo načrtovanja. Če v klasičnem zaporednem modelu programsko opremo v celoti načrtujemo in predvidevamo vnaprej, pri agilnih metodologijah to enoto načrtovanja prilagodimo eni iteraciji (odvisno od metodologije). Velike razlike so tudi v sami dokumentaciji programske opreme; ker so agilne metode tekom razvoja zelo prilagodljive naročnikovim zahtevam, je pisanje dokumentacije vnaprej tako kot pri tradicionalno plansko vodenih metodah velikokrat nesmiselno in predvsem drago [*24*].

2.3 Principi vitkosti v razvoju programske opreme

Z izidom knjige [*13*] je koncept vitkosti dobil prostor tudi v domeni programske opreme. Definicija je težavna, saj ne oblikuje točno določenih metod in večinoma abstraktno opisuje delovanje v primerjavi z ostalimi tradicionalno plansko vodenimi metodami razvoja. V knjigi [*13*] je zapisanih sedem (7) principov in dvaindvajset (22) orodij. Avtorja poudarjata, da se principi tekom časa ne spreminjajo, spreminjajo pa se orodja, kako te principe v praksi realiziramo [*25*]. Pomembno se je namreč zavedati, da je domena programske opreme relativno široka in se hitro ter vseskozi spreminja. Kot primer lahko vzamemo sam premik tehnologij mobilnih naprav in letnice izida knjige [14] (2003). Čeprav so koncepti vitkosti razvoja relativno novi, se njihova implementacija skupaj s časom in tehnologijo spreminja, sama paradigma razvoja ostaja enaka.

Mary Poppendieck in Tom Poppendieck sta definirala sedem (7) glavnih principov:

1) odpravljanje odpadkov (angl. eliminate waste),
2) pospešeno učenje (angl. amplify learning),
3) čim bolj pozne odločitve (angl. decide as late as possible),
4) čim hitrejša dostava (angl. deliver as fast as possible),

5) povečanje pooblastil ekipe (angl. empower your team),

6) vgraditev integritete (angl. build integrity in),

7) pregled nad celoto (angl. see the whole).

Za razumevanje implementacije razvoja programske opreme in njihovih orodij je pomembno, da razumemo principe in koncepte, po katerih se ta orodja oblikujejo.

2.3.1 Odpravljanje odpadkov

Glavni princip, iz katerega izhajajo načeloma vsi drugi, je odpravljanje odpadkov. Kot smo že omenili, je odpadek nekaj, kar ne doda dodatne vrednosti izdelku za uporabnika. To pomeni, da je treba odpadke najprej prepoznati in jih nato odpraviti.

Primeri odpadkov pri razvoju programske opreme, ki jih lahko primerjamo z odpadki v sami proizvodnji, so (1) nedokončano delo (angl. partially done work), (2) dodatni procesi (angl. extra processes), (3) dodatne funkcije (angl. extra features), (4) preklapljanje med procesi (angl. task switching), (5) čakanje (angl. waiting), (6) gibanje (angl. motion) in (7) napake (angl. defects) [*13*].

Velike finančne stroške v razvoju programske opreme predstavljajo stroški nedokončanega dela, kar praktično pomeni del kode, katero funkcionalnost izključimo, ko dostavimo programsko opremo do uporabnika. Ravno to je po navadi težava plansko vodenih metod razvoja, saj so predvidevanja o rešitvah poslovnih problemov vnaprej velikokrat napačna.

Dodatne procese označujemo kot procese, ki jih tradicionalne metode razvoja programske opreme zahtevajo, vendar ne dodajajo nobene dodatne vrednosti na strani uporabnika. Primer tega so dokumenti za odobritev sprememb zahtev naročnika.

Mnogokrat pri implementaciji dodatnih oz. nepotrebnih funkcij razvijalci povečujejo stroške in kompleksnost sistema, poleg tega ne smemo pozabiti, da je prav to lahko izvor težav in napak pri delovanju programske rešitve, kar lahko načeloma predstavlja celo dvojne stroške (implementacija in odprava).

Nepotrebne stroške pri razvoju prispeva tudi čakanje na ostale procese. Po navadi pride do stroškov zaradi dokumentacije, pri pregledih, spremembah zahtev in testiranju.

Pri gibanju imamo predvsem v mislih prenos informacij med udeleženci razvoja programske opreme. Če informacije, ki jih razvojna ekipa potrebuje, niso na voljo, to predstavlja odpadek, saj je te informacije v prihodnosti treba pridobiti, kar načeloma vodi v manjšo produktivnost.

Koncepti vitkega razvoja predlagajo, da ljudi, ki sodelujejo na projektu, postavimo v isti prostor oz. pomembno je, da so najbližje en drugemu, kolikor je to mogoče (komunikacija iz oči v oči).

Zavedanje, da je napake tekom razvoja treba v najkrajšem možnem času odpraviti, ne glede na velikost le-teh, je del osnovnih principov filozofije vitke proizvodnje.

2.3.2 Pospešeno učenje

Ob predpostavki, da želimo programsko rešitev izdelati pravilno v eni sami iteraciji, to pogosto vodi do slabih končnih rezultatov. Filozofija konceptov vitkega razvoja poudarja nenehno učenje, ki izhaja iz nenehnega preizkušanja in iskanja rešitev in preurejanja (angl. refactoring) kode in zasnove programske rešitve. V tradicionalno plansko vodenih metodah je spreminjanje začetnih načrtov nezaželeno, torej povratne informacije nimajo prave vrednosti. S tem omejimo tudi učenje, kar bi nas načeloma lahko pripeljalo do boljših programskih rešitev.

Rešitev za odpravo omejitev, ki jih s predvidevanjem definiramo ob začetnih fazah razvoja pri plansko vodenih metodah, najdemo v samih agilnih metodologijah z iteracijami. Z iteracijami ne pridobimo le večjega pretoka povratnih informacij vseh udeležencev razvoja, temveč tudi večjo fleksibilnost za spremembe, ki jih vodi sprotno učenje. Kratki razvojni cikli so del filozofije vitkega razvoja in so nujni za uspešen končni produkt. Kot že omenjeno, razvoj programske opreme po konceptih vitkosti ni razvoj brez načrtovanja. S pričetkom vsake iteracije je potreben načrt, katere funkcije bodo implementirane in dostavljene uporabnikom oz. naročnikom. Funkcije, ki imajo največjo poslovno vrednost v programski rešitvi, imajo prednost v razvoju.

2.3.3 Čim bolj pozne odločitve

Koncepti vitkosti poudarjajo pomembnost, da odločitve zajemajo čim širše področje. Ožje kot je omejitveno območje določeno, večja verjetnost je, da bo pri nadaljnjih odločitvah prišlo do popravkov in posledično odpadka. To je seveda v nasprotju s filozofijo plansko vodenih metod razvoja, kjer je treba večino odločitev sprejeti v začetnih fazah. Če idejo omejitev uporabimo v domeni programske opreme, pridemo do principa najkasnejše možne odločitve. Ta princip je tesno povezan s hitrejšim učenjem, saj preko novih spoznanj in ugotovitev pridemo do boljših končnih odločitev, ker imamo pri tem izbiro in več znanja. Če so omejitve zelo ozko določene, imata učenje in znanje posledično manjši vpliv na sam razvoj. Pogosto je najenostavnejše omejiti pogoje na natančne meritve v začetnih fazah razvoja, vendar študije dokazujejo, da to ni nujno najučinkovitejša pot, saj je fleksibilnost pomemben faktor pri hitro

spreminjajočih se razvojnih produktih, med katere zagotovo spada tudi razvoj programske opreme [*26*].

Pri razvoju programske opreme ne gre le za implementacijo rešitev, ampak tudi za načrtovanje programske rešitve. Načrtovanje in sama arhitektura se po principih vitkosti razvijata skozi čas preko iteracij, vendar je treba od samega začetka upoštevati njuno fleksibilnost. Torej, cilj ni načrtovanje popolne arhitekture vnaprej v začetnih fazah razvoja, ampak ustvarjanje arhitekture, ki bo robustna na spremembe skozi celotno življenjsko obdobje programske opreme, vključno z vzdrževanjem. Prav tako bo razvojna ekipa skozi čas lahko ugotovila, kateri deli programske opreme se spreminjajo največkrat, kar označuje del kode, ki potrebuje največjo fleksibilnost. S tem bomo na dolgi rok dodatno omejili stroške razvoja pri spremembah, ki sledijo.

Odpornost na spremembe lahko v praksi dosežemo na več načinov. Mary Poppendieck in Tom Poppendieck predlagata (1) uporabo modulov, (2) uporabo vmesnikov, (3) uporabo parametrov, (4) uporabo abstraktnih elementov, (5) uporabo deklarativnega programiranja namesto proceduralnega, (6) previdnost pri programskih ogrodjih (angl. programme freamework), (7) izogibanje ponavljanju kode, (8) funkcionalno odgovornost modula, (9) enkapsulacijo variacij, (10) odložitev implementacije trenutno nepotrebnih funkcionalnosti programske opreme in (11) izogibanje dodatnim funkcionalnostim [*13*] [*27*].

2.3.4 Čim hitrejša dostava

Princip najhitrejše možne dostave je tesno povezan s principom najkasnejše možne odločitve. Hitrejša kot je dostava, dlje časa lahko odlašamo pri pomembnih odločitvah, saj imamo na voljo več informacij s strani uporabnikov in drugih udeležencev v razvoju programske opreme.

Izvlečni sistem (angl. pull system) temelji na ideji, da se razvoj odziva le na trenutne zahteve, za razliko od potisnega sistema (angl. push system), kjer zaposleni razvijajo sistem po vnaprej določenih zahtevah oz. predvidevanjih. Izvlečni sistem je trden člen celotne ideje vitke proizvodnje, saj se poskuša znebiti odpadkov tako, da iz razvoja izloči predvidevanja. Ta predvidevanja se v tipičnih proizvodnih linijah kažejo v količini zaloge, v samem razvoju programske opreme po nepotrebnih dodatnih funkcionalnostih. Treba je poudariti, da določenega faktorja predvidevanja ni mogoče povsem izključiti.

Izvlečni sistem odlično deluje s sistemom za urejanje urnikov kanban. Besedna zveza kanban je sestavljena iz besede kan, ki označuje vizualno, in ban, ki označuje kartico [*28*]. Sistem deluje tako, da delovne naloge zapišemo na kartice in jih postavljamo v različne razvojne faze

na tabli. Eden od glavnih namenom sistema kanban je celotna vizualna preglednost nad razvojem na enem mestu (končane naloge, naloge v čakanju, razvijalci na nalogi). Sistem kanban za razvoj programske opreme bomo podrobneje predstavili v nadaljevanju.

2.3.5 Povečanje pooblastil ekipe

Ideja principa okrepitve pooblastil temelji na predpostavki, da so največja motivacija za razvijalce povratne informacije in možnost eksperimentiranja pri razvoju. Možnost sprejemanja odločitev na najnižjem nivoju podjetja ali organizacije daje razvijalcem več svobode, zadovoljstva, priznavanja, na dolgi rok pa predvsem več izkušenj. Spodbujanje zaposlenih, da pri svojem delu uporabijo vse svoje zmožnosti in sposobnosti, je tisto, kar vodi v napredek, saj velikokrat napake motivirajo zaposlene oz. razvijalce, da razvojne procese izboljšajo. Ta motivacija je seveda odvisna od tega, kakšen odnos ima vodstvo razvojne ekipe do napak, ki se zgodijo tekom razvoja.

Pomembno je, da rezultate predstavimo v imenu ekipe in ne v imenu posameznikov. Izpostavljanje posameznikov pogosto vodi v demotiviranje ekipe, saj člani pri tem izgubijo občutek pripadnosti. Pogosta zmota pri motiviranju razvijalcev je pričakovanje, da ekipa tekom razvoja ne bo ustvarila nobenih napak. To je seveda v nasprotju s principom povečanja pooblastil ekipe in navsezadnje s principom pospešenega učenja, pri čemer je smiselno, da določeno število napak tudi pričakujemo. Člani razvojne ekipe morajo imeti občutek, da so kompetentni pri svojih opravilih in naloga vodstva je, da jim s spodbujanjem in pomočjo pokaže, da zmorejo razrešiti probleme, ki so jim bili zaupani. Ne smemo pozabiti, da izredno motivacijo pri delu ustvarjajo občutek napredka in doseženi (vmesni ali končni) cilji.

Pomemben član razvojne ekipe je glavni razvijalec (angl. master developer), ki ga za razliko od ostalih metodologij ni nujno treba določiti ob začetku projekta, vendar je zaželeno, da ima poglobljeno znanje na tehnološkem področju in področju domene, da zna motivirati ekipo in oceniti smer razvoja programske opreme, ima dobro predstavo razvoja na abstraktnem nivoju in obvladuje komuniciranje z vsemi udeleženimi v razvojnem procesu. Njihova najpomembnejša naloga je, da prepoznajo točko, kjer programska oprema potrebuje preoblikovanje in načrtovanje nove arhitekture ter ima odločilno vlogo pri njeni vzpostavitvi.

2.3.6 Vgraditev integritete

Pri razvoju programske opreme sta pomembna dva (2) koncepta integritete:

1) zaznana integriteta (angl. perceived integrity),

2) konceptualna integriteta (angl. conceptual integrity).

Zaznana integriteta je način pretoka informacij, kjer podatki učinkovito potujejo med naročnikom oz. uporabnikom in razvijalci programske opreme. Problem, ki ga zasledimo pri tradicionalno plansko vodenih metodah, je ta, da se med dokumentom zahtev in implementacijo nahajata še dve fazi razvoja: analiza in načrtovanje. S tem lahko pride do sprememb ali izgub informacij, kar pripelje do drugačnih rezultatov pri implementaciji, saj razvijalci zahteve drugače tolmačijo. Za to obstaja več rešitev, kot so (1) dostopnost do ljudi, ki lahko integriteto programske opreme v trenutku ovrednotijo, (2) testi s strani naročnika, (3) orodja, ki omogočajo preprosto razumevanje zahtev kompleksnih sistemov na strani naročnika (podatkovni model domene, slovar izrazov, primeri uporabe, kvalifikatorji), (4) glavni razvijalci naj predstavljajo interese naročnikov v večjih in zapletenejših sistemih.

Konceptualna integriteta za razliko od opazne zajema ekipo razvijalcev programske opreme in predstavlja podrobne tehnične podatke, ki se navezujejo na sam izdelek. Najpomembnejša enota konceptualne integritete je arhitektura, ki je fleksibilna in prilagodljiva. Pri tradicionalno plansko vodenih metodah razvoja načrtovanje izhaja iz dokumenta zahtev, kjer informacije načeloma potekajo le v eno smer, medtem ko koncepti vitkosti priporočajo, da se informacije med načrtovalci sistema z ostalimi člani razvojne ekipe prenašajo pogosto, hitro in v obe smeri.

Arhitektura sistema se za razliko od ostalih metodologij razvija in spreminja, saj temelji na predpostavki, da se delovanje in razumevanje določenih enot programske opreme izkažeta šele tekom razvoja in ne ob samem začetku. To velja predvsem za kompleksnejše sisteme. Arhitekturo programske opreme je treba vseskozi izboljševati in preurejati (angl. refactoring), s čimer se bo investicija povrnila v večji učinkovitosti pri nadaljnji implementaciji. Če je arhitektura statična in se ne spreminja, je velika verjetnost, da se s povečanjem kompleksnosti sistema zmanjša preglednost in posledično tudi produktivnost razvojne ekipe. Karakteristike konceptualne integritete dosegamo in vzdržujemo predvsem s (1) preprosto in funkcionalno zasnovo sistema, (2) jasno implementacijo (komentarji, imena funkcij, preproste rešitve, enkapsulacija), (3) jasnim namenom arhitekture, (4) izogibanjem ponavljanja kode in (5) izogibanjem nepotrebnih funkcionalnosti.

Pomembna faza v razvoju programske opreme je tudi testiranje. Teste praviloma delimo na (1) testiranje modulov, (2) integracijsko testiranje in (3) sistemsko testiranje. Vsak izmed njih preverja delovanje kode na različnih nivojih programske opreme. Pomemben člen, ki predstavlja razvoj po principih vitkosti, so uporabniški testi (angl. customer tests), vendar jih ne smemo zamenjevati s sprejemnimi testi (angl. acceptance tests). Sistem testirajo na način,

da preverijo, ali delovanje programske opreme ustreza zahtevam naročnika. Prav tako predstavljajo vir informacij razvijalcem in pomoč, kako naj programska oprema deluje.

2.3.7 Pregled nad celoto

Princip pregleda nad celoto pojasnjuje pogled nad interakcijo enot sistema, ki sestavljajo organizacijo. Pogosto se ideja pri optimizaciji sistema začne pri izboljšanju enot, vendar pogosto spregledamo interakcijo med različnimi deli sistema. Ker meritve učinkovitosti opravljamo pri vsaki enoti posebej, je sodelovanje med razvojno ekipo pogosto sekundarnega pomena, ker si želi vsak posameznik primarno izboljšati rezultat pri meritvah. Rešitev je dokaj enostavna, saj lahko uspeh ali neuspeh razvojne ekipe ocenjujemo le na podlagi končnega izdelka in ne vsake enote posebej. Ekipo je treba organizirati na podlagi izdelka, ki ga razvija, in ne na podlagi posameznih nalog, ki jih opravlja vsak član ekipe. Organizacija ekip na podlagi izdelka prinaša večjo pripadnost, odgovornost, kvaliteto in inovativnost, več sodelovanja med člani in prijetnejši ekipni duh [*29*].

2.4 Vzporednice in razlike v primerjavi agilnih in vitkih principov

Čeprav se v mnogih primerih zdi, da principe vitkosti razvoja programske opreme štejemo med agilne metode razvoja, med njima pravzaprav ni jasne ločnice. V metodah in principih sta si marsikje sorodni, vendar je njun izvor različen. Kot smo že omenili, jasen začetek razvoja agilnih metod štejemo z izidom dokumenta Agilni manifest (angl. Agile Manifesto [*17*]) kot odgovor okornim plansko vodenim metodam, medtem ko razvoj po konceptih vitkosti izvira iz idej, ki so bile iz industrijskega okolja PST prenesene v domeno programske opreme. Označevanje vitkega razvoja programske opreme kot agilne metodologije ni smiselno, saj so orodja, ki sta jih opisala Mary Poppendieck in Tom Poppendieck, veliko bolj abstraktna v primerjavi z metodologijami kot sta scrum ali ekstremno programiranje. Bolj smiselna je namreč primerjava principov, na kar bomo odgovorili v tem delu diplomskega dela.

Agilne metode in koncepti vitkosti postavljajo v ospredje uporabnika in ne samega procesa razvoja, vendar je način obravnave uporabnika nekoliko drugačen. Agilne metode kot največjo prioriteto postavljajo zadovoljstvo uporabnika [*17*], medtem ko pri principih vitkosti ustvarjamo le tisto, kar ustvarja dodatno vrednost programske opreme. Ta razlika je morda sprva nekoliko nejasna, vendar je treba poudariti, da razvoj po konceptih vitkosti predvideva, da ni nujno, da uporabnik razume, kaj v resnici želi.

Principe vitkosti v razvoju programske opreme, ki smo jih v prejšnjem poglavju že spoznali, bomo označili:

- V1 – odpravljanje odpadkov (angl. eliminate waste),
- V2 – pospešeno učenje (angl. amplify learning),
- V3 – čim bolj pozne odločitve (angl. decide as late as possible),
- V4 – čim hitrejša dostava (angl. deliver as fast as possible),
- V5 – povečanje pooblastil ekipe (angl. empower your team),
- V6 – vgraditev integritete (angl. build integrity in),
- V7 – pregled nad celoto (angl. see the whole).

Principi agilnega razvoja programske opreme:

- A1 – najvišja prioriteta je zadovoljiti stranko,
- A2 – sprejemanje spremembe zahtev,
- A3 – pogosto izdajanje programske opreme,
- A4 – nenehno sodelovanje med ljudmi, ki delujejo v poslovnih procesih, in razvijalci,
- A5 – projekti se gradijo okrog motiviranih posameznikov,
- A6 – prioritetni način komunikacije je pogovor iz oči v oči,
- A7 – delujoča programska oprema je primarno merilo napredka,
- A8 – promoviranje trajnostnega razvoja,
- A9 – nenehna težnja k tehnični odličnosti in dobremu načrtovanju,
- A10 – preprostost,
- A11 – samoorganizacija ekip,
- A12 – v rednih časovnih obdobjih ekipa išče načine, kako postati učinkovitejša ob rednem prilagajanju svojega delovanja.

Podobnosti in razlike V1 z ostalimi agilnimi principi

Oba razvojna modela sta si podobna v tem, da želita omejiti nepotrebne funkcije (princip A10), saj se zavedata, da predvidevanja pogosto vodijo v neuspeh. Naslednji agilni princip, s katerim lahko omejimo odpadke, je A4, saj izboljšuje razumevanje pri implementaciji in načrtovanju poslovnih procesov na strani razvijalcev, ter pogosto izdajanje programske opreme (A3).

Kot smo že omenili, se agilni razvoj osredotoča na uspešne rešitve naročniških oz. uporabniških zahtev, medtem ko principi vitkosti ciljajo na dodajanje vrednosti izdelku brez nepotrebnih odpadkov. Podobne rezultate torej paradigmi dosegata na nekoliko različne načine.

Podobnosti in razlike V2 z ostalimi agilnimi principi

Hitrejše učenje (V2) pogostokrat povezujemo s hitrejšimi iteracijami (A3). Ker je pri agilnih metodah pomembno, da se iteracije končajo z delujočo programsko opremo, gre izdelek skozi vse faze razvoja. Iz tega sledi, da v primerjavi s tradicionalno plansko vodenimi metodami hitreje pridemo do napak in problemov, ki se pojavijo tekom razvoja. Velik del filozofije vitke proizvodnje zajema idejo o nenehnem izboljševanju, podobnost teh idej pa lahko v agilnem razvoju najdemo pri principih A8, A9 in A12. Oba načina pričenjata razvoj pri funkcionalnostih, ki predstavljajo največjo poslovno vrednost.

Podobnosti in razlike V3 z ostalimi agilnimi principi

Agilen razvoj nikjer ne zajema principov, ki bi zajemali prenašanje odločitev v prihodnost. Mnogokrat je razumevanje vitkega principa odlašanja odločitev napačno, zato se mnogi uporabniki agilnih metodologij z njim ne strinjajo [*30*]. Treba je poudariti, da gre za odgovorno odlašanje odločitev mnogokrat le v obliki omejitev določenih enot razvoja programske opreme in ne za namerno odlašanje aktivnosti, ki jih je prioritetno treba opraviti. Agilne metode predlagajo izogibanje analizam in pogovorom o podrobnostih vnaprej, saj preusmerijo pozornost od trenutnih razvojnih nalog [*30*]. Principi obojih delujejo na principih samoorganizacije (A11).

Podobnosti in razlike V4 z ostalimi agilnimi principi

Najhitrejša možna dobava je vsekakor lastnost, ki jo najdemo tudi pri agilnih principih razvoja (A3). Povezave lahko najdemo tudi v principu preprostosti (A10), saj se razvoj osredotoča na trenutno najbolj pomembne funkcionalnosti, s čimer dosežemo pogosta izdajanja.

Podobnosti in razlike V5 z ostalimi agilnimi principi

Principi vitkosti predlagajo večja pooblastila članom ekipe s strani vodstva, kar lahko odločno vpliva na motivacijo razvijalcev (A5). Poleg tega oboji principi zagovarjajo samooblikovanje ekipe z motiviranimi posamezniki, principi vitkega razvoja velik poudarek namenjajo jasnemu namenu in ciljem vseh razvijalcev.

Eno izmed razlik lahko opazimo v vodstvu razvojne ekipe. Če imamo pri agilnih metodologijah samoorganizacijo ekip pred začetkom projekta, principi vitkosti dovoljujejo, da se glavni razvijalec (angl. master developer) določi tekom samega razvoja, s čimer ekipa izbere tistega, ki najbolje obvlada tako tehnične elemente razvoja kot tudi domeno programske opreme, ki jo ekipa razvija [*13*].

Podobnosti in razlike V6 z ostalimi agilnimi principi

Nenehne izboljšave so velik del filozofije, ki jih zagovarjajo vitki principi, najdemo pa jih tudi pri agilnih metodah razvoja, predvsem pri A9 in A12. Podobnost lahko opazimo v načinu komunikacije, saj obe strani namenjata prednost pogovoru iz oči v oči (A6).

Razliko lahko opazimo v poudarku konceptualne integritete (angl. conceptual integrity), ki je bolj opazna pri konceptih vitkega razvoja, medtem ko agilni večji poudarek namenja opazni (angl. perceived integrity).

Podobnosti in razlike V7 z ostalimi agilnimi principi

Princip vitkosti, ki govori o pomembnosti interakcije med enotami sistema (V7), ni zajet v agilnih metodologijah, kar lahko prikazuje nekatere pomanjkljivosti agilnih principov.

Poglavje 3 Vpeljava konceptov

3.1 Kanban

Metoda kanban se je v avtomobilski proizvodni industriji Toyota razvila za podporo sistemov JIT, kjer se je uporabljala za nadzor zaloge [*31*]. To dosega s posebnimi karticami, ki označujejo izdelek (kos, del) v razvoju, s katerimi signalizira potrebo po novi zalogi. Specifičnost sistema je v tem, da ima omejeno število kartic. Če pride do potrebe po novi proizvodnji aktivnosti in nobene kartice ni na voljo, je treba kartico uvrstiti v vrsto, dokler se mesto za pričetek nove aktivnosti ne sprosti [*32*].

Mehanizem, ki stoji za metodo kanban, se imenuje izvlečni sistem (angl. pull system). Za razliko od potisnega sistema (angl. push system) si pri izvlečnem razvijalci oz. delavci načeloma sami izbirajo naloge, pri potisnem pa jih praviloma delegira vodstvo.

3.2 Kanban v procesu razvoja programske opreme

V razvoju programske opreme kartice kanban uporabljamo kot enoto dela ali aktivnosti in ne kot signal [*32*]. Največja pridobitev uporabe sistema kanban je v omejevanju količine dela v teku (angl. work in progress, v nadaljevanju WIP) [*31*]. Tekom razvoja agilnih metodologij so se pojavile podobne metode za vizualizacijo delovnega toka, vendar je treba poudariti, da brez omejevanja WIP in izvlečnega sistema ne predstavljajo elementov metode kanban.

David J. Anderson je v knjigi [*32*] definiral 5 lastnosti metode kanban, in sicer (1) vizualizacija delovnega toka, (2) omejevanje dela v teku, (3) merjenje in upravljanje delovnega toka, (4) eksplicitna pravila procesov in (5) uporaba modelov za prepoznavanje priložnosti za izboljšave.

Pri razvoju se pojavljajo tudi različice sistema kanban v kombinaciji z agilno metodologijo scrum, nekateri jo označujejo kar *scrumban*.

3.3 Implementacija metode kanban

Za implementacijo metode kanban v fizični obliki potrebujemo tablo kanban (angl. kanban board) in kartice kanban (angl. kanban tickets) za označevanje aktivnosti v procesu razvoja. Tabla kanban je v osnovi bela deska oz. tabla, na kateri razdelimo prostor (po vrsticah ali tipično po stolpcih) za različne procese, ki so specifični glede na potek razvoja programske opreme v sami organizaciji. Pomembna prednost je prilagodljivost razvojnega procesa, saj metoda omogoča, da organizacija oblikuje procese po lastni presoji. Omejevanje WIP dosežemo tako, da omejimo maksimalno število kartic v procesu [*31*]. Na Sliki 3.1 vrednost *n* označuje zaporedno število procesa, vrednost x_n pa predstavlja največje dovoljeno število kartic v posameznem stolpcu (omejitev WIP). Tabla kanban izpolnjuje prvi princip vizualizacije delovnega toka, vrednost x_n pa princip o omejevanju WIP. Aktivnosti po navadi razdelimo v dve (2) skupini: tiste, ki so trenutno v teku, in končane.

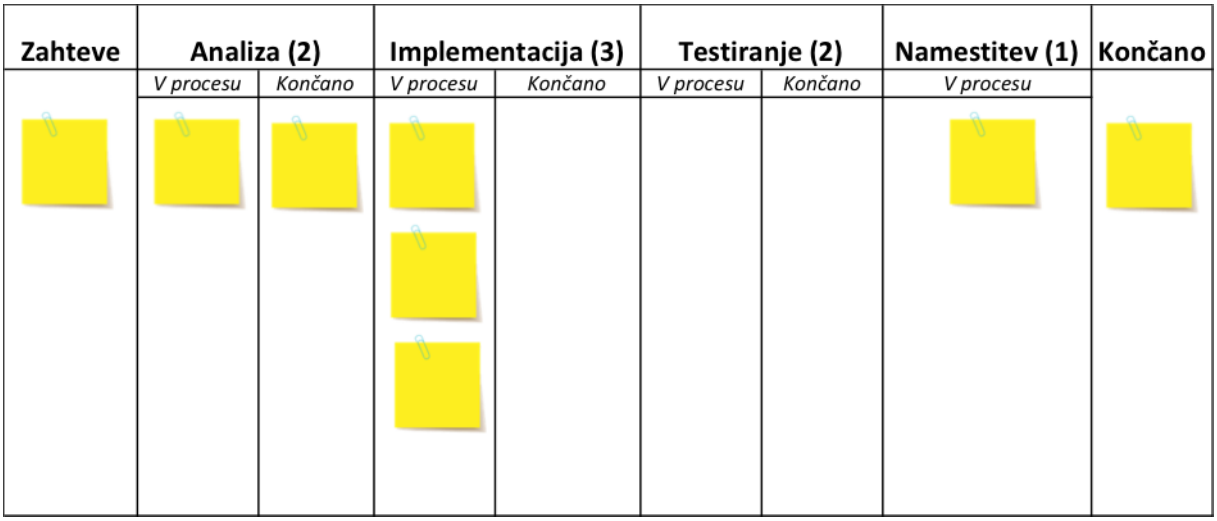
Elementi dela	**Korak 1 (x_1)**		**Korak 2 (x_2)**		**…**	**Korak n (x_n)**		**Končano**
	V procesu	*Končano*	*V procesu*	*Končano*		*V procesu*	*Končano*	
					…			

Slika 3.1: Prikaz izhodiščne table kanban.

Kartice kanban praviloma predstavljajo uporabniške zahteve, načeloma so to lahko tudi funkcionalnosti ali uporabniške zgodbe (angl. user stories). Uporaba kartic je od primera do primera lahko zelo različna, po navadi pa nanje zapišemo (1) kratek opis, (2) pomembne časovne mejnike, če so ti potrebni, (3) prioriteto v vrsti, (4) podatke o sistemu ali programski opremi in (5) podatke o razvijalcih, ki so aktivni na kartici. Pri razvoju lahko z uporabo različnih barv kartic dosegamo večjo preglednost. Kot primer lahko barve kartic razdelimo na tiste, ki ponazarjajo (1) časovno-omejene zahteve, (2) normalne zahteve, (3) napake in (4) tehnični dolg (angl. technical debt, izraz, s katerim označujemo posledice slabega razvojnega načrtovanja) [*33*]. Kartice izpolnjujejo princip vizualizacije.

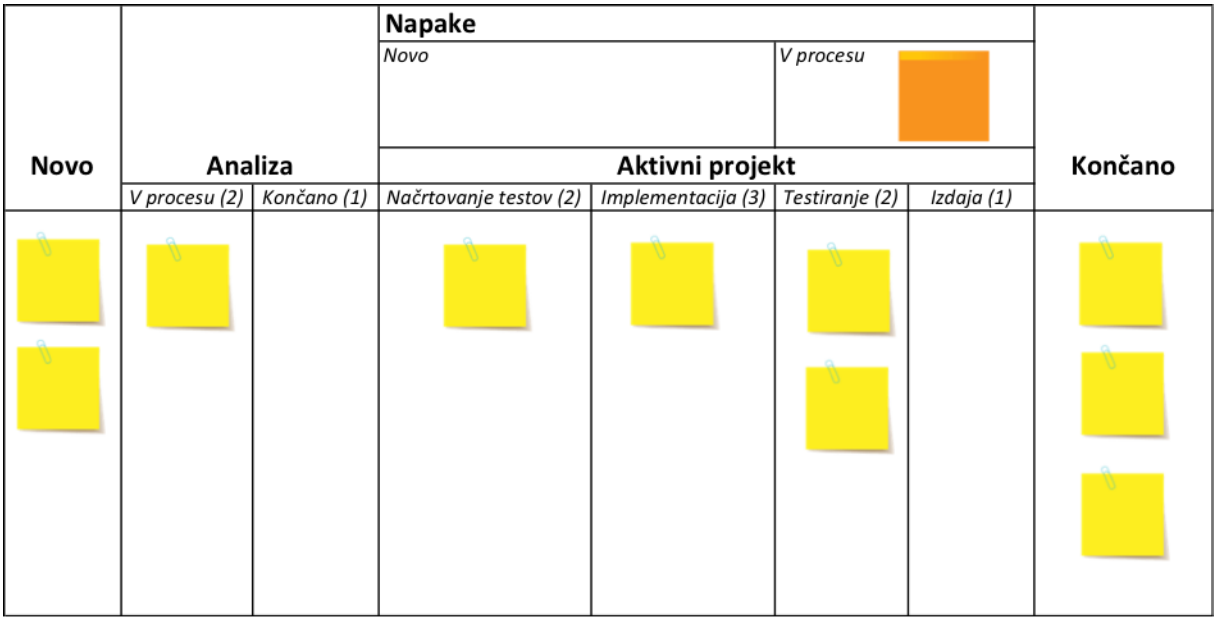
3.3.1 Primeri uporabe table kanban

Primer, ki ga vidimo na Sliki 3.2, je osnovna razdelitev, ki jo uporabljajo razvojne ekipe, kjer so člani nekoliko bolj specializirani za določeno delo. Prednost predstavlja stolpec, ki označuje končano delo [*33*]. To je delo, ki signalizira ostalim razvijalcem, da je delo pripravljeno na nadaljnjo obdelavo.

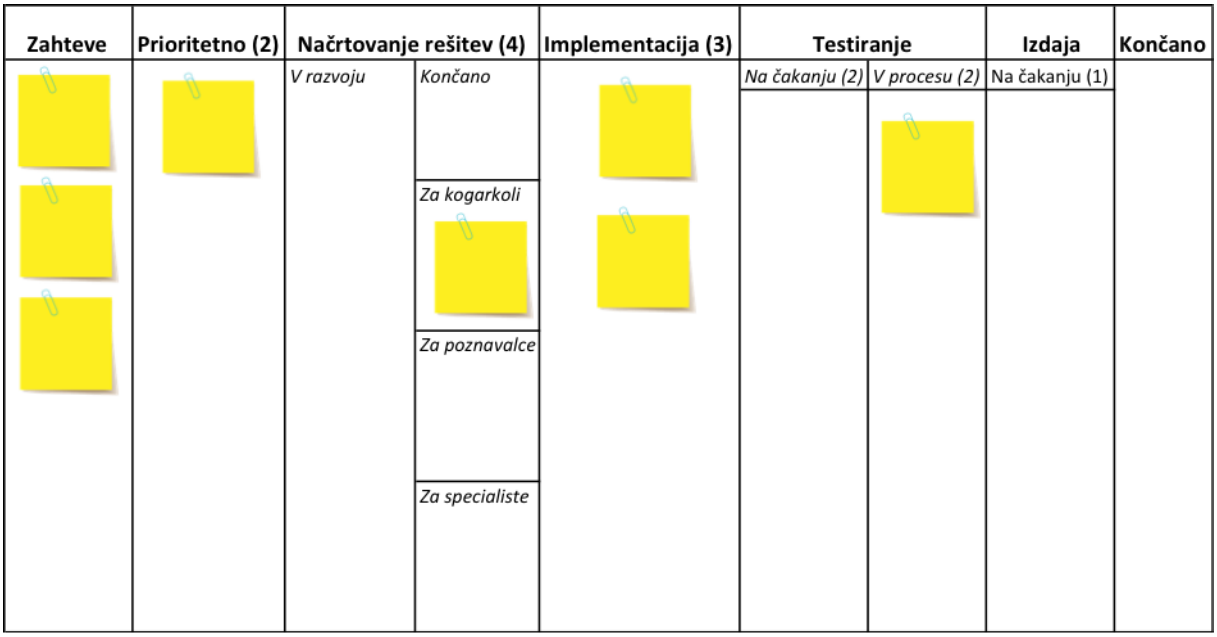


Slika 3.2: Najpogostejša razdelitev procesov pri metodi kanban [*33*].

V naslednjem primeru (Slika 3.3) nekoliko večji poudarek namenjamo napakam, ki se dogajajo tekom razvoja, za razliko od prejšnjih pa omejitev WIP postavljamo na druga, bolj specifična mesta. Poleg tega velja omeniti, da lahko pri označevanju napak uporabljamo različne barve kartic ali druge oblike označevanja, brez da temu namenjamo poseben prostor na tabli kanban. Odločitev seveda temelji na primeru razvojne ekipe.



Slika 3.3: Prikaz primera, ki poseben prostor namenja napakam in težavam [*33*].

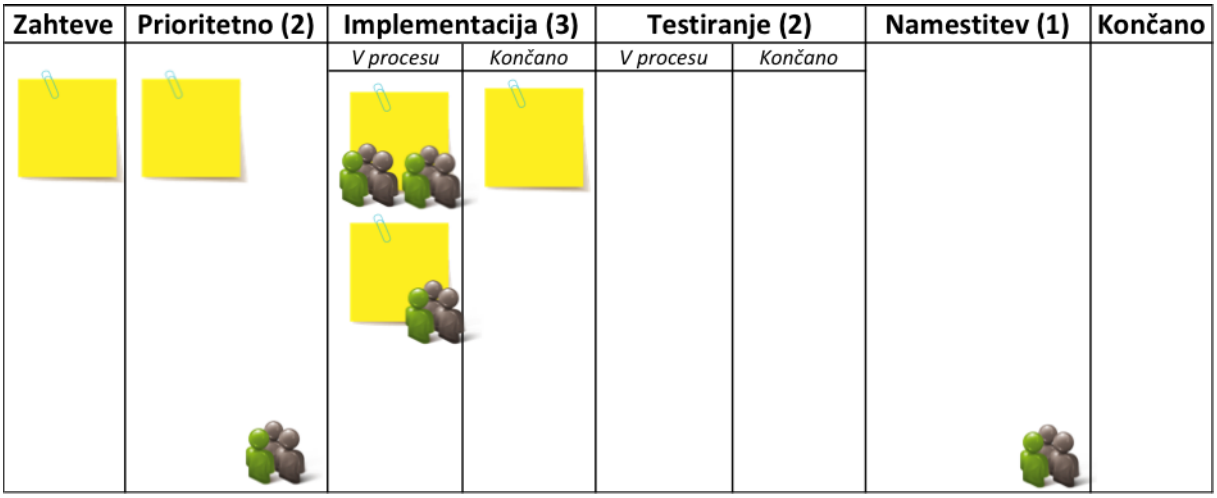


Slika 3.4: Prilagojen primer table kanban [*33*].

Tablo kanban (Slika 3.4) lahko prilagodimo tudi tako, da kartice umestimo v različne težavnostne stopnje, kjer naloge razdelimo med razvijalce z različnimi znanji in izkušnjami. S tem primerom si lahko pomagamo tudi pri drugih razvojnih procesih, npr. pri testiranju. Na levi strani table kanban imamo stolpec prioritetno, kjer dovolimo sodelovanje z naročnikom oz. uporabnikom, s tem da sami izberejo aktivnosti razvoja, pri katerem bo programska

oprema pridobila največjo poslovno vrednost. V primeru na Sliki 3.4 sta to dve (2) zahtevi. S tem pokrijemo koncepte vitkosti V1 in V6. Nadaljnje sodelovanje lahko spodbudimo tudi s sprejemnimi testi v stolpcu testiranja.

Slika 3.5 je namenjena prikazu medsebojnega sodelovanja različnih skupin v primeru, ko je v neki fazi dosežena omejitev WIP. Z medsebojnim sodelovanjem dosežemo hitrejšo izdajo (Princip V4) in posledično pospešimo učenje razvojne ekipe (Princip V2). Razvijalci se popolnoma posvetijo trenutnemu delu, omejitve dela v teku pa jim onemogoča preklapljanje med procesi (Princip V1).



Slika 3.5: Tabla kanban in omejitev dela v teku pri implementaciji.

Določene koncepte vitkosti, kot je npr. povečanje pooblastil ekipe, težje umestimo v metodo kanban, zato se je treba zavedati, da je sistem kanban v procesu razvoje programske opreme le komplementaren del celotne slike. Iz podanih primerov lahko vidimo, da lahko z dokaj enostavno in lahko metodo pokrijemo kar nekaj razvojnih principov po konceptih vitkega razvoja. Najpomembnejši učinek, ki ga pri metodi kanban dosežemo, je konstanten in enakomeren razvoj tekom celotnega projekta.

Omenili smo tiste primere, ki jih v sami literaturi najpogosteje srečamo oz. se najpogosteje omenjajo. Možnosti je praktično neskončno. Prav tako je treba poudariti, da metoda kanban ni namenjena samo razvoju programske opreme po konceptih vitkosti. Komplementarna je praktično z vsemi agilnimi metodologijami razvoja, saj je okvir delovanja zelo širok in predlaga le malo število omejitev, vendar so le-te mnogokrat lahko zelo učinkovite.

3.4 Kumulativni diagram poteka

Kumulativni diagram poteka (angl. cumulative flow diagram, v nadaljevanju CFD) uporabljamo pri meritvah porazdelitve in učinkovitosti dela pri agilnih metodologijah in pri razvoju programske opreme po principih vitkosti [*32*] [*34*]. Prikazuje količino opravljenega dela v določenih razvojnih procesih.

Pri branju podatkov z diagrama je treba ločiti med časom cikla (angl. cycle time) in potrebnim časom (angl. lead time). Razlika je v tem, da čas cikla predstavlja obdobje od pričetka dela do zadnjega procesa pred samo dostavo programske opreme do naročnika, medtem ko potrebni čas predstavlja obdobje med vnosom zahteve in končno dostavo [*35*]. Čas cikla v praksi torej predstavlja čas razvoja funkcije in je pomembnejši v kontekstu razvojne ekipe in organizacije, informacije o potrebnem času pa več povedo samemu naročniku oz. stranki. Oba časa sta izredno pomembna koncepta, saj lahko predstavljata konkurenčno prednost organizacije z izboljšanjem konkurenčnosti, hitrejšega odziva potrebam na trgu ter inovativnosti [*36*].

Kot smo že omenili, je potrebna previdnost pri definiciji časa cikla. Do sedaj smo v diplomskem delu uporabljali definicijo, ki pravi, da je čas cikla čas, ki preteče od začetka do konca delovnega procesa. V kontekstu Littlovega zakona (angl. Little's law) pa uporabljamo definicijo, ki pravi, da je čas cikla povprečni čas, ki preteče od izhoda dveh zaporednih elementov v delovnem procesu [*37*].

Littlov zakon pravi: *"Povprečno število elementov v vrsti L je enako povprečni hitrosti, s katero elementi prihajajo v vrsto, pomnoženo s povprečnim časom, ki ga element porabi v vrsti"* [*31*] [*38*].

Torej velja

$$L = \lambda W \tag{3.1}$$

pri čemer L predstavlja povprečno število elementov v vrstnem sistemu, W predstavlja povprečen čas za čakanje v vrsti, λ (lambda) pa predstavlja povprečno hitrost prihajajočih elementov v določeni časovni enoti. Če Littlov zakon prenesemo v metodo kanban, velja

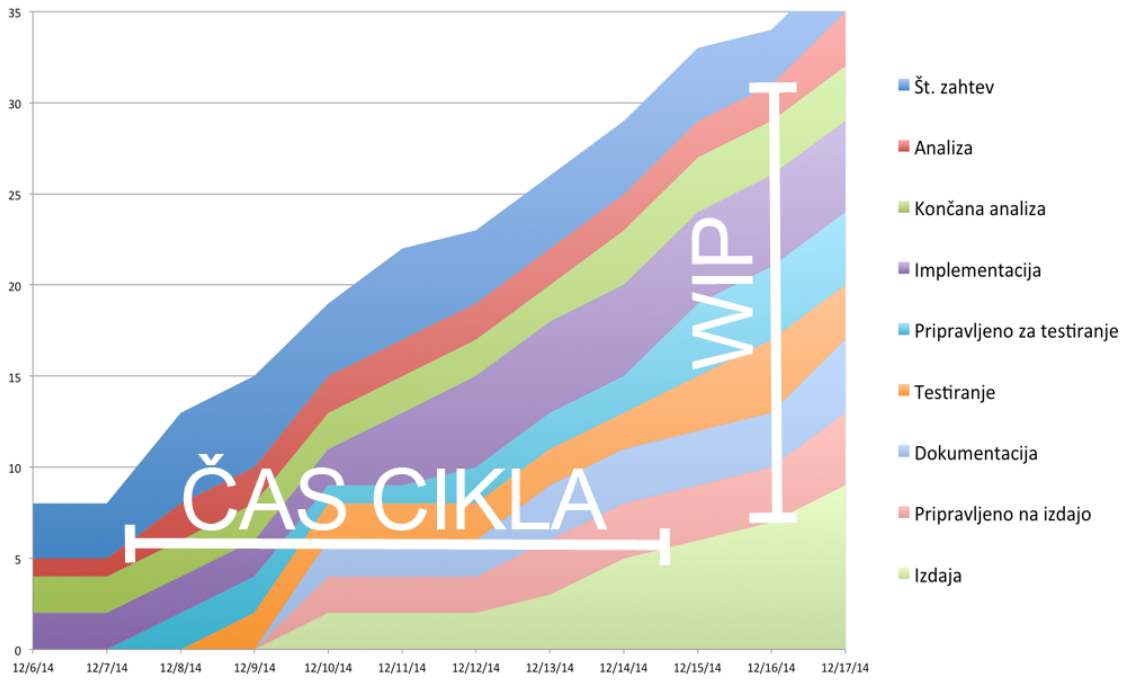
$$\overline{\check{c}as\ cikla} = \frac{\overline{WIP}}{\overline{pretok}} \tag{3.2}$$

pri čemer *WIP* predstavlja omejitev števila kartic, *pretok* hitrost kartic, ki prihajajo v vrsto, in *čas cikla* čas za dokončanje naloge na kartici. Iz tega lahko sklepamo, da s spremembami WIP

vplivamo na čas cikla. Pri razvoju programske opreme velikokrat pride do odstopanj pri oceni stroškov dela, kar je izredno pomembno za razvojno organizacijo. V veliko pomoč nam je pri tem podatek o času cikla, saj lahko z njim tako bolje predvidevamo o prihodnosti projekta [*31*].

Pri analiziranju grafa na Sliki 3.6 os *x* predstavlja čas, iz njenih vzporednic pa lahko razberemo čas cikla in potrebni čas, ki ga potrebujemo za dokončanje aktivnosti v določenem razvojnem procesu. Os *y* predstavlja število nalog [*35*].

Najpomembnejši element, ki ga lahko razberemo iz grafa, je WIP, saj nam ponazori učinkovitost delovanja metode kanban, in sicer tako, da za uspešno implementacijo velja graf, ki je konstanten v vrednosti časa cikla (Littlov zakon) [*32*]. Višino grafa predstavlja število zahtev, ki jih sprejmemo v razvoj, pri čemer število ni konstantno in se lahko tekom časa spreminja.



Slika 3.6: Primer kumulativnega diagrama poteka.

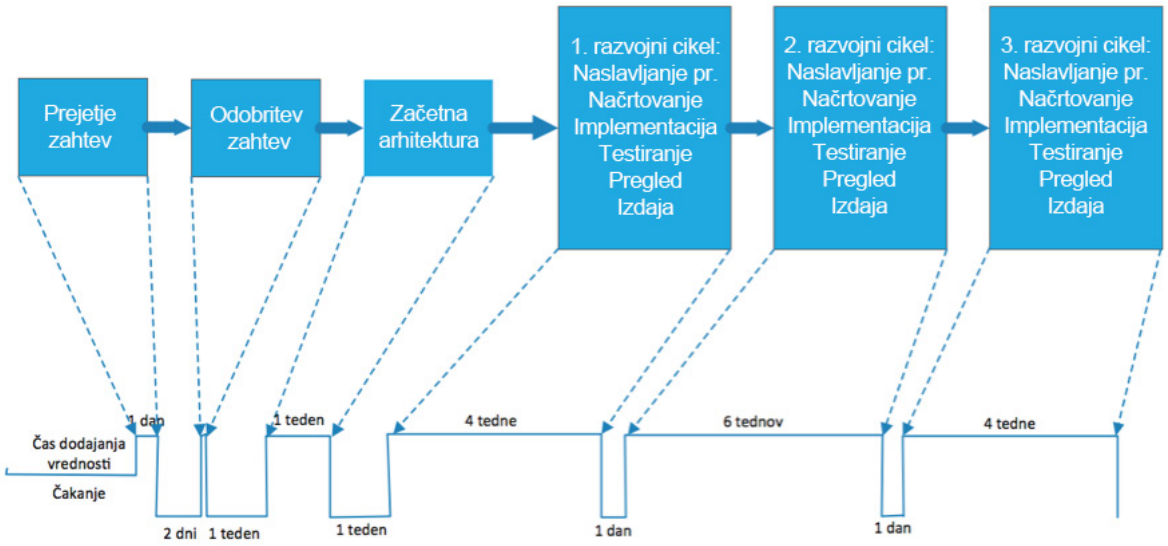
3.5 Diagram toka dodane vrednosti

Načrt toka dodane vrednosti (angl. value stream map, v nadaljevanju VSM) je eden izmed načinov ugotavljanja odpadkov v razvoju izdelka ali produkta. Načrt nam ponazori, koliko časa v razvoju namenjamo aktivnostim, ki produktu dodajajo dejansko vrednost. Produkt z

diagramom spremljamo preko razvoja od začetne zahteve do izdaje. S tem pridobimo koristne podatke, ki nam lahko pomagajo pri prepoznavanju priložnosti za dvig kvalitete in hitrosti dostave programske opreme. Nenehno izboljševanje procesov je eden izmed ključnih konceptov vitkega razvoja.

Na Sliki 3.7, kjer smo prikazali osnovni VSM, vidimo, kolikšen delež razvoja smo namenili aktivnostim, ki prinašajo dodatno vrednost izdelku (visok signal), in koliko tistim, ki ga namenimo neproduktivnim procesom (nizek signal). Nizek signal je torej prostor, kjer lahko izboljšamo razvojni proces.

Pomembno je zavedanje, da zgornji signal predstavlja le tisti delež aktivnosti, ki predstavljajo dodano vrednost naročniku oz. uporabniku. Elementi razvoja, ki prinašajo vrednost podjetju in ne kupcu, veljajo kot odpadek v procesu [*15*]. Cilj VSM je prepoznavanje odpadkov in doseganje višjega razmerja med časom trajanja visokega signala proti trajanju nizkega. Višje kot je razmerje, več odpadkov smo iz procesa uspešno izključili.



Slika 3.7: Prikaz VSM po konceptih vitkosti.

Omejevanje WIP po metodi kanban in izboljšanje razmerja pri VSM sta metodi, ki komplementarno vpeljujeta koncepte vitkosti v razvoj programske opreme.

Poglavje 4 Izdelava elektronskega sistema kanban

4.1 Predstavitev uporabljenih tehnologij za razvoj spletne programske opreme

Za izdelavo spletnega sistema smo uporabili nekatere aktualne spletne tehnologije, in sicer (1) označevalni jezik HTML5, (2) spletni programski jezik PHP skupaj s programskim ogrodjem Laravel, (3) prekrivne sloge CSS skupaj s skriptnim jezikom Sass, (4) skriptni jezik Javascript skupaj s knjižnico jQuery, (5) ogrodje Twitter Bootstrap ter (6) programsko zbirko MAMP skupaj z odprtokodnim sistemom za upravljanje relacijske podatkovne zbirke MySQL. V nadaljevanju bomo na kratko predstavili omenjene tehnologije.

4.1.1 HTML5

HTML5 (angl. Hypertext Markup Language) je zadnja različica označevalnega jezika za oblikovanje večpredstavnostnih dokumentov HTML, ki nam omogoča predstavitev vsebine na internetu. Tehnologija HTML je neodvisna od programske platforme na računalniškem sistemu. Uporabljamo ga lahko torej na katerem koli operacijskem sistemu, dokler nam le-ta omogoča programsko opremo, ki pravilno interpretira dokumente HTML. HTML različice 5 podpirajo vse novejše različice priljubljenih brskalnikov (Internet Explorer, Firefox, Chrome, Safari, Opera, Mobile Safari itd.) [*39*].

4.1.2 CSS in Sass

CSS (angl. Cascading Style Sheets) so prekrivni slogi, ki jih uporabljamo za oblikovanje dokumentov HTML. S tehnologijo CSS lahko izboljšamo postavitev HTML-strani, obarvamo elemente spletne strani, izbiramo med različnimi pisavami, skratka prekrivni slogi ločijo oblikovanje spletne strani in njihovo vsebino [*40*].

V diplomskem delu smo tekom razvoja spletne aplikacije uporabljali skripti jezik Sass[4] (angl. Syntactically Awesome Stylesheets), ki se interpretira v CSS. Pri pisanju prekrivnih slogov v tehnologiji Sass lahko izbiramo med dvema (2) različnima sintaksama (angl. syntax), in sicer prva, starejša različica sintakse zahteva .sass končnico datoteke, medtem ko pri novejši

[4] Več o skriptnem jeziku Sass na spletni strani http://sass-lang.com.

datoteko shranimo s končnico .scss. Največja razlika med njima je ta, da je starejša različica (.sass) občutljiva na prazne znake (angl. whitespace sensitive) in upošteva zamike (angl. indentation-based), medtem ko novejša različica (.scss) teh lastnosti nima. Med njima obstajajo še druge razlike, vendar se bomo pri izdelavi spletne aplikacije v celoti posvetili novejši različici. Skriptni jezik Sass za razliko od CSS omogoča spremenljivke (angl. variables), gnezdenje (angl. nesting), parcialne elemente (angl. partials), uvoz (angl. import), vzorce (angl. mixins), dedovanje (angl. inheritance) in operatorje (angl. operators).

4.1.3 Javascript in knjižnica jQuery

Javascript je spletni programski jezik, ki ga najpogosteje uporabljamo v kombinaciji s tolmačem (angl. interpreter), ki je vgrajen (angl. embedded) v sam spletni brskalnik na strani odjemalca (angl. client-side). Tehnologijo uporabljamo v kombinaciji z DOM (angl. Document Object Model, tj. drevesna struktura HTML-dokumenta), ki ga vzpostavi spletni brskalnik ob vsaki zahtevi, kar nam omogoča manipulacijo s podatki, ki jih na spletni strani prikazujemo na strani odjemalca [*41*].

JQuery je odprtokodno programsko ogrodje, ki dopolnjuje funkcionalnost programskega jezika Javascript. Ogrodje jQuery omogoča (1) enostavno iteracijo preko DOM-elementov z uporabo metod, (2) določanje DOM-elementov, ki je podobno CSS-sintaksi, (3) enostavno dodajanje novih metod in (4) veliko število dodatnih funkcionalnosti, ki omogočajo lažje delo z zavihki, dialogi, animacijami in tranzicijami [*42*].

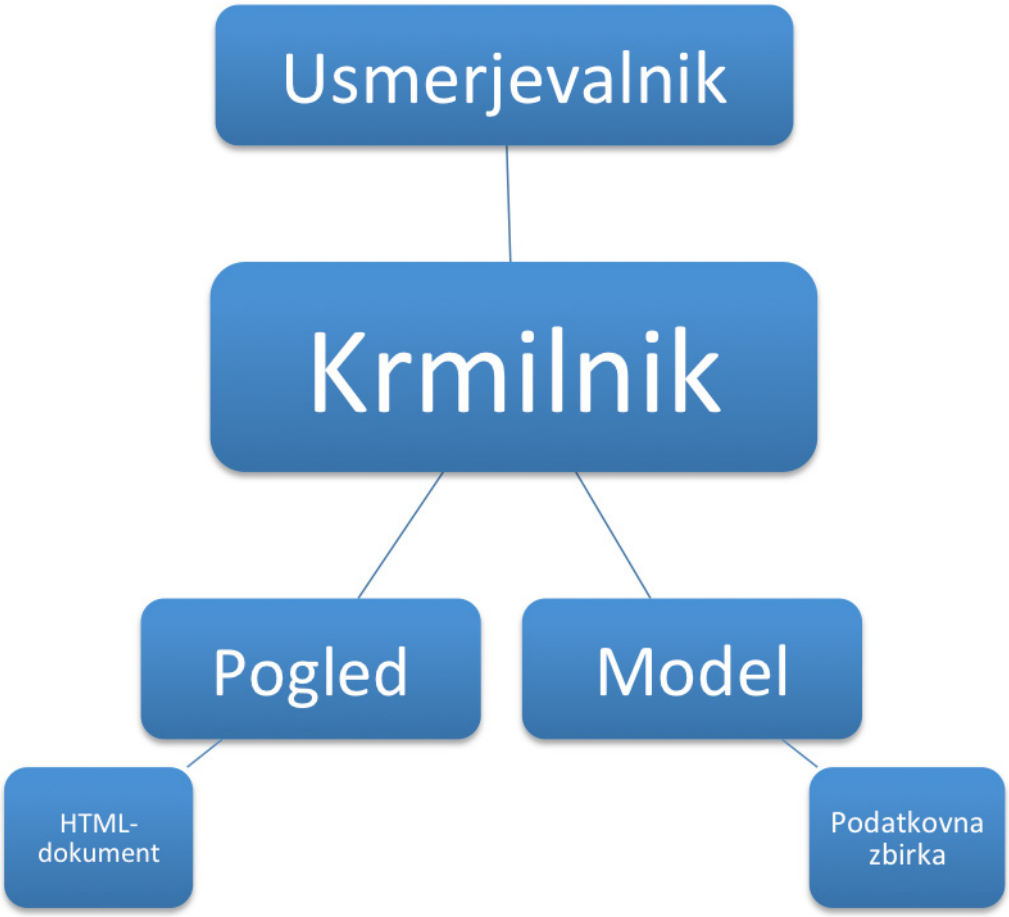
4.1.4 Bootstrap

Bootstrap je brezplačna zbirka HTML, CSS in Javascript elementov, s katerimi na hitrejši in enostavnejši način ustvarjamo spletne aplikacije. Bootstrap pri svojem delovanju uporablja tehnologije HTML, CSS, Javascript in jQuery. Največje prednosti uporabe zbirke Bootstrap so (1) odzivnost spletnih elementov (angl. responsive utilities) na širino brskalnika, (2) mrežni sistem (angl. grid system), (3) navigacija (angl. navbar), tabele (angl. tables), obrazci (angl. forms), tipografija (angl. typography), gumbi (angl. buttons) idr. [*43*].

4.1.5 PHP in Laravel

Programski jezik PHP (angl. Hypertext Preprocessor, v začetku znan kot Personal Home Page) je odprtokodni skriptni jezik, ki se najpogosteje uporablja pri razvoju interaktivnih spletnih aplikacij. Koda programskega jezika PHP se interpretira na strani strežnika, generirane rezultate pa pošlje v obliki HTML-dokumenta nazaj do odjemalca. PHP omogoča objektno usmerjeno programiranje (angl. Object Oriented Programming, v nadaljevanju

OOP), kar je bilo mogoče od različice 3 naprej (v času pisanja tega diplomskega dela je zadnja stabilna različica 5.6.4). V začetku, predvsem v različicah 3 in 4, se je programski jezik PHP soočal s številnimi težavami, ki jih je prinesel objektno usmerjen razvoj, sedaj pa omogoča vse standardne principe OOP (objekti, abstraktni razredi, dedovanje, konstruktorji, destruktorji, izjeme, vmesniki idr.) [*44*].



Slika 4.1: Prikaz arhitekture ogrodja Laravel [*45*].

Laravel je ogrodje (angl. framework) za razvoj spletnih aplikacij, napisanih s programskim jezikom PHP. Razvil ga je Taylor Otwell, prva različica pa je bila izdana pod licenco MIT Open Source leta 2011 [*46*].

Arhitektura ogrodja (Slika 4.1) temelji na načrtovalskem vzorcu (angl. design pattern) Model-Pogled-Krmilnik (angl. Model-View-Controller, v nadaljevanju MVC). Na kratko, model (angl. model) predstavlja domeno aplikacije in opisuje interakcijo med podatki ter poslovna pravila, po katerih se podatki hranijo v podatkovni zbirki. Pogled (angl. view) je namenjen

vizualnemu prikazu podatkov. S tistimi, ki jih prejme od odjemalca, se praviloma ne ukvarja. Krmilnik (angl. controller layer) predstavlja vez med pogledom in modelom. V našem primeru to pomeni obdelavo vhodnih podatkov, interakcijo s podatki in logiko aplikacije. Ob zahtevi odjemalca usmerjevalnik (angl. route) preusmeri zahtevo na določeno funkcijo in krmilnik, odvisno od tega, katera stran je bila zahtevana. Arhitektura MVC je izredno priljubljena med razvijalci spletnih aplikacij, saj omogoča preglednejši in hitrejši razvoj ter predvsem lažje vzdrževanje [*45*] [*47*].

Ostale značilnosti ogrodja Laravel [*46*] [*47*]:

- modularnost (angl. modularity): omogoča enostavno dodajanje in spreminjanje funkcionalnosti z uporabo gonilnikov (angl. drivers) in paketov programske opreme (angl. bundle). Ogrodje Laravel uporablja upravljalnik odvisnosti (angl. dependency manager) Composer, ki skrbi za upravljanje PHP knjižnic;

- migracije (angl. migration): omogočajo nadzor različic (angl. version control) relacijske sheme podatkovne zbirke (angl. database schema), prav tako so migracije neodvisne od sistema za upravljanje s podatkovnimi zbirkami (angl. database management system, v nadaljevanju DBMS). Lahko jih uporabljamo v kombinaciji z MySQL, PostgreSQL, MSSQL ali SQLite;

- graditelj podatkovnih poizvedb (angl. query builder): omogoča implementacijo poizvedb, ki so neodvisne od uporabe DBMS;

- orodje Eloquent ORM: omogoča enostavno uporabo ukazov za delo s podatkovno zbirko. Kratica ORM pomeni objektno-relacijsko povezovanje (angl. object-relational mapping) in deluje na podlagi vzorcev aktivnega zapisa (angl. active record pattern), s katerimi podatkovne sheme tabele predstavljamo kot razrede in instance teh razredov kot zapis ali vrstico v tabeli;

- orodje Artisan: omogoča interakcijo preko vmesnika z ukazno vrstico (angl. command-line interface) med aplikacijo in nekaterimi funkcijami, ki jih ogrodje Laravel omogoča (ustvarjanje in zagon migracij, ustvarjanje krmilnika, zagon strežnika idr.);

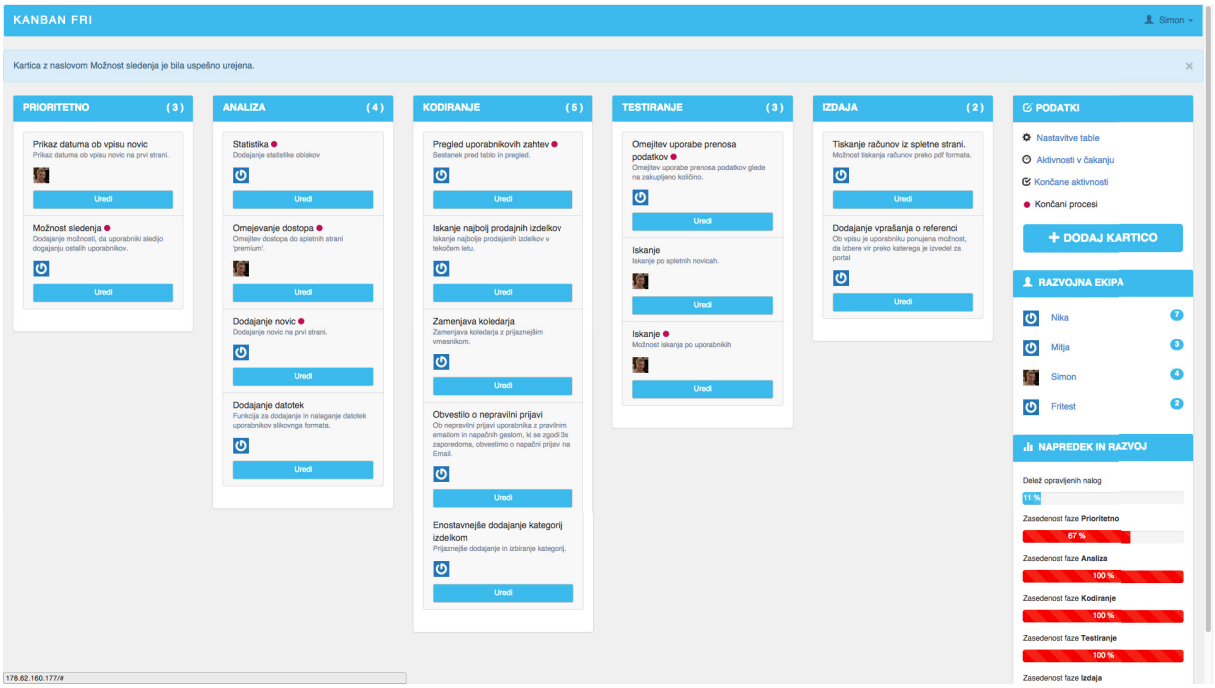
- orodje Blade: je mehanizem vzorcev (angl. templating engine), ki omogočajo preglednejšo kodo pogledov.

4.1.6 MySQL

MySQL (angl. MySQL ali MySequel) je odprtokodni DBMS. Je stabilen, enostaven in odziven sistem, zato je še posebej primeren za razvoj spletnih aplikacij [*48*].

4.2 Predstavitev spletne aplikacije

Tehnologije, ki smo jih našteli in opisali v podpoglavju 4.1, smo uporabili pri izdelavi spletne aplikacije. Spletna aplikacija predstavlja enostaven sistem kanban, ki je namenjen uporabi pri razvoju programske opreme. Njeni najpomembnejši deli so tabla kanban in kartice kanban, ki predstavljajo delovne naloge (Slika 4.2).



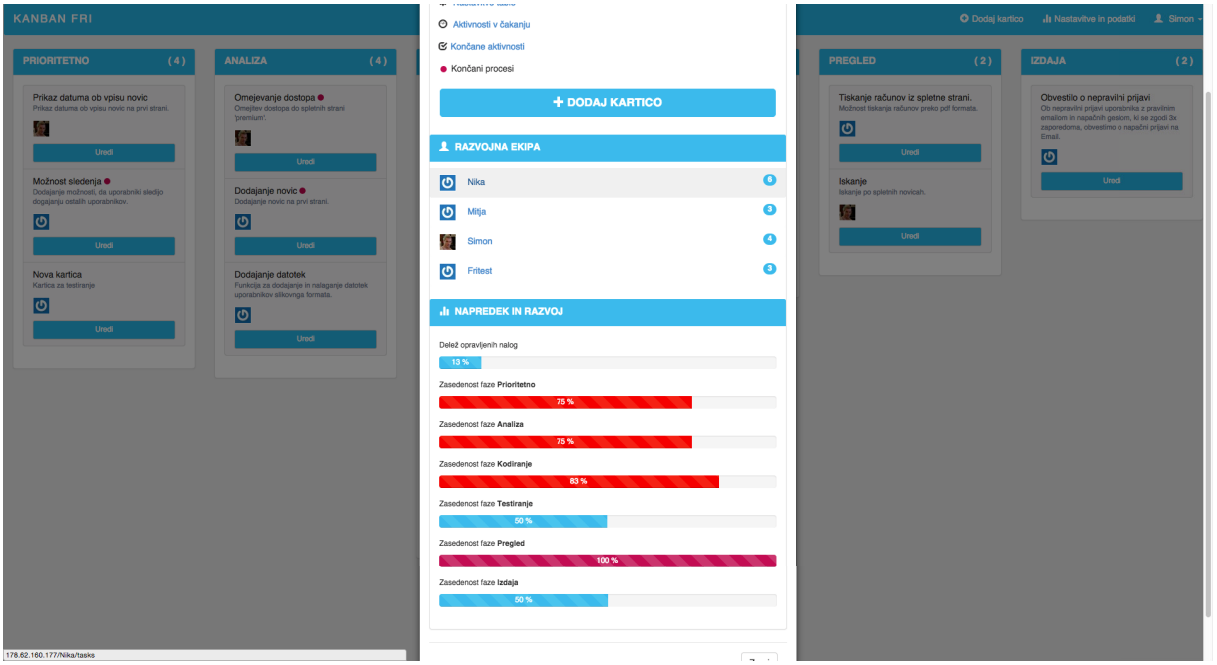
Slika 4.2: Zaslonski prikaz table kanban.

Ob vstopu na spletno stran nam brskalnik ponudi obrazec za vpis. Ko vpišemo pravilne podatke (elektronski naslov in geslo), nas strežnik preusmeri na glavno spletno stran.

Pri sestavi table kanban si lahko izberemo poljubno število stolpcev, vse do zgornje omejitve faz oz. aktivnosti razvoja, ki je šest (6). Kot primer lahko vzamemo prvih pet (5) stolpcev na Sliki 4.2 (označeni kot prioritetno, analiza, kodiranje, testiranje in izdaja), ki predstavljajo različne faze v procesu razvoja programske opreme, šesti (6.) stolpec pa vsebuje pomembne povezave ter enostavno statistiko. Ker lahko aktivnosti ali faze razvoja brišemo ali dodajamo, se spletna stran prilagaja glede na število stolpcev. Tabla kanban omogoča omejevanje WIP,

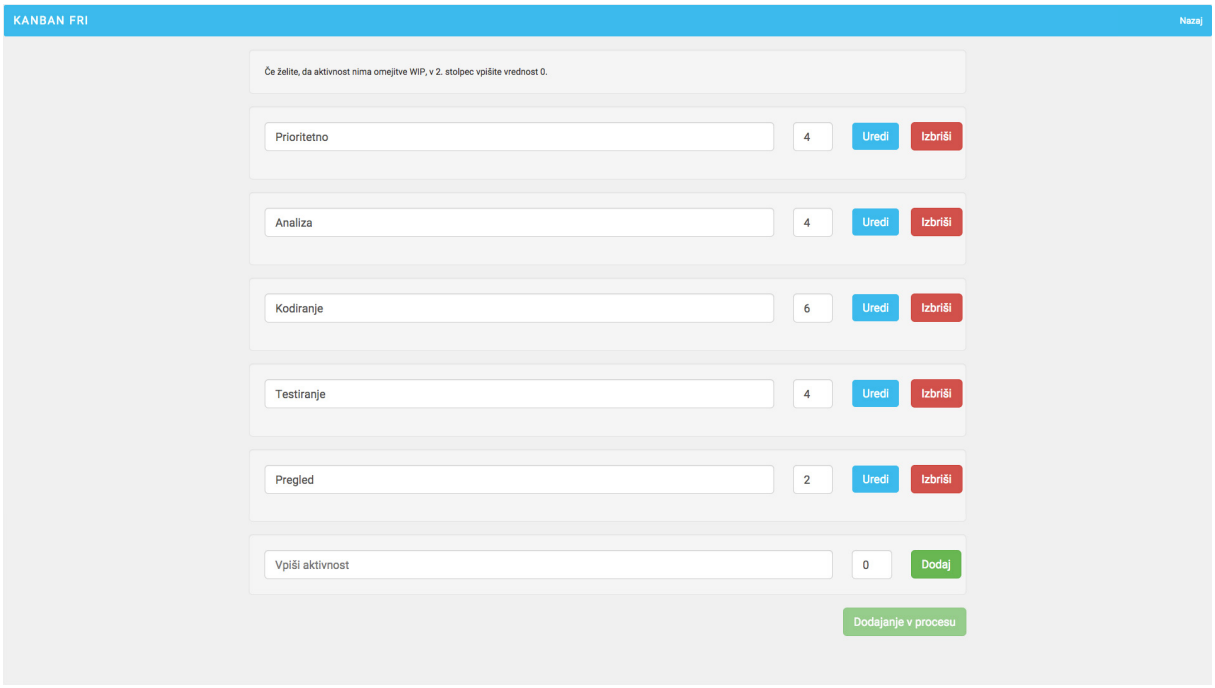
prav tako nam spletna aplikacija ob strani v šestem (6.) stolpcu na Sliki 4.3 prikazuje oz. nas opozarja na razmerje med številom nalog v določeni fazi razvoja in omejitvijo WIP (zadnjih 5 vrstic napredka). Prva vrstica napredka predstavlja razmerje med dokončanimi nalogami in skupnim številom vseh nalog.

Na Sliki 4.3 smo tablo kanban organizirali na šest (6) faz razvoja. Pri tem smo povezavo do statistike in nastavitev prestavili v navigacijsko vrstico (angl. navigation bar). Celoten stranski del (šesti (6.) stolpec na Sliki 4.2) tako odpremo s pomočjo modalnega okna (angl. modal window), saj faze razvoja zasedajo celotno tablo.



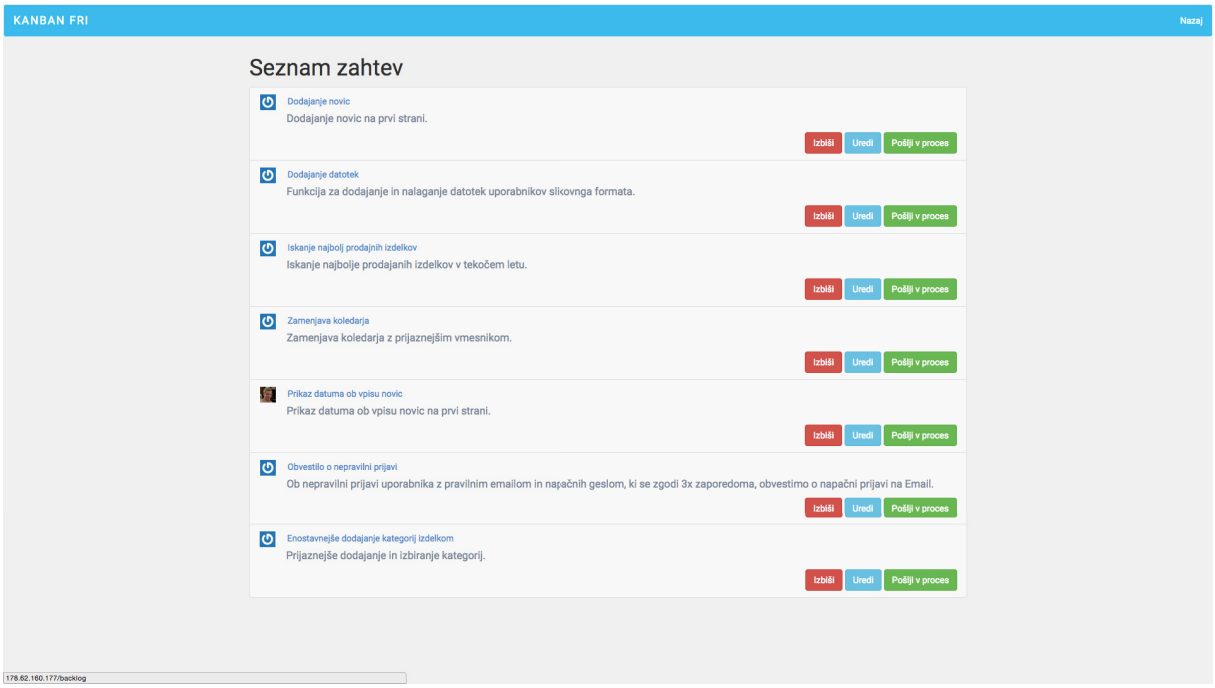
Slika 4.3: Prikaz modalnega okna pri tabli s šestimi (6) aktivnostmi.

Število v oklepaju predstavlja omejitev WIP. Če je število nič (0), potem omejitev WIP ni določena. Če pri dodajanju presežemo število WIP, se prikaže sporočilo o napaki, pri tem pa se proces prenosa kartice oz. delovne naloge na izbrano fazo ne zgodi. Ime faze in omejitev WIP lahko poljubno spreminjamo (Slika 4.4).



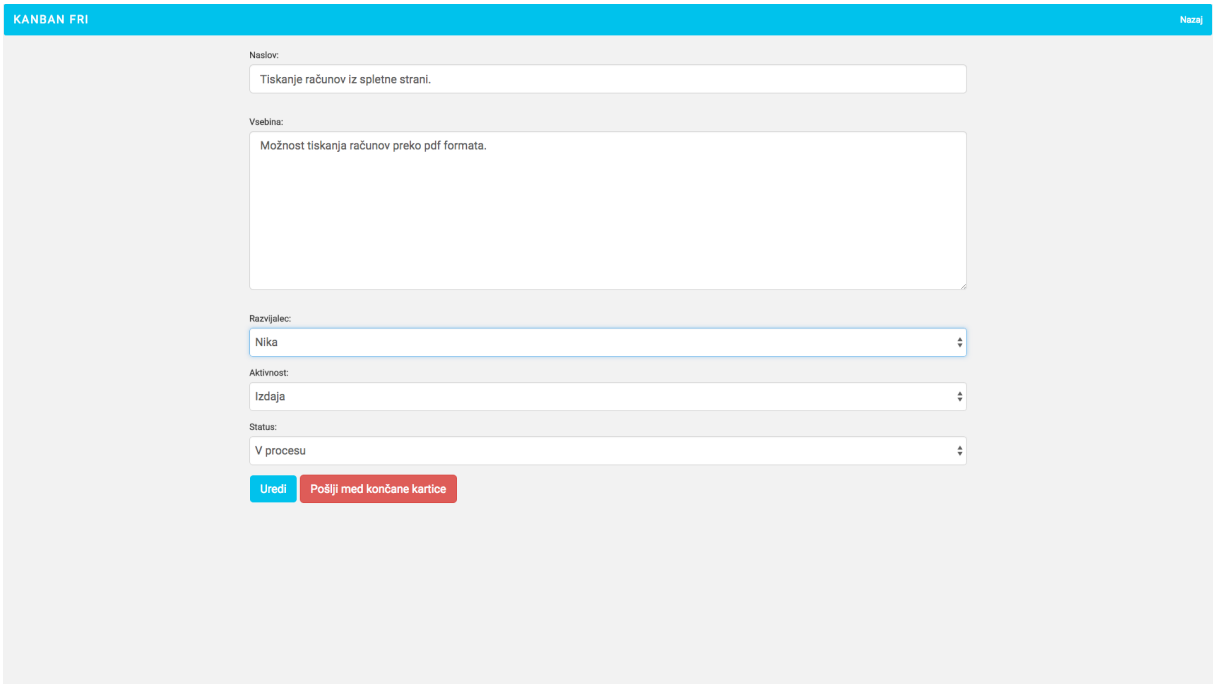
Slika 4.4: Prikaz obrazcev za spremembo imena aktivnosti (razvojne faze) in omejitev WIP.

Na desni strani table v šesti (6.) koloni lahko opazimo povezave za nastavitve table (Slika 4.2), aktivnosti v čakanju in končane aktivnosti. Aktivnosti na čakanju so tiste aktivnosti, ki smo jih dodali s pritiskom na gumb *Dodaj kartico* in še niso v nobeni razvojni fazi. Slika 4.5 prikazuje omenjeni seznam zahtev, kjer so implementirane funkcionalnosti za izbris, urejanje in pošiljanje kartice oz. delovne naloge v prvi (1.) razvojni proces. Kartice delimo na tiste v procesu oz. delu (brez barvne pike) in tiste, ki so dokončane (z barvno piko desno pri naslovu kartice). Pri dodajanju kartic je treba vnesti (1) naslov kartice, (2) besedilo kartice in (3) razvijalca. Na desni strani je prav tako seznam razvijalcev in povezava za prikaz nalog, ki jih trenutno opravljajo. Vsak klik na naslov kartice v aplikaciji odpre stran z vsemi informacijami o delovni nalogi.



Slika 4.5: Prikaz zaslonske slike seznama zahtev na čakanju.

Kartice urejamo tako, da kliknemo na povezavo *Uredi*, ki se nahaja sredi kartice. S tem se odpre obrazec na Sliki 4.6. Pri urejanju kartic program vedno preverja prisotnost in pravilnost podatkov.



Slika 4.6: Prikaz zaslonske slike urejanja kartice oz. delovne naloge.

4.3 Razvoj spletne aplikacije

V tem podpoglavju bodo predstavljeni nekateri (morda) zanimivejši deli razvoja spletne aplikacije sistema kanban.

S skriptnim jezikom Sass lahko privarčujemo veliko časa že z uporabo najosnovnejših elementov. Na gornji strani Slike 4.7 je koda Sass in na spodnji strani koda, ki se interpretira v CSS. Z uporabo spremenljivk in gnezdenja v skriptnem jeziku Sass je oblikovanje hitrejše, enostavnejše in bolj pregledno.

```
// ====== Sass ======== //
.progress-bar {
  background-color:$skyblue;

  &.progress-alert{
    background-color:red;

    &.full{
      background-color: $darkpink;
    }
  }

  &.progress-good {
    background-color:green;
  }
}

// ====== CSS ======== //
.progress-bar {
  background-color: #00bae9; }
  .progress-bar.progress-alert {
    background-color: red; }
    .progress-bar.progress-alert.full {
      background-color: #cc1455; }
  .progress-bar.progress-good {
    background-color: green;
```

Slika 4.7: Prikaz skriptne kode Sass (zgoraj) in CSS (spodaj).

Ker ogrodje Laravel omogoča migracije, ki smo jih že omenili v podpoglavju 4.1, so spremembe v bazi enostavne in hitre. Naslednji primer prikazuje dodajanje novega atributa v tabelo. V ukazni vrstici z ukazom

php artisan migrate:make add_activity_id_to_tasks_table --table=tasks

z ogrodjem Laravel ustvarimo novo migracijo, s katero bomo dodali atribut *activity_id* v tabelo *tasks* (Slika 4.8). Migracijo poženemo z ukazom

php artisan migrate.

```php
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class AddActivityIdToTasksTable extends Migration {

    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('tasks', function(Blueprint $table)
        {
                $table->integer('activity_id');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('tasks', function(Blueprint $table)
        {

        });
    }

}
```

Slika 4.8: Prikaz razreda, ki smo ga ustvarili preko ukazne vrstice.

```php
Route::filter('auth.basic', function()
{
    return Auth::basic();
});
```

Slika 4.9: Prikaz sita za avtentikacijo v spletno aplikacijo.

Na Sliki 4.10 so predstavljeni datoteka *routes.php,* usmerjevalnik, ki preusmerja odjemalce glede na zahtevane strani, in metode, s katerimi te strani zahtevajo. V datoteki *filter.php* implementiramo funkcije, ki skrbijo za določene omejitve v aplikaciji. Sito (angl. filter) realiziramo s kodo, ki je prikazana na Sliki 4.9. Na Sliki 4.10 pa prikazujemo, kako sito uporabimo pri omejevanju dostopa do spletne strani, ki zahtevajo avtentikacijo.

```
Route::group(['before' => 'auth'], function(){

    // Activities Routes
    Route::get('/activities/edit', [
        'as' => 'activities.edit',
        'uses' => 'ActivityController@index'
    ]);

    Route::patch('/activities/edit/{id}', [
        'as' => 'activity.update',
        'uses' => 'ActivityController@update'
    ]);

    Route::post('/addActivity', [
        'as' => 'activity.add',
        'uses' => 'ActivityController@create'
    ]);

    Route::delete('/activities/edit/{id}', [
        'as' => 'activity.destroy',
        'uses' => 'ActivityController@destroy'
    ]);

});
```

Slika 4.10: Prikaz dela usmerjevalnika, kjer uporabljamo omenjeno sito.

Tako kot migracije lahko tudi krmilnike ustvarjamo preko ukazne vrstice. Z ukazom

php artisan controller:make BacklogController

nam ogrodje Laravel ustvari datoteko BacklogController.php, ki jo zapiše po začetnih vzorcih. Primer krmilnika z implementacijo vidimo na Sliki 4.11, kjer smo implementirali dve (2) metodi: (1) *show($id)*, s katero pošiljamo podatke o določenem objektu *Task* z vhodnim parametrom *$id,* in (2) *update($id),* s katero urejamo atribute objekta *BacklogTask* z vhodnim parametrom *$id* in ga shranjujemo nazaj v podatkovno zbirko.

```
/**
 * Display the specified resource.
 *
 * @param  int  $id
 * @return Response
 */
public function show($id)
{
	$task = Task::findOrFail($id);
	return View::make('tasks.show')->withTask('task');
}

/**
 * Update the specified resource in storage.
 *
 * @param  int  $id
 * @return Response
 */
public function update($id)
{

	$task = BacklogTask::find($id);
	$task->title = Input::get('title');
	$task->body = Input::get('body');
	$task->user_id = Input::get('user_id');

	$task->save();

	Flash::message('Kartica z naslovom ' . $task->title . ' je bila uspešno
urejena.');

	return Redirect::to('backlog');
}
```

Slika 4.11: Prikaz dela krmilnika *BacklogController.php*.

Na Sliki 4.12 je prikazan pogled *activity.blade.php*. Vse datoteke, ki uporabljajo orodje Blade, se končujejo s končnico *.blade.php*. Zaradi preglednosti razvoja smo dele pogledov razdelili v več različnih datotek (funkcije *@extends* in *@include*) in bili pozorni na to, da se koda ne ponavlja. Iz sintakse lahko opazimo, da se stavki if in foreach začenjajo z znakom @, končajo pa se z oznakama *@endif* in *@endforeach*. Pri tem zavitih oklepajev ne uporabljamo.

```
@extends('layouts.master')

@section('content')

    <div class="well well-lg">Če želite, da aktivnost nima omejitve WIP, v 2.
stolpec vpišite vrednost 0.</div>
    @foreach($activities as $activity)
        @include('activities.layouts.activityForm')
    @endforeach

    <div class="clearfix"></div>

    @if($activities->count() < 6)
        <div class="add-form-row"></div>
       <button type="button" class="btn btn-success btn-lg pull-right add-
activity">Dodaj novo aktivnost</button>
    @else
        <div class="well well-lg">Dodajanje ni mogoče, maksimalna omejitev je 6
aktivnosti.</div>
    @endif

@stop
```

Slika 4.12: Prikaz pogleda aktivnosti.

Poglavje 5 Sklepne ugotovitve

Koncepti vitkega razvoja so v domeni programske opreme relativno mlada paradigma. Čeprav se zdi, da spada med agilne metode razvoja, ji mnogi prepisujejo samostojno vejo v procesu razvoja. Kakšne so podobnosti in razlike, smo poskušali odgovoriti v tem diplomskem delu.

Za umestitev konceptov vitkosti v proces razvoja programske opreme je v prvi vrsti potrebno razumevanje principov vitkega razvoja. Orodja so komplementarna in jih prilagodimo glede na potrebe razvojnega okolja in ekipe. Agilne metode in koncepti vitkosti poudarjajo različnost razvojnih ekip in organizacij, zato delovanje opisujejo širše in bolj abstraktno, kar se opazi tudi v diplomskem delu. V literaturi se mnogokrat opazi pristranskost in težnjo k paradigmi, ki jo opisuje, ko primerjamo različne metode razvoja. Med raziskovanjem domene je moč zaznati pomanjkanje literature in virov, ki se nepristransko opredeljujejo glede teh vprašanj. Vsekakor se zavedamo, da je strokovna primerjava metod zelo težavna, saj noben proces v razvoje programske opreme ni popolnoma enak in zajema izredno veliko število faktorjev, ki jih je treba pri tem upoštevati.

V diplomskem delu smo na kratko opisali začetke razvoja vitke proizvodnje, nato je sledil prenos konceptov vitkega razvoja v sam proces razvoja programske opreme. Opisali smo glavne značilnosti tradicionalnih plansko vodenih in agilnih metod ter se osredotočili na njihove glavne razlike. V nadaljevanju smo primerjali agilne in vitke principe razvoja in se podrobneje lotili obravnave konceptov vitkega razvoja. Predstavili smo osnovni model metode kanban in uporabo sistema pri razvoju programske opreme, pomembnost razumevanja kumulativnega diagrama toka in diagrama toka dodane vrednosti. V zadnjem poglavju diplomskega dela smo predstavili elektronski sistem kanban in opisali tehnologije, ki smo jih pri tej izdelavi aplikacije uporabili. Na konec smo umestili tudi nekaj zanimivih delov programske kode.

Literatura

[1] (2014) Lean Manufacturing Tools. Lean Manufacturing Tools. [Online]. Dostopno na: http://leanmanufacturingtools.org/49/history-of-lean-manufacturing/

[2] M. Poppendieck in T. Poppendieck, Implementing Lean Software Development: From Concept To Kash. Boston: Addison Wesley Professional, 2006.

[3] A. Kešetović. (marec 2012) Filozofija vitke proizvodnje - koncept, ki prinaša poslovne uspehe. [Online]. Dostopno na http://www.fm-kp.si/zalozba/ISBN/978-961-266-135-9/prispevki/025.pdf

[4] (2014) D. Browning. Lean Defined: What is Waste? [Online]. Dostopno na: http://www.leancor.com/blog/lean-defined-what-is-waste-in-logistics/

[5] T. Ohno, Toyota Production System. Tokyo: Diamond, Inc., 1978.

[6] J. Liker, The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer. New York: McGraw-Hill, 2004.

[7] S. Shingo, A Study of the Toyota Production System: From an Industrial Engeineering Viewpoint. Tokyo: Productivity Press, 1981.

[8] D. P. Cushman, S. S. King. (december 2014) Best Practices at the Dell Computer Corporation: Benchmarking a High-Speed Management Communication System. [Online]. Dostopno na http://www.sunypress.edu/pdf/60758.pdf

[9] J. P. Womack, D. T. Jones, Lean Thinking: Banish Waste and Create Wealth in Your Corporation. London: Simon & Schuster UK Ltd., 2003.

[10] (2014) R. M. Becker. SAE International. [Online]. Dostopno na: http://www.sae.org/manufacturing/lean/column/leanjun01.htm

[11] W. W. Royce, "Managing the development of large software systems: concepts and techniques," Proceeding ICSE '87 Proceedings of the 9th international conference on Software Ingineering, str. 328-338, 1987. [Online]. Dostopno na: http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

[12] T. Dyba in T. Dingsøyr, "Empirical studies of agile software development: A systematic review," Information and Software Technology 50, str. 834-858, 2008.

[13] M. Poppendieck in T. Poppendieck, Lean Software Development: An Agile Toolkit. Crawfordsville: Addison-Wesley, 2003.

[14] (2011) D. Huether. Waste in Software Projects. [Online]. Dostopno na: http://thecriticalpath.info/2011/07/07/waste-in-software-projects/

[15] Y. T. Krishnegowda H. K. Raju, "Value Stream Mapping and Pull System for Improving Productivity and Quality in Software Development Projects," Int. J. of Recent Trends in Engineering & Technology, zv. 11, str. 24-38, 2014.

[16] (2014) Wikiversity. Plan-driven Software Development. [Online]. Dostopno na: http://en.wikiversity.org/wiki/Plan-driven_software_development

[17] (2014) Agile Manifesto. Manifest agilnega razvoja programske opreme. [Online]. Dostopno na: http://agilemanifesto.org/iso/sl/

[18] K. Schwaber, Agile Project Management with Scrum. Redmon, Washington: Microsoft Press, 2004.

[19] K. Bent, Extreme Programming Explained: Embrace Change. Upper Saddle River, New Jersey: John Wait, 2005.

[20] P. Abrahamsson, N. Oza in C. Ebert, "Lean Software Development," IEEE Software, 2012.

[21] D. West in T. Grant, "Agile Development: Mainstream Adoption Has changed Agility," Forrest Research, Inc., 2010.

[22] B. Boehm in R. Turner, Balacing Agility and Discipline, Prva izdaja ed. Boston: Addison-Wesley, 2003.

[23] H. Kniberg in A. Ivarsson. (oktober 2012) Scaling Agile at Spotify. [Online]. Dostopno na http://ucvox.files.wordpress.com/2012/11/113617905-scaling-agile-spotify-11.pdf

[24] (2014) E. Bristow. The Wall Street Journal. [Online]. Dostopno na: http://deloitte.wsj.com/cio/2014/03/25/9-myths-about-agile/

[25] Net Objectives. (december 2014) An Agile Developers Guide To Lean Software Development. [Online]. Dostopno na http://www.netobjectives.com/files/AnAgileDevelopersGuideLeanSoftwareDevelopment.pdf

[26] D. K. Sobek II, A. C. Ward, J. K. Liker, "Toyota's Principles of Set-Based Concurrent Engineering," MIT Sloan Management Review, zv. 1, št. 2, 1999. [Online]. http://sloanreview.mit.edu/article/toyotas-principles-of-setbased-concurrent-engineering/?use_credit=0738069b244a1c43c83112b735140a16

[27] S. Peeters, "Applying Lean Thinking To Software Development," InfoQ eMag: Lean and Kanban, št. 10, str. 4-9, 2014.

[28] A. Sillitti, O. Hazzan, E. Bache, X. Albaladejo, "Simulating Kanban and Scrum vs. Waterfall with System Dynamics," Agile Processes in Software Engineering and Extreme Programming, str. 117-131, 2011.

[29] (2011) K. Waters. All About Agile. [Online]. Dostopno na: http://www.allaboutagile.com/lean-principle-7-optimise-the-whole/

[30] (2006) D. H. Preuss. InfoQ. [Online]. Dostopno na: http://www.infoq.com/news/Discussion-Decide-Late-as-Poss

[31] T. M. Budau. (avgust 2012) Universitat Bonn. [Online]. Dostopno na http://sewiki.iai.uni-bonn.de/_media/teaching/labs/xp/2012b/seminar/2-kanban.pdf

[32] D. J. Anderson, Kanban: Successful Evolutionary Change For Your Technology Business. Washington: Blue Hole Press, 2010.

[33] M. Skarin. (januar 2013) Ten Kanban Boards and their context. [Online]. Dostopno na

https://dl.dropboxusercontent.com/u/1638038/publikationer/10%20kanban%20boards%20and%20their%20context/10%20different%20kanban%20boards%20and%20their%20context%20-%20mskarin.pdf

[34] M. Griffiths. (2007) Leading Answers. [Online]. Dostopno na http://leadinganswers.typepad.com/leading_answers/files/creating_and_interpreting_cumulative_flow_diagrams.pdf

[35] Agile Alliance and Institut Agile. (december 2014) Agile Alliance. [Online]. Dostopno na http://guide.agilealliance.org/guide/leadtime.html

[36] S. Arnold in P. White. (junij 2014) Leaner software development using DevOps. [Online]. Dostopno na https://www.ibm.com/developerworks/rational/library/leaner-software-development-with-the-aid-of-collaborative-lifecycle-management/

[37] (2013) S. Tendon. Replacing Cycle Time with Flow Time. [Online]. Dostopno na: http://tameflow.tendon.net/post/56252147592/replacing-cycle-time-with-flow-time

[38] D. Chhajed, T. J. Lowe, Building Intuition: Insights From Basic Operations Management Models And Principles. New York: Springer-Verlag, 2008.

[39] (2011) G. Marshall. HTML5: What is it? [Online]. Dostopno na: http://www.techradar.com/news/internet/web/html5-what-is-it-1047393

[40] Y. Fan, Cascading Style Sheets.: Kemi-Tornio University Of Applied Sciences Tehnology, 2010.

[41] D. Flanagan, JavaScript: The Definitive Guide, 6th ed. Sebastopol: O'Reilly Media Inc., 2011.

[42] R. York, Beginning Javascript and CSS Development with jQuery. Indiana: Wiley Publishing, Inc., 2009.

[43] (2015) Bootstrap. Bootstrap: The world's most popular mobile-first and responsive front-end framework. [Online]. Dostopno na: http://getbootstrap.com

[44] S. Kumar, J. Kumar, S. Kumar, "Introduction on PHP - Hypertext Preprocessor," Shravan et al./ IJAIR, zv. 2, št. 6, str. 328 - 343, 2003.

[45] (2015) Laravel Book. Laravel Arhitecture. [Online]. Dostopno na: http://laravelbook.com/laravel-architecture/

[46] (2013) M. Surguy. History of Laravel PHP Framework. [Online]. Dostopno na: http://maxoffsky.com/code-blog/history-of-laravel-php-framework-eloquence-emerging/

[47] S. McCool, Laravel Starter. Birmingham, Združeno Kraljestvo: Packt Publishing Ltd., 2012.

[48] S. Suehring, Mysql Bible. New York: Wiley Publishing Inc.

[49] H. K. Chan, X. Wang B. Hou, "A Case Study of Just-In-Time System in the Chinese Automotive Industry," Proceedings of the World Congress Of Engineering 2011, zv. 1, str. 904 - 908, julij 2011.

[50] M. S. Islam in S. Tura, Exploring the difference between Agile and Lean: A stakeholder perspective. Uppsala: Uppsala University, 2013.

[51] R. J. Schonberger, Let's fix it!: Overcoming the Crisis In Manufacturing, Prva izdaja ed. New York: Free Press, 2001.

[52] M. Windholtz. (februar 2008) Lean Software Development. [Online]. Dostopno na http://www.engineering.upm.ro/master-ie/sacpi/mat_did/info206/docum/LeanSoftwareDev-Short.pdf