PROGRAMSKI ALAT ZA SAMOPOSLUŽNE SUSTAVE ZADATAKA

Lukač, Luka

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

Programski alat za samoposlužne sustave zadataka

Luka Lukač

Zagreb, veljača 2020.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, 10.02.2020.

Predgovor

Želim se zahvaliti svojoj supruzi, obitelji i prijateljima na pruženoj neizmjernoj potpori tijekom studija.

Zahvaljujem se svome mentoru profesoru Miroslavu Popoviću na ukazanom povjerenju, suradnji i svesrdnoj pomoći oko izrade ovog završnog rada.

Želio bih se i zahvaliti svim predavačima i asistentima Visokog učilišta Algebra koji su mi nesebično prenosili znanja, iskustva, vještine i pomagali mi na mom akademskom putu.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Cilj ovog završnog rada je predstaviti programsko rješenje za samoposlužne sustave zadataka kao jednu od glavnih metoda prilikom izrade aplikacija agilnim razvojem. Za izradu programskog rješenja korišteni su alati Visual Studio i SQL Server 2014 Managment Studio. Programski okvir koji je korišten je ASP.NET.

Uz pregled i opis glavnih *Lean* metoda ovaj rad donosi i praktične primjere korištenja nekih od metoda u izradi programskog rješenja. Kanban ploča koja služi za pregled i upravljanje zadacima, grupama i timovima uvelike pomaže pri organizaciji i vizualizaciji radnih zadataka. Korisnici u svakome trenu znaju stanje iskorištenih i raspoloživih resursa i prema tome se mogu ravnati u daljnjem razvoju programskog rješenja.

Kvalitetno upravljanje resursima je ključ uspješnosti projekta, garancija zadovoljstva krajnjeg korisnika rješenjem koje zadovoljava uvjete i isporukom u zadanom vremenskom roku. Za nas kao izvođača to u konačnici znači zadovoljstvo kvalitetnim programskim rješenjem i veći profit.

Ključne riječi: samoposlužni sustavi zadataka, agilni razvoj aplikacija, *Lean* metode, upravljanje resursima, uspješnost projekta, zadovoljstvo korisnika.

Summary

The goal of this project is to present a software solution for pull systems as one of the main methods when designing applications with agile development. Tools Visual Studio and SQL Server 2014 Managment Studio were used to create the software solution. The programming framework used is ASP.NET.

In addition to reviewing and describing the main Lean methods, this paper also provides practical examples of using some of the methods in developing a software solution. The Kanban board, which is used to view and manage tasks, groups and teams greatly helps organize and visualize work tasks. The users in each moment are aware of the state of the used and available resources and can therefore be guided in the further development of the software solution.

Good resource management is the key to project success, guaranteeing end-user satisfaction with an eligible solution and delivery within the given timeframe. For us as a contractor, this ultimately means satisfaction with a quality software solution and higher profits.

Keywords: pull systems, agile application development, Lean methods, resource management, project success, customer satisfaction.

Sadržaj

1. Uvod

Prilikom izrade naručenog programskog rješenja potrebno je znati rasporediti resurse koji se posjeduju kako bi se osigurao završetak u zadanom roku i na zadovoljstvo naručitelja. Za praćenje i organiziranje uspješnog vođenja projekta potreban je alat koji to osigurava. Kao jedna od glavnih *Lean* metoda razvoja aplikacija prepoznati su samoposlužni sustavi zadataka kojima je cilj upravo praćenje i organizacija procesa proizvodnje programskog rješenja. Kanban ploča je vizualna reprezentacija navedene metode i ona omogućava raspoređivanje i praćenje zadataka po tijeku, fazama njihovog rješavanja.

Cilj ovog rada je prezentirati rješenje za upravljanje resursima agilnim razvojem aplikacije. Proučavajući sve *Lean* metode razvoja došlo se do zaključka da bi baš odabrana – samoposlužni sustavi zadataka bila odgovarajuća. Na početku rada je pregled svih *Lean* alata, metoda koje se koriste. Nije potrebno koristiti sve metode prilikom razvoja aplikacije kao garanciju uspjeha nego samo koristiti one koje nam dodaju vrijednost projektu. Odabir je pao na samo neke od njih a to su prepozanavnje otpada, izrada sheme proizvodnog procesa, iteracije i refaktoriranje.

U prvome poglavlju ovog rada predstavlja se *Lean* filozofiju razvoja aplikacija i sve metode. Zatim na red dolazi detaljan opis samoposlužnih sustava zadataka i prikaz nekih od alata koji postoje na tržištu i pružaju traženu funkcionalnost. Poglavlje nakon toga donosi analizu strukture aplikacije i alata korištenih pri izradi programskog rješenja. U završnom poglavlju pruža se uvid u samo praktično rješenje aplikacije. Navodi se njena svrha, analizira se razvoj aplikacije po korištenim *Lean* metodama i obrazlaže se ista prema strukturnim slojevima.

2. Lean metode

2.1. Uklanjanje otpada

- prepoznavanje otpada (*tool1*)
- izrada sheme proizvodnog procesa (*tool2*)

Temelj pravilo u *lean* načinu razmišljanja je uklanjanje otpada. On je izvor za sva druga pravila i principe. Za početak potrebno je prepoznati otpad, zatim pronaći uzroke njegova nastanka i ispraviti pogreške. Zatim konstantno ponavljati ovaj proces da bi se imale pravovremene informacije za poboljšanje procesa.

2.1.1. Prepoznavanje otpada

Otpad predstavljaju sve komponente koje ne pridonose, ne dodaju vrijednost projektu. Kako prepoznati potencijalni otpad jedan je od glavnih ciljeva *lean* načina razmišljanja. Nameće se zaključak da jedino analiza i pisanje kôda nisu otpad.

Za početak je potrebno napraviti analizu svih dijelova uključenih u projekt koji nisu direktno vezani uz analizu i pisanje kôda te ustvrditi da li oni na bilo koji način pridonose povećanju vrijednosti konačnog proizvoda.

Treba se pripaziti na sljedeće:

- Nedovršeni posao
- Dodatni procesi
- Dodatne značajke proizvoda
- Prebacivanje iz zadatka u zadatak
- Prazan hod u proizvodnji
- Suvišno kretanje (centralizacija)
- Nedostaci na proizvodu

2.1.2. Izrada sheme proizvodnog procesa

Izrada sheme cjelokupnog proizvodnog procesa uvelike pomaže utvrditi koliko je vremena i resursa potrebno za stvaranje sebi i krajnjem korisniku vrijednosti. Analizom proizvodnog procesa ukazuje se na probleme na koje se može.

Sama izrada sheme proizvodnog procesa je zadatak koji se može obaviti samostalno ili uz pomoć dodatnih alata. Najboljim se pokazala praksa da samostalnog prolaska kroz vlastiti

proizvodni proces i bilježenje svih dijelova. Time se dobivaju podatke koji nam govore o vremenu koje protekne od početaka procesa do trenutka završetka.

Najbolje je započeti na mjestu gdje zahtjevi korisnika ulaze u proces. Nakon toga valja proći sve pojedine dijelove sustava koji su uključeni u proces, zabilježiti sve potrebne korake koji se obavljaju te njihovo vrijeme trajanja. U to su uključeni svi sudionici, oni u vlastitoj organizaciji i oni s strane korisnika. Na kraju se izrađuje vremenska liniju na kojoj se vidi koliko je potrebno proći vremena od zaprimanja korisnikovih zahtjeva do isporuke proizvoda. Dobiva se uvid koliko vremena se provede u aktivnostima koje dodaju vrijednost proizvodu ali i koliko vremena se provede u čekanju. Detaljnom analizom dolazi se do zaključaka koji bi mogli pomoći u skraćivanju vremena proizvodnje.

Izrada sheme u duhu agilnog razvojnog pristupa podrazumijeva korištenje inkrementalnog razvoja proizvoda. Razvija se u fazama, inkrementima a prelazak na novu fazi se događa tek nakon što je prethodna uspješno završena. Prikupljaju se zahtjevi prema potrebi, pokušavaju se uskladiti promjene dizajna proizvoda sa istovremenom izradom kôda, planiraju se što je ranije moguće redovne implementacije dijelova proizvoda.

Cilj je skratiti vrijeme čekanja i povećati vrijeme provedeno na stvaranju vrijednosti korisnicima. Uzimajući u obzir sve dijelove proizvodnog procesa lako se uočavaju uska grla, mjesta na kojima se povećava vrijeme čekanja. Brz protok kroz sve dijelove procesa osigurava i brzu reakciju na zahtjeve korisnika.

Najveća prednost izrade shema proizvodnog procesa je uvid u sve dijelove istog. Fokusiranjem isključivo na proizvod i stvaranjem dodane vrijednosti za korisnika ona omogućava promjene na cjelokupnom procesu iz tog gledišta, bez dodatnih internih faktora koje posjeduje svaka tvrtka poput imovine i radnog kadra.

2.2. Stalno unaprjeđenje znanja

- povratne informacije (*tool3*)
- iteracije (*tool4*)
- sinkronizacija (*tool5*)
- razvoj temeljen na skupu informacija (*tool6*)

Iako *lean* način razmišljanja vuče korijene iz proizvodnje, *lean* principi se mogu primijeniti na raznim područjima. Važno je napomenuti se *lean* principi iz proizvodnje ne mogu direktno prevesti u razvoj programskih rješenja, postoje neke razlike. Dok se proizvodnja

bazira na izradi proizvoda koji je bez varijacija, svaki je isti, razvoj programskih rješenja spada u kategoriju razvojnih procesa. Za razliku od proizvodnih, razvoji procesi uključuju stvaranje na temelju pokušaja i pogrešaka, izradu raznih varijacija dok jedna ne zadovolji potrebe korisnika. Proizvod koji se isporučuje je uvijek isti dok je usluga podložna promjenama, zahtjevima korisnika pa se kao takva stalno mijenja. Razlike postoje i u pogledu na kvalitetu. Kod proizvoda je bitno da zadovoljava kriterije koji su definirani prije proizvodnje dok se kod usluge kvaliteta svodi na zadovoljavanje korisnikovih zahtjeva o funkcionalnosti iste.

Kod razvoja usluga stalno unaprjeđenje znanja uvelike dolazi do izražaja prilikom ispravaka u dizajnu i njihovoj implementaciji. Koristi se princip iteracija s preslagivanjem kôda. Nakon svake revizije postojećeg kôda dolazi se do nekih zaključaka kako isti preraditi da bolje odgovara na zahtjeve korisnika. Konstantnim promjenama stječu se nova znanja koja se dalje koriste u razvoju ove ali i svake nove usluge.

2.2.1. Povratne informacije

Povratne informacije su ključ uspješnog rješavanja problema u razvoju i neophodne su za daljnji pravilan razvoj proizvoda. Služe kao kontrola da li je sve pravilno implementirano i da li nešto zadovoljava postavljene kriterije.

Povratne informacije se koriste svuda i iako donekle dižu razinu kompleksnosti sustava poželjne su jer daju uvid u pravo stanje stvari. Pravovremena reakcija je ključ u dobivanju pozitivnih kritika od strane korisnika. Kao primjer može se pogledati dio rješenja koji ispituje da li korisnik ima prava pristupa. Koristi se forma polja koje je potrebno ispuniti s korisničkim imenom i lozinkom. Ukoliko se uneseni podaci ne nalaze u bazi podataka korisnik se ne pušta u sustav. Na temelju povratne informacije koja se ovdje dobiva direktno se utječe na rad cjelokupnog proizvoda. Drugi primjer korištenja povratnih informacije je kod završetka pisanja kôda. Bilo bi poželjno imati nekoga tko bi glumio potencijalnog korisnika. Povratna informacija se odbije odmah i eventualni ispravci se mogu raditi na licu mjesta. Ovo je dobra preporuka za svakog programera jer se ne gubi vrijeme na prebacivanje s zadatka na zadatak.

2.2.2. **Iteracije**

Iteracija je korisno povećanje softvera koji je dizajniran, programiran, testiran, integriran i isporučen tijekom kratkog, fiksnog vremenskog okvira[1]. Prilikom odluke poduzeća da počne primjenjivati *lean* metode razvoja važno je usvojiti načelo izrade i isporuke određenih dijelova proizvoda baš na vrijeme kada su potrebni. Velika prednost ove tehnike je povećanje komunikacije između zaposlenika koji u pravo vrijeme i na pravome mjestu donose odluke i rješavaju eventualne probleme. Da bi se pravilno započelo potrebno je izraditi prototip na kojem će se simulirati sve potrebe i zahtjevi koje se očekuju pri procesu proizvodnje. Prototip daje korisne povratne informacije na kojima se može temeljiti daljnji razvoj proizvoda. Prototip služi za simulaciju dok iteracije služe za proizvodnju radnog dijela proizvoda. Proizvod se tako poboljšava u svakoj pojedinoj iteraciji a dio proizveden u pojedinoj iteraciji je odmah je spreman za korištenje jer je prošao fazu testiranja i integracije. Zato je važno da dio zaposlenika koji je zadužen za testiranje bude uključen od prvog trenutka, prve iteracije u proces proizvodnje. Na temelju povratih informacija dobivenih tijekom i nakon svake iteracije proizvod se može razvijati u pravome smjeru.

Tri su temeljna načela kojih se treba pridržavati. Prvo je kako organizirati svoj red proizvodnje. Bitno je da je što kraći radi bolje kontrole nad njime. Dobiva se brz protok kroz sustav, komunikaciju među zaposlenicima i bolju raspodjelu resursa. Drugo su kratke iteracije koje pomažu da se reagira na stvarne probleme a ne na prognoze mogućih problema. Treće je da iteracije treba prihvatiti kao točke za usklađivanje svih timova zaposlenika koji rade na projektu ali i krajnjih korisnika. Svakom iteracijom dobiva se dio proizvoda koji je spreman biti pušten na korištenje. Tim pristupom potiče se donošenje odluka i omogućava se da timovi rade neovisno jedan o drugome a da pri tome ne odlutaju daleko od zajedničkog cilja.

Kako onda planirati pravilno što će biti napravljeno u jednoj iteraciji i koliko će biti njeno trajanje? Iteracije bi trebale obuhvaćati točno onoliko posla koliko je potrebno da se riješi korisnikovi zahtjevi. Ako su zahtjevi previše kompleksni onda ih treba rastaviti u više iteracija. Na početku svake iteracije je faza planiranja u kojoj razvojni tim zajedno s predstavnicima korisnika odlučuje o prioritetima zahtjeva na temelju njihovog troška jer je važno što prije stvoriti što veću vrijednost za korisnika. Rizične stvari treba rješavati što

[1] Poppendieck, 2003.

ranije, one bi trebale imati najveći prioritet. Kada se odlučuje o trajanju same iteracije dva su pristupa. Jedan zagovara da sve iteracije traju isto vremensko razdoblje kako bi se zaposlenici navikli na određeni ritam rada. Drugi zagovara da duljina iteracije ovisi o trenutnim okolnostima za svaku pojedinu iteraciju. Nema savršenog odgovora koji pristup je bolji. Najtočniji odgovor je da bi iteracija trebala trajati onoliko dugo koliko je potrebno da se korisniku pruže informacije o tijeku proizvodnog procesa a da se pritom odradi smisleni ciklus dizajniranja, izrade i testiranja proizvedenog dijela proizvoda. Razvojni tim tu ima veliku ulogu da procjeni koliko posla je njegov tim ili timovi sposobno odraditi da se ispune uvjeti kratkih iteracija i pouzdane isporuke. Greške su uvijek moguće, u tome slučaju uvijek je dobro isporučiti i samo dio napravljenog zadatka nego sve isporučiti kada bude završeno.

Na temelju korisničkih zahtjeva razvojni tim mora donijeti odluku što će biti napravljeno u određenoj iteraciji. Najbolji način za to su konzultacije s timom da se vidi koliko posla oni vjeruju da može biti odrađeno u periodu te iteracije. Bitno je razumno odabrati. Dobro iterativno planiranje pruža zadovoljstvo kupcu jer mu isporučuje značajke koje je tražio i daje mu nadu da će i ostale značajke biti ispunjene u zadanome roku. To samom timu daje motivaciju da dalje nastavi sa uspješnim radom pružajući im pri tome osjećaj uspjeha.

Ovdje se mora spomenuti i pitanje da li će ovakav razvoj dovesti do krajnjeg završetka proizvoda. Velika je bojazan da će neprestane konvergencije u razvoju produljiti krajnji rok završetka proizvoda u nedogled. Što ako se problemi uočeni povratnim informacijama gomilaju toliko brzinom da ih se ne stigne riješiti? To se može spriječiti ograničavanjem količine promjena korisničkih zahtjeva na početku iteracije. Na taj način tim se fokusira na isporuku značajki donesenih u planu na početku iteracije. Prema tome optimalno razdoblje za povratne informacije bi trebalo biti kratko, ali opet ne prekratko da se na njih ne stigne reagirati. Sve zapravo ovisi o dinamici svake pojedine situacije i prilagodbi na nju. U dinamičnom okruženju česte povratne informacije povećavaju produktivnost tima.

Dobra konvergencija se postiže rješavanjem prioritetnih značajki prije onih koje nisu toliko značajne. Utvrditi koje značajke imaju najviši prioritet korisniku nije lako. Na početku svakog projekta korisnik ni sam nije svjestan što će se zapravo koristiti od cijelog opsega mogućnosti proizvoda. Zato je potrebno s korisnikom dogovoriti koje značajke će se prve napraviti kako bi proizvod donio maksimalnu vrijednost. Kada se to ispuni može se krenuti na novi set značajki ali isto tako se može i revidirati napravljeno a potom i uraditi korekcije ako su potrebne. Što dalje napreduje razvoj korisnik će imati bolju sliku što mu je zapravo

potrebno i dragocjeno vrijeme neće se gubiti na razvoj značajki koji u stvari nisu bitne. Temeljem tih kratkih iteracija može se procijeniti koliko vremena je potrebno da se određeni dio posla obavi, točnije može se mjeriti brzinu rada timova. Praćenjem rada u prošlosti dobiva se slika koliko će rad trajati u budućnosti te na taj način doći do krajnje točke završetka proizvoda. Uvid u pravo stanje stvari trebalo bi biti transparentno i zaposlenicima i krajnjem korisniku. Naime ako se primijeti da tim ne može reagirati na stalno povećanje korisničkih zahtjeva i da procijenjeno vrijeme završetka projekta ne konvergira prema željenoj točci treba razmisliti o uklanjanju nekih značajki sa popisa za izradu. Može se dogoditi i situacija da tim ne može ispuniti očekivanja bez da su tražene izrade novih značajki, u tome slučaju treba pružiti pomoć timu. Sve ove činjenice mogu biti vidljive ako se iterativno razvija proizvod, a ako su vidljive na njih se može i reagirati. Tom reakcijom smanjuje se rizik od otkazivanja cijelog projekta.

2.2.3. **Sinkronizacija**

Prilikom rada nekoliko osoba na istom zadatku potrebno je njihov rad sinkronizirati. Za svaki složeni razvojni proces sinkronizacija je temelj za uspješan rad.

Organizacija rada timova može biti različita, završetak nekih dijelova kôda pojedinih osoba može varirati. Izgradnja cjelokupnog sustava zapravo ovisi o sinkronizaciji urađenih poslova više osoba koje su vlasnici određenog dijela kôda. Svatko radi svoj dio posla, konstantno provjerava da li je netko radio kakve promjene u međuvremenu i na njih reagira. Provodi testove na svojem modelu te nakon toga integrira rješenja na zajednički kôd. Ukoliko testovi prođu bez problem sinkronizacija je uspješna, ukoliko testovi ne prođu potrebno ju je ponoviti dok ne uspije sve proći bez grešaka. U praksi je bolje imati što kraća razdoblja za sinkronizaciju jer se u kraćem vremenu gomila manje problema a lakše se rješava manju količinu problema nego veću. Potreba za automatiziranim testiranjem kôda je ovdje neizbježna. Potrebno je automatizirati testiranja kako bi se pravovremenim uvidom u stanje stvari uspjelo što prije reagirati. Ponekad se javljaju problemi s automatiziranim testovima u smislu da predugo traju ili su previše kompleksni. U tim slučajevima je potrebno simulirati slojeve aplikacije koje nam usporavaju rad (npr. pristup bazi podataka) i na taj način omogućiti bržu izvedbu testova. Pravilo je da male i česte isporuke mogu stvoriti male probleme a isporuke u dužim vremenskih razdobljima stvaraju veće i kompleksnije probleme koje je teže i dugotrajnije za riješiti.

Prolazna aplikacija je još jedno od rješenja koje se može koristiti kao pomoć kod sinkronizacije. Njena svrha je stvoriti okvire na kojima će se temeljiti daljnji razvoj aplikacije. Nju najčešće izrađuje napredni tim. Sama aplikacija bi trebala biti jednostavan model koji će simulirati način rada konačne aplikacije. Može pokazati koji dijelovi su problematični a koji nisu. Potrebno ju je pustiti u produkciju čim bude gotova. Putem nje ulazi se u sami sustav a dobivene informacije određuju put kojim za konačno rješenje za optimizaciju resursa.

Tradicionalniji pristup je napravit kompletan nacrt na samome početku rada. Pri tome valja imati na umu da se rješenje sastoji od dva dijela – interakcije između raznih dijelova i unutar samih dijelova. Prvo je potrebno razviti interakciju između dijelova proizvoda kako bi svi zajedno mogli funkcionirati. Kada se izvede taj obujam posla svaki od timova može krenuti s izradom vlastitog dijela i dalje imajući na umu što češću integraciju sa svim ostalim dijelovima aplikacije. Prednosti ovog pristupa su da se problemi interakcije između sustava rješavaju u ranoj fazi projekta i da su pravi temelj za daljnju izgradnju aplikacije. Sinkronizacija koja je na ovaj način postignuta je odličan pokazatelj. Pruža brz razvoj ostatka pojedinih dijelova bez mogućnosti da se dovede u pitanje rad cjelokupnoga sustava.

2.2.4. **Razvoj temeljen na skupu informacija**

Dva su pristupa komunikaciji prilikom donošenja odluka i prikupljanja informacija. Jedno se temelji na komunikaciji vezano za izbore a drugo na ograničenjima. Ušteda vremena i resursa je u drugom pristupu – pristupu temeljenom na ograničenjima – na temelju skupa informacija.

Kako to primijeniti na razvoj proizvoda? Ponudi se više opcija, odrede se ograničenja i na temelju toga se pronađe zadovoljavajuće rješenje. Na taj način se ne radi odmak od iterativnog načina razvoja već mu se daje punina. Cilj svake iteracije je proizvesti minimalnu količinu funkcionalnosti da se zadovolje osnovni zahtjevi te iteracije. Ne razvijaju se pojedini dijelovi do kraja već se pušta da dizajn pojedinih dijelova uslijedi kada to bude potrebno. Za primjer može se pogledati aplikacija koja treba prikazati stanje skladišta na temelju nekog unosa, npr. šifre artikla. U iteraciji kada se izrađuje forma za upis ili izbor željenih podataka neće se raditi i konačan dizajn, izgled te forme. Neće se paziti na boju ili font slova već se koncentrira da forma ima sve potrebne dijelove i da je moguće poslati prikupljene informacije dalje onoj komponenti koja će dati rezultat prema zadanim informacijama. Imati više rješenja za dijelove aplikacije znači veću mogućnost da se iz svega

na kraju izvede najbolje rješenje. U trenutku kada će to biti potrebno ima više izbora kako spojiti više rješenja u jedno ili jednostavno se može odabrati najbolje. Imajući konstantno na umu sva ograničenja olakšava se donošenje pravih odluka u pravo vrijeme jer uključuje u proces odlučivanja sve one koji na bilo koji način imaju veze s istima. Uskladiti dizajn i provedbu istog u kôd je ključno a ovom metodom se dobiva ispravan i najbrži mogući način savladavanje prepreka za daljnji uspješan razvoj konačnog proizvoda.

2.3. Odlučivanje što je kasnije moguće

- mogućnosti (*tool7*)
- posljednji odgovorni trenutak (*tool8*)
- donošenje odluka (*tool9*)

U ovome poglavlju govoriti će se o donošenju odluka što je kasnije moguće u razvoju proizvoda. Kako se dizajn proizvoda konstantno mijenja tijekom vremena tako dolazi do potrebe da za izmjenom već nekih napravljenih dijelova koji zahtijevaju bolja rješenja. Dolazi do velikih gubitaka u resursima ukoliko se mora sve raditi ponovo zato što nije sve pripremljeno za situaciju da će u narednim iteracijama doći do promjena. Za primjer može se uzeti slijedeće: u početku postoji konačno rješenje za dohvat podataka, npr. baza podataka na temelju zahtjeva i potreba korisnika na početku projekta. Tijekom vremena i razvoja proizvoda došlo se do novih spoznaja koje zahtijevaju potpuno drugačiju strukturu baze podataka. Tada je potrebno iznova raditi bazu podataka i na taj način raditi ponovo dio posla koji je već jednom napravljen. Puno bolje rješenje bi bilo da je u početku napravljena baza podataka samo s osnovnim dizajnom kojeg je moguće nadograđivati kako se razvija proizvod te joj stalno pomalo dodavati dijelove koji su potrebni kada se za njih ukaže potreba. Dolazi se do zaključka da je bitno postaviti prave temelje proizvoda na najvišoj razini pa onda tek razvijati pojedine detalje i značajke. Postavljanjem pravih temelja u budućim iteracijama dobiva se mogućnost istraživanja koje rješenje bi najbolje odgovaralo potrebama i sve to na već dobro postavljenim temeljima. Takav istovremeni razvoj pruža veću sigurnost za nepotrebno povećanja troškova i jednostavnije rješavanje promjenljivih zahtjeva jer omogućava da se velike promjene odgode sve dok se ne razmotre sve opcije. Promjene će uvijek biti prisutne a na ovaj način dobiva se pravo rješenje kako ih uspješno savladati.

2.3.1. **Mogućnosti**

Kako pružiti korisniku zadovoljstvo korištenjem određenog proizvoda? Koja je garancija da će korisnik njime biti zadovoljan? Kao i svugdje oko nas najteže je promijeniti način na koji netko razmišlja. Većina poslovnih odluka se ne može s vremenom promijeniti nego ostaju takve kave jesu. Na koji onda način pružiti korisniku jamstvo da će dobiti proizvod koji je tražio? Jedini ispravan odgovor je dati mu više različitih mogućnosti na temelju kojih on svojim izborom donosi odluku koja verzija njega najviše zadovoljava. Mogućnosti su alat kojim se dobiva na vremenu odgode donošenja konačnih odluka do zadnjeg trenutka. U nikojem pogledu ne predstavljaju obavezu već daju za pravo da se poradi na njima dok se ne ustanovi koja je verzija najprihvatljivija za korisnika. To će se postići tako da se pojedinu mogućnost napravi i testira da li zadovoljava potrebe korisnika. Ukoliko zadovoljava, završen je taj dio posla, ukoliko ne onda se kreće izrađivati i testirati neku drugu pruženu mogućnost i gleda da li će ona odgovarati korisniku. U ovom slučaju gube se jedino resursi potrebni za izradu mogućnosti koja nije zadovoljila. Što onda činiti glede odabira načina rada? Da li koristiti predvidljivi ili prilagodljivi proces? Svaki ima svojih prednosti i mana. Predvidljivi proces je dobar za korištenje ako se krene s pretpostavkom da se ništa neće mijenjati do samog završetka proizvodnog tijeka. Odluke koje se donose na temelju pretpostavki na početku razvoja proizvoda vežu na sebe sve daljnje odluke u tijeku proizvodnog procesa. Što se dalje napreduje s istim veća je mogućnost da će se nešto promijeniti i da će biti potrebno i neke prethodne odluke promijeniti što može zadavati jako puno poteškoća. Prilagodljivi proces se temelji na pretpostavkama da će se sigurno nešto mijenjati tijekom proizvodnog procesa. To je realnije razmišljanje jer puno se stvari ne može predvidjeti. Zato ovaj način razmišljanja pruža više mogućnosti da se u samom razvoju proizvoda donosu odluke na temelju provedenih testiranja i istraživanja. Temeljem dobivenih podataka u pravo vrijeme i na pravi način rade se promjene s vrlo malim troškovima. Činjenica je da puno više resursa zahtijevaju promjene na stvarima koje su napravljene ranije u procesu proizvodnje kod predvidljivog nego kod prilagodljivog procesa. Agilni prilagodljivi proces smanjuje nesigurnosti odgađanjem odluka do trenutka kada potrebe i zahtjevi korisnika ne postanu potpuno jasni. To ne znači da ne postoje početni planovi. Planovi postoje ali ne idu u detalje koji nisu potrebni. Planovi daju fleksibilnost za odgovor na korisnikove potrebe. Mogućnosti su važne jer omogućuju odluke kojim smjerom se kreće na temelju činjenica dobivenih učenjem tijekom razvojnog procesa a ne nagađanjem činjenica na početku razvojnog procesa. Važno je imati na umu da svaka mogućnost zahtjeva

određene resurse i da je bitno iskustvo voditelja projekta u odluci koje mogućnosti razvijati dalje a koje ne. Mogućnosti su temelj za kvalitetno rješavanje korisnikovih zahtjeva i potreba ako uvijek neizvjesna budućnost krene u nepovoljnom smjeru.

2.3.2. Posljednji odgovorni trenutak

Istovremeni razvoj podrazumijeva razvijanje proizvoda s poznatim tek djelomičnim zahtjevima koje se dalje razvija u iteracijama te na temelju povratnih informacija polako se izgrađuje cijeli sustav. Takav razvoj omogućava donošenje odluka u pravom – posljednjem odgovornom trenutku. To je trenutak kada ne donošenje odluke uklanja alternativnu mogućnost.

Evo nekoliko primjera kako postići donošenje odluke u posljednjem odgovornom trenutku:

- Dijeliti nepotpune podatke o dizajnu unutar tima, zahtijevanje potpune informacije prije konačnog nacrta dizajna dovodi do povećanja trajanja povratnih petlji što uzrokuje da se prijevremeno donesu odluke. Kvalitetan dizajn treba nastati iz procesa otkrivanja kroz iteracije
- Organizirati dobru komunikaciju između zaposlenika koji znaju na koji način bi proizvod trebao pružiti krajnju vrijednost korisniku i zaposlenika koji izrađuju kôd. Svako rano razmjenjivanje nepotpunih informacija u tijeku razvoja u konačnici pridonosi proizvodu fleksibilnijem za nove nadogradnje ili promjene
- Razviti osjećaj za rješavanje promjena, ne preopteretiti se u početku s gomilom posla nego rješavati promjene postepeno u toj mjeri da ne uzrokuju probleme kada se ukaže potreba za njihovom implementacijom. To se može postići koristeći metode objektno orijentiranog programiranja (moduli, sučelja stabilna i odvojena od kôda, parametri, apstrakcije, algoritmi koji ne ovise o redoslijedu izvršavanja, izbjegavanje ponavljajućeg kôda, raspodjela odgovornosti-korištenje klasa, izrada samo kôda koji nam je u tome trenutku potreban, izbjegavanje izrade dodatnih opcija prije nego se za njima ukaže potreba ....)
- Razvijati osjećaj za ono što je neophodno na način da će se u ranim implementacijama brzo doći do zaključaka da li nešto u dizajnu nije kako treba
- Razvijati osjećaj kada donositi pojedine odluke, neke odluke ne mogu čekati jer na njima leže temelji vlastitog proizvoda dok neke mogu biti odgođene do vremena kada se ima sve potrebne podatke za njihovu izradu
- Razvijati sposobnost brze reakcije na promjene, što je kraće vrijeme odaziva na promjene to se dulje može odgađati odluke i u pravo vrijeme reagirati na korisnikove zahtjeve

2.3.3. Donošenje odluka

Postoje dva načina razmišljanja kako rješavati probleme – ići u dubinu ili širinu problema. Za oba je potrebna stručnost i iskustvo zaposlenika kako bi pravilno donio odgovarajuće

odluke. Ići u dubinu problema u načelu se svodi na pojednostavljenje cjelokupnoga sustava u početku kako bi se mogli postaviti temelji. Najčešće se događa da se početni zahtjevi previše pojednostave i da se tek kasnije primijeti da je trebalo još nešto dodati. To uzrokuje značajne troškove i kašnjenja. Za ovaj pristup potrebno je imati iskusnog zaposlenika koji će već na početku procijeniti i postaviti prave temelje za daljnji razvoj ali i garanciju da se zahtjevi neće mnogo mijenjati do kraja razvoja proizvoda. Ići u širinu problema znači odgoditi donošenje odluka do krajnjeg trenutka. Također je bitno iskustvo zaposlenika da može odlučiti što je u kojem trenutku neophodno. Za ovaj pristup ne mora se imati detaljno postavljene sve parametre u početku nego temeljem procesa učenja i eksperimentiranja dopustiti da razvoj tima evoluira tijekom vremena.

Postoje i dva načina kako se donose odluke – racionalno ili intuitivno. Racionalno znači provesti analizu svih faktora tako da ih se izdvoji iz konteksta i obradi analitičkim alatima. Intuitivno znači donositi odluke na temelju stečenih znanja i iskustava koji se mogu primijeniti ne trenutni kontekst situacije. Koje onda koristiti? Pokazalo se da intuitivno razmišljanje brže pronalazi probleme koji su rizični za projekt i na taj način dovode u pitanje isplativost i eventualni završetak. Ukoliko nema dovoljno iskustva i znanja da se procijeni za zadanu situaciju što bi trebalo, uvijek se može okrenuti racionalnom načinu donošenja odluka i na taj način doći do rješenja.

Ubrzavanje donošenja pravovremenih odluka je ključno. U hijerarhijskoj strukturi uvijek postoje oni koji se na višem položaju odlučivanja. Što je taj lanac odlučivanja duži, što uključuje više sudionika, vrijeme donošenja odluke pa tako i vrijeme reagiranja na promjene se smanjuje. Da bi se to vrijeme pokušalo smanjiti treba prvo pogledati strukturu složenog sustava. Tri glavne karakteristike su fleksibilnost, robusnost i samoorganizacija. Fleksibilnost znači brzu prilagodbu u promjenjivim uvjetima. Robusnost je značajka koja nam govori o tome ako i neki dio sustava ne funkcionira ostali dijelovi mogu obavljati svoju zadaću i bez njega. Samoorganizacija znači da sustav ne treba veliki nadzor od više instance u hijerarhiji da bi se održala njegova struktura i njegovo djelovanje kao cjelina. Kako onda ubrzati donošenje odluka na temelju ovih karakteristika? Odgovor leži u postavljanju jednostavnih pravila na svim razinama. Pravila nam omogućavaju odlučivanje bez potrebe da se traži odobrenje s više razine a da pri tome odluke koje se donesu budu opće prihvaćene. Na taj način se kompetentno može reagirati na okolinu koja se brzo mijenja jer se posjeduje alat kojim se u pravo vrijeme na temelju aktualnih informacija donose odluke u pravome trenutku.

Jednostavna pravila nisu instant rješenja ponuđena za određenu situaciju. Ona predstavljaju okvir prema kojem se na intuitivan način dolazi do rješenja za trenutni problem. Skup jednostavnih pravila su smjernice za zaposlenike koje im omogućavaju slobodu pri donošenju odluka u skladu s korporativnom politikom. Na taj se način podiže i osjećaj vrijednosti pojedinca u organizaciji.

2.4. Isporuka u što kraćem mogućem roku

- samoposlužni sustavi zadataka - pull sistem (*tool10*)
- teorija redova (*tool11*)
- trošak kašnjenja (*tool12*)

Isporuka u što je kraćem mogućem roku ne znači žuriti s isporukom već isporučivati u što kraćim intervalima gotove dijelove proizvoda. Na taj način smanjuju se mogućnost da korisnik promjeni svoje zahtjeve i da na taj način poveća obujam posla eventualnim promjenama. Bitno je naglasiti da svaki kôd koji nije testiran, koji nije implementiran, koji nije u produkciji potencijalno predstavlja rizik. Ako se pokaže da kôd ima greške ili ako dođe do promjena u zahtjevima od strane korisnika sav taj kôd postaje otpad. U konačnici s kraćim vremenom proteklim između isporuka dobiva se na vremenu i može se produljiti svoje vrijeme za donošenje odluka. Na taj način smanjuju se rizici i postavljaju se dobri temelji za odlučivanje na temelju pravovremeno dobivenih informacija.

Samoposlužni sustavi zadataka su rješenja na kojima se bazira ovaj rad. Njih će se pobliže objasniti u narednom poglavlju.

2.4.1. Teorija redova

Mjesto u proizvodnom procesu gdje dolazi do zastoja zove se usko grlo. Rješenje koje je ponuđeno za rješavanje ovog problema zove se teorija redova. Kada se čeka u redu prolazi vrijeme. To uključuje čekanje na uslugu i odrađivanje same usluge. Može uključivati i više redova. Mjerenje počinje ulaskom u red čekanja a zaustavlja se kada cijeli proces završi. To mjerenje se naziva vrijeme ciklusa. Čekanje u redu znači da ne postoji dovoljno resursa u procesu da se zadatak obavi bez čekanja. Postoje dva načina na koja se može postići smanjenje vremenskog ciklusa – osiguravanjem stalnog priljeva zadataka i osiguravanjem stabilnog vremena rješavanja zadataka.

Stalni priljev zadataka može se osigurati na način da se zadaju neki normativi koliko posla se može odraditi. Kanban ploča je primjer za normativ koliki je limit rada u pojedinoj fazi. Problem se može rješavati tako da se u proces puštaju male količine zahtjeva, da ne se ne čeka veliku količinu posla kojoj je onda vrijeme čekanje dugo barem koliko je i sama količina posla velika.

Kada se osigura stalni priljev zadataka može se pozabaviti i redovitim rješavanjem tih zadataka. Vrlo je važno i u ovoj fazi da zadatci budu manji jer što je manji zadatak manje je rizika i grešaka pa se time i smanjuje vrijeme potrebno za njegovu obradu. Generalno vrijeme rješavanja, obrade zadataka može se povećati ako se uvede više osoba, timova koji obavljaju zadatke. Još jedna od prednosti manjeg obujam zadatka je ta da se na taj način više zadataka može obavljati usporedno. Korištenje iteracija je vrlo osjetljivo na ovu temu, potrebno je završiti sve dijelove iteracije u pravo vrijeme kako bi se dobile povratne informacije i krenulo na vrijeme u novu iteraciju.

Ako ne postoji vrijeme čekanja onda je to savršen proces a to je nemoguće ostvariti. Mora se zaključiti da svaka pojava vremena čekanja pruža priliku za unapređenje a smanjivanjem vremena čekanja povećava se vlastitu efikasnost. Znači da se raste i razvija u pravom smjeru.

2.4.2. **Trošak kašnjenja**

Svako kašnjenje uzrokuje financijske gubitke. Da bi se znalo umanjiti trošak kašnjenja mora se prvo napraviti financijski model u kojem će se popisati sve čimbenike i na temelju dobivenih rezultata zaključiti koji nam više utječu na dobit od drugih. Važni čimbenici su: trošak razvoja, trošak proizvoda, trošak održavanja i vrijeme isporuke proizvoda. Da bi se trošak umanjio potrebno je donositi odluke na svim razinama poslovanja u pravome smjeru. Da bi se nešto moglo usporediti i odrediti što je bitnije važno je imati istu mjeru za sve čimbenike. Važno je da i svi uključeni u proizvodni proces budu upoznati s financijskim proračunom. Ako je svima ista jedinica mjere onda svi mogu donositi odluke prema istoj bazi. Pružiti svojim zaposlenicima financijske proračune znači dati im dobre smjernice u kojem smjeru će donijeti odluke. Odluke koje se na taj način donesu su efikasnije, zaposlenici su motiviraniji za rad a organizacija postaje sposobnija za brže reagiranje na promjene što je na današnjem konkurentnom tržištu od velike važnosti. Financijski planovi su idealna podloga za opravdanje smanjenja vremenskog ciklusa čekanja, rješavanja uskih grla i nabavke novih alata za unaprjeđenje vlastitog proizvoda i proizvodnog procesa.

2.5. Ugraditi integritet

- vidljivi integritet (*tool13*)
- sistemski integritet (*tool14*)
- refaktoriranje (*tool15*)
- testiranje (*tool16*)

Integritet podrazumijeva potpunost i cjelovitost nečega. Za programsko rješenje postoje dvije vrste integriteta: vidljivi i sistemski. Vidljivi integritet znači da se proizvodom zadovoljavaju potrebe korisnika ravnotežom funkcionalnosti, upotrebljivosti, pouzdanosti i ekonomičnosti. Sistemski integritet podrazumijeva da su svi važni dijelovi sustava u sinergiji, da zajedno funkcioniraju kao cjelina[2]. Ostvarivanje vidljivog integriteta nije moguće bez ostvarivanja sistemskog integriteta. Ali samo ostvariti sistemski integritet nije dovoljno da bi se postigao vidljivi integritet. Jedini način za postizanje integriteta proizvoda je uspostavljanje kvalitetne dvosmjerne komunikacije kako između raznih dijelova tima tako i tima s predstavnicima korisnika ili samim korisnicima.

2.5.1. Vidljivi integritet

Vidljivi integritet je u očima korisnika. Cilj je zadovoljiti sve korisnikove zahtjeve kako bi on bio zadovoljan proizvodom. Da bi se to moglo zadovoljiti stalno pred očima moraju biti korisnički zahtjevi. Problem kod klasičnog načina izrade programskih rješenja je taj što taj model nije prilagodljiv što se tiče promjena. Čak ni sam korisnik na početku ne znam točno što želi, kao i sve drugo vrijeme je faktor koje utječe na njegovu percepciju i potrebe. Najveći je problem što dolazi do gubitka informacija kada krenu od korisnika do trenutka kada stignu do programera. To je put na kojem su zahtjevi korisnika morali proći više koraka kako bi stigli do onoga tko će ih realizirati kôdom. To je zahtijevalo neko određeno vrijeme a dok se to odvijalo zahtjevi su se već možda promijenili. Pa kako onda riješiti ovakve probleme? Odgovor daje *lean* način razmišljanja – uspostaviti kvalitetnu komunikaciju između korisnika i programera. To će se postići na način da se koriste kratke iteracije te nakon svake od njih može se provjeriti zadovoljstvo korisnika isporučenim. Ukoliko su potrebne promjene mogu se odmah napraviti. Koristiti modele i rječnik komunikacije kojeg razumiju i korisnik i programeri. Za kompleksnije sustave koristiti usluge glavnog programera koji ima iskustvo i znanje te je sposoban pravilno prenijeti korisnikove potrebe dizajnerima

[2] Poppendieck, 2003.

sustava i programerima. Razviti testove na kraju svake iteracije koji će korisniku kvalitetnim primjerima pokazati kako sustav radi. Uspješnost rada programera vidi se u korisnikovom shvaćanju i prihvaćanju prezentiranog sadržaja.

Vremenom se sve mijenja pa tako i korisnikova percepcija proizvoda. Da bi se osiguralo vidljivi integritet kroz dulji period mora se samim dizajnom omogućiti jednostavnu implementaciju traženih promjena. Iskustva pokazuje da izrada dokumentacije samo u svrhu da je se posjeduje nije pravi put već je u stvari otpad. Pravi put je učiniti programere odgovornima za buduće promjene. Dobra komunikacija i prijenos važnih činjenica o sistemu između programera koji su stvorili proizvod i onih koji će ga održavati je ključna. Od velike pomoći nam je i izrada automatiziranih testova koji će nam provjeravati da li je došlo do narušavanja već postojećeg kôda novo napravljenim.

2.5.2. Sistemski integritet

Sistemski integritet podrazumijeva rad svih dijelova proizvoda kao cjeline. Rad svih komponenti je uravnotežen i odlikuje ga fleksibilnost, održivost i učinkovitost. Stvorena arhitektura sustava strukturirana je tako da ostvaruje sve značajke i mogućnosti koje je korisnik zahtijevao. Osnova tako izgrađene arhitekture sustava je kvalitetna dvosmjerna komunikacija među svim sudionicima.

Za osiguranje sistemskog integriteta potrebno je usvojiti određene prakse u radu. Prva se odnosi na korištenje već postojećih dijelova kôda što umanjuje probleme s kompatibilnošću te smanjuje količinu potrebne komunikacije. Primjer bi bio da se ne pokušava stvoriti vlastiti format za prijenos podataka već koristiti postojeće kao što je XML. Druga se odnosi na integrirano rješavanje problema kojom se omogućava kvalitetan protok tehničkih informacija unutar organizacije. Integrirano rješavanje problema podrazumijeva shvaćanje i rješavanje problema u istoj iteraciji, dvosmjerni protok informacija u trenutku kada ih se dobije, izmjenu informacija u manjim cjelinama i to razgovorom u nepisanom obliku.

Kako bi se arhitektura proizvoda postavila na prave temelje potrebno koristiti arhitekturu razgranatu po slojevima. Slojevi su sami za sebe neovisni. Primarni slojevi su: prezentacijski, sloj poslovne logike i podatkovni sloj. Važno je napomenuti da podatkovni sloj ne mora znati kako funkcionira sloj poslovne logike, niti sloj poslovne logike ne mora znati kako funkcionira prezentacijski sloj. Pošto je bitno osigurati i vidljivi integritet posebnu pažnju je potrebno posvetiti izradi prezentacijskog sloja jer je on taj kojeg korisnik vidi.

Glavna uloga arhitekture je osigurati poslovnu i tehničku promjenu sustava tako da bude financijski isplativa. To se postiže tako da se razdvoji sustav po slojevima. Promjene u jednom sloju lokalno utječu samo na njega a ne sustav u cjelini. Zato je vrlo bitno na početku postaviti prave temelje a kako proces izrade napreduje pustiti da ostatak ispliva sam od sebe. Osiguranje da će se dobiti prave informacije u kojem smjeru krenuti u danom trenutku osigurava korištenje već prije navedene prakse: korištenje već gotovih rješenja i integrirano rješavanje problema. Važno je imati i osobe s znanjem i iskustvom na kritičnim mjestima u sustavu. Potrebno je njihovo poznavanje kompleksnosti sustava i niza rješenja na koji način rješavati nadolazeće probleme. To je uloga za glavnog programera. On usmjerava timove da zajedno donose odluke kako bi u konačnici zadovoljili korisnika.

2.5.3. **Refaktoriranje**

Činjenica je da je gotovo nemoguće u početku napraviti arhitekturu kompletnog sustava i očekivati da se neće mijenjati tijekom vremena. Sve se mijenja tijekom vremena a što je sustav kompleksniji to je potreba za promjenama veća. U *lean* načinu razmišljanja nitko ne očekuje da će sve biti savršeno već na početku. Potrebom za promjenama tijekom vremena dolazi se do pojma refaktoriranje što znači poboljšanje arhitekture sustava kako sustav evoluira.

Arhitektura sustava ima određene karakteristike koje se mogu pratiti i na temelju njihove promjene reagirati u pravo vrijeme refaktoriranjem kako bi se spriječio gubitak sistemskog integriteta:

- Jednostavnost kôda – najjednostavniji dizajn je i najefikasniji
- Jasnoća napisanog kôda – kôd mora biti razumljiv da bi ga bilo tko mogao koristiti u budućnosti
- Pogodnost za korištenje – ako se pokaže da neki dijelovi ne zadovoljavaju zahtjeve potrebno ih je popraviti makar to značilo i promjenu arhitekture sustava
- Bez ponavljanja kôda – svako ponavljanje kôda povećava mogućnost mjesta na kojima je moguća pogreška a na taj način smanjuje fleksibilnost
- Bez dodatnih značajki – ne raditi dodatne značajke osim ako ih korisnik ne traži, svako predviđanje što bi korisnik mogao željeti je otpad

Svaka promjena kôda nije refaktoriranje, refaktoriranje mora imati svrhu i količina potrebnog refaktoriranja ovisi o samoj arhitekturi i zahtjevima koje treba riješiti.

Nameće se pitanje da li je refaktoriranje dodatan posao. U *lean* načinu razmišljanja svakako nije jer kvalitetna arhitektura sustava nastaje tijekom cijelog procesa proizvodnje. Dapače

ono je potreba koja osigurava kvalitetu i postojanost arhitekture aplikacije. Što se događa ako se odgađa s refaktoriranjem? U početku se dobiva na vremenu jer se može krenuti dalje s radom ali i konačnici dovodi do problema. Jednostavnije i jeftinije je problem odmah riješiti jer ako se to ne učini vrlo je velika mogućnost da će neki novo napravljeni dijelovi kôda koristiti problematični dio. Kao što se u ranijim poglavljima govorilo protokom vremena eksponencijalno se povećava trošak rješavanja problema. Refaktoriranje je alat koji pruža da proizvod bude spreman za promjene koje će nastati u procesu njegove izrade ali i u cijelom njegovom životnom vijeku.

2.5.4. **Testiranje**

Alat koji omogućava refaktoriranje i upozorava da postoji potreba za istim je testiranje. Glavna uloga testova je dobiti potvrdu da proizvod zadovoljava korisnikove potrebe. No testovi se ne provode samo na kraju razvoja proizvoda već moraju postati dio kôda u tijeku same izrade. Da bi se mogli primjenjivati postulati *lean* načina razmišljanja jedna od najvažnijih stvari na koju se mora osigurati su povratne informacije a testovi su idealan način za to. Sve se svodi da dobivanje pravih informacija u pravo vrijeme da bi se umanjili troškovi vezani za popravak grešaka. Osiguranje da će sve raditi i nakon promjena koje su urađene je neprocjenjivo a testovi nam pružaju baš to. Testovi su i dobar način da se zamijeni silna dokumentacija za proizvod.

Vrste testova su:

1) testovi manjih zasebnih jedinica (*unit tests*) - testira se funkcionalnost pojedinog modula
2) testovi sustava (*system tests*) – testira se da li cijeli sustav radi kako je zamišljen
3) testovi integracije (*integration tests*) – testira se integriranje pojedinih dijelova kôda u cjelinu
4) testovi za korisnike (*customer tests*) – testira se zadovoljstvo korisnika ponuđenim rješenjima

Prve tri kategorije spadaju pod programerske testove (*developer tests*) jer služe onima koji pišu kôd da provjere da li ono što su napravili zadovoljava funkcionalnosti koje su zadane. Zadnja kategorija je vezana uz korisnike i tek ti testovi daje konačnu potvrdu da se napravilo točno ono što je korisnik želio. Testovi se mogu obavljati ručno ili automatizirano. Automatizirani testovi pružaju veću sigurnost jer postoji garancija da će biti provedeni.

Testovi su vrlo korisni prilikom nadogradnji proizvoda kada je već isporučen korisniku i u funkciji. Održavanje proizvoda nije nimalo lak zadatak. Mora se osigurati da proizvod i dalje pruža funkcije koje je do tada pružao te da nove funkcionalnosti ne naruše arhitekturu

sustava. *Scaffolding* je pomoćna struktura koja nam to omogućava. Setom automatiziranih testova za programere i korisnike simulira se integracija novih značajki u postojeći proizvod na način da pruža podatke o posljedicama koje će se pojaviti ako se implementira to novo rješenje. Ovdje dolazi do izražaja isporučivati kôd u manjim serijama kako ne bi se previše vremena trošilo na trajanje testova i čekanje na rezultate. Na manjoj količini kôda uvijek će biti i manji broj grešaka i mogućih nesukladnosti koje će se brže i lakše ispraviti.

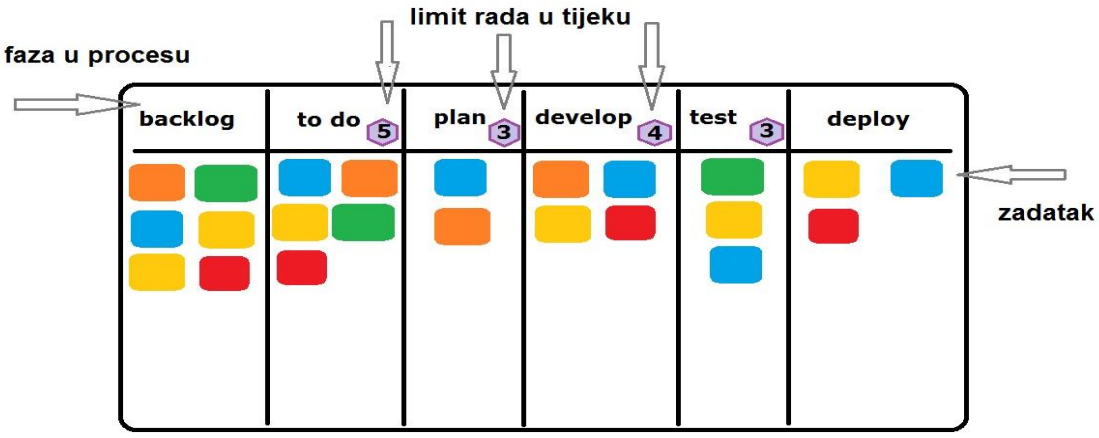
Mnogi će reći da je pisanje testova gubitak vremena i novca ali u stvari nije tako. Vrlo je lako moguće da na kraju razvoja proizvoda količine kôda za proizvod i testove budu jednake. No taj uloženi trud i vrijeme u izradu testova višestruko se vraća i smanjuje sve buduće intervencije na kôdu uvijek osiguravajući da će uz minimalno truda sustav i dalje funkcionirati kako bi korisnik dobio sve ono što je želio.

3. Samoposlužni sustavi zadataka (pull sistem)

Kako se maksimizirala produktivnost unutar tima potrebno je dobro organizirati vrijeme provedeno na radnom mjestu. Postoje dvije mogućnosti za to. Prva je da se ljudima organizira vrijeme s više točke upravljanja. Druga je da se postave neka pravila prema kojima zaposlenici sami mogu zaključiti što treba raditi u pojedinim situacijama. Prvi pristup podrazumijeva raspored kojim se gura posao prema završetku dok za drugi pristup vrijedi da se povlači onoliko posla koliko je u tome trenutku moguće. Taj drugi način naziva se samoposlužni sustav zadataka a njegova glavna prednost je skraćivanje vremena donošenja i prenošenja odluka u zapovjednom lancu.

Samoposlužni sustavi zadataka koriste mehanizam za organizaciju rada zvan kanban. Kanban je metoda raspoređivanja posla koja promjene i varijacije uzima u obzir na kraju radnog procesa što umanjuje stres od promjena kod zaposlenika. Tako je dobiven rad koji je uređen i ima svoj tijek bez potrebe koordinacije od strane upravljačkih struktura. Vječno je pitanje upravljačkih struktura da li je vrijeme rada zaposlenika maksimalno iskorišteno. Niti jedan raspored ne može biti potpuno učinkovit jer oni koji upravljaju imaju drugu viziju i druga iskustva od onih koji ga izvode. U tome smislu kanban pruža optimalan pristup koji omogućava zaposlenicima da sami sebi biraju koji će dio posla odraditi ovisno o svojim sposobnostima i mogućnostima. Pokazalo se da je motivacija zaposlenika na puno većoj razini prilikom korištenja ovog mehanizama organizacije posla.

Pojasniti će se koncept kanban mehanizma. Potrebno je rastaviti proces proizvodnje na faze. Za vizualizaciju nam služi kanban ploča (Slika 3.1).



Slika 3.1 Primjer kanban ploče

Kanban ploča je snažan alat koji nam služi u dvije svrhe: vizualizira proces i limitira količinu rada u tijeku po fazama. Zadatke se upisuje i pomiče ovisno o tome da li je rad na njima obavljen kako bi prebili prebačeni u drugu fazu ili bili vraćeni natrag u prethodnu. Oznaka limita rada u tijeku govori koliko pojedinih zadataka se može odrađivati u pojedinoj fazi.

Neke od faza mogu biti:

- *backlog* – skup svih korisnikovih zahtjeva
- *to do* – odabir korisnikovih zahtjeva koje će se odrađivati
- *plan* – planiranje kako odraditi određeni zadatak
- *develop* – razvoj zadatka, izrada kôda
- *test* – testiranje napravljenog kôda
- *deploy* – implementacija i puštanje u produkciju odrađenog zadataka.

Zahtjeve koje se postavlja na kanban ploču moraju imati određenu struktura da bi se moglo s njima pravilno upravljati (Slika 3.2).

Slika 3.2 Primjer strukture zadatka s kanban ploče

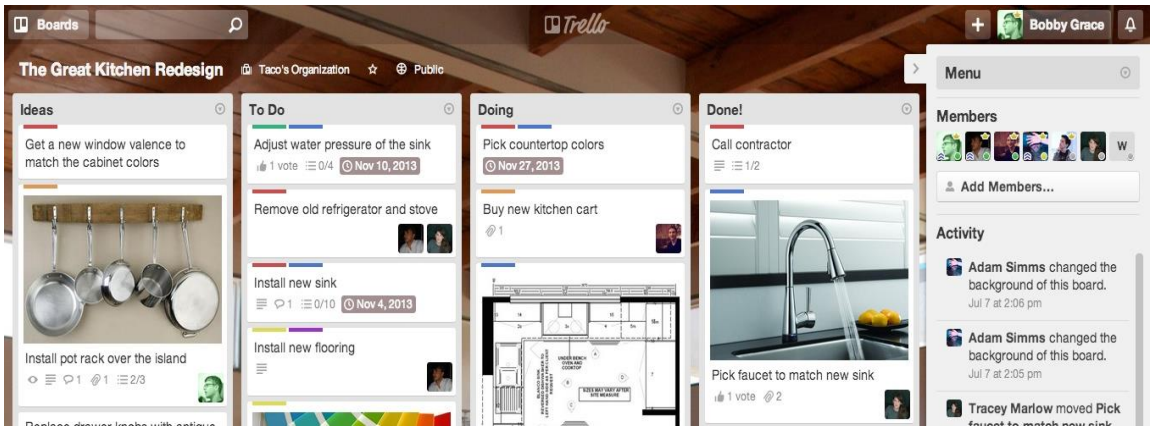
Svaki zahtjev mora sadržavati slijedeće:

- oznaku koja jedinstveno označava pojedini zahtjev
- naziv zadatka
- procjenu trajanja izvršenja zadatka
- popis zaposlenika koje rade na zadatku
- nekakvu oznaku koja ukazuje na odgodu ili zabranu rada na zadatku

Kanban mehanizmi uspješno upravljaju tijekom procesa proizvodnje, a s time i implementacijom promjena. Unaprijed određenim jasnim pravilima osigurava se protok zahtjeva kroz faze. Dobivene povratne informacije vrlo brzo i jednostavno se implementiraju u tijeku procesa.

3.1. Trello

Trello je alat koji nudi razne opcije pri organizaciji projekta. Nije isključivo vezan uz programske projekte već je primjenjiv i na ostale svakodnevne zadatke koji imaju potrebu za organizacijom i planiranjem. Glavne karakteristike su jednostavnost uporabe i prilagodljivost. Koristiti ga se može na raznim uređajima: Apple i Android uređaji, tableti, pametni satovi i na računalima. Sastoji se od glavne ploče s popisom grupa zadataka (Slika 3.3).



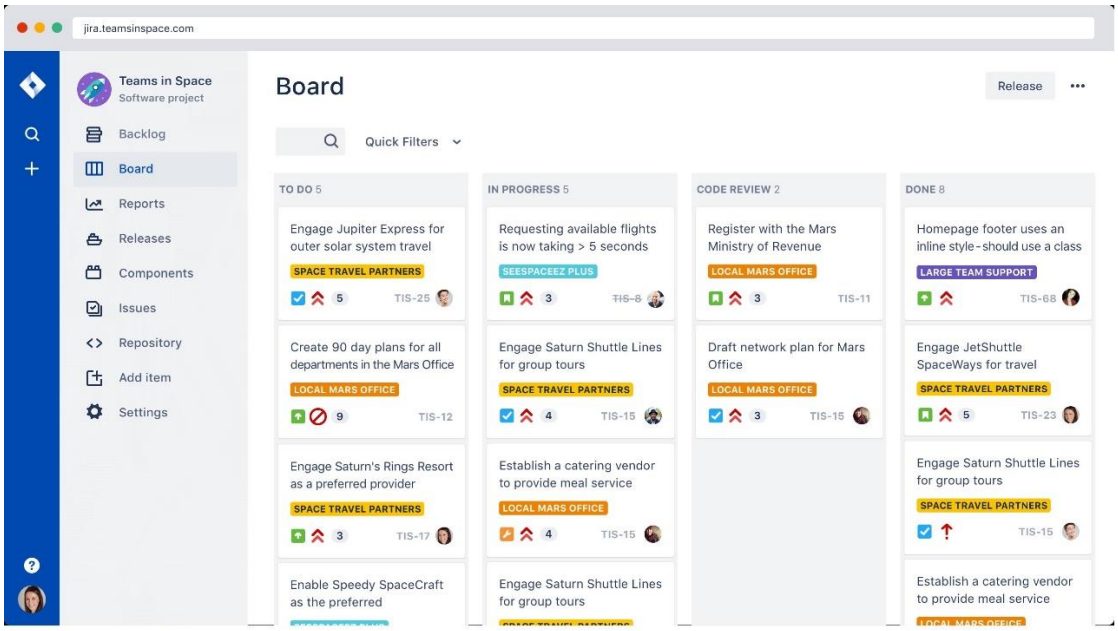
Slika 3.3 Glavna ploča Trello[3]

Svaki zadatak je moguće pomicati među grupama, uređivati i brisati postojeće te dodavati nove zadatke. Glavne značajke zadataka koje nudi su: komentari, rokovi završetka, prilozi, konverzacije, liste za odabir i drugo. Omogućava jednostavno korištenje za neograničen broj korisnika a posebno se ističe funkcionalnost koja nudi konverzaciju između istih. Ukoliko se u konverzaciji spominje neki korisnik on će biti obavješten o promjeni statusa. Napredan sustav nudi obavještavanje na više različitih konzola. Tako je moguće podesiti da se obavijesti primaju unutar same aplikacije, na e-mail, kao desktop obavijesti putem web preglednika, na mobilne uređaje. Postoji i mogućnost filtriranja svih podataka čime je traženje pojedinih podataka uvelike olakšano. U osnovnoj verziji nudi se jedna nadogradnja koja može biti specijalizirana za pojedine zahtjeve, npr. za potrebe programskih projekata. Nudi se i profesionalno rješenje koje omogućava ukupno tri nadogradnje ali njega se dodatno naplaćuje. Za programere postoji i besplatan API koji omogućava da se Trello implementira u vlastite projekte.

[3] Izvor: https://trello.com/tour

3.2. Jira Software

Jira Software je jedan od najkorištenijih alata za organizaciju procesa. Više je prilagođen procesima za izradu programskih rješenja od alata Trello. Dostupan je raznim platformama: Apple i Android uređaji, tableti, računala. Prednosti pred konkurencijom su: fleksibilno planiranje, precizan izračun procjena trajanja i troškova, tijek i izrada projekata bazirana na važnosti zadataka, svima vidljiva trenutna sinkronizacija podataka, pravovremeni i kvalitetni izvještaji, podrška za promjene kako projekti rastu. Svoje glavne karakteristike i funkcionalnosti raspodjeljuje na četiri dijela. Prva od njih je planiranje. Nude se opcije stvaranja korisničkih zahtjeva i problema, planiranja iteracija i raspodjelu zadataka po grupama. Ostvarenje zadatka ovisi o prolasku kroz grupe zadataka koje su: za napraviti, u tijeku izrade, testiranje, završeno. Druga je praćenje projekta. Ističe praćenje zadataka po njihovoj važnosti i nudi mogućnost diskusije između svih uključenih korisnika. Naglasak je na transparentnosti i jednostavnoj vizualizaciji koju pruža korisničko sučelje (Slika 3.4).



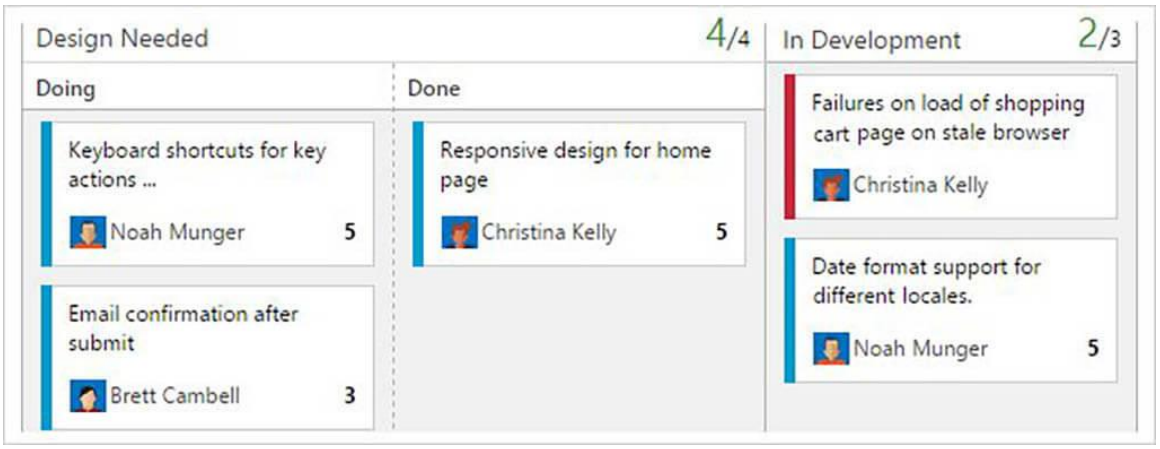
Slika 3.4 Glavna ploča Jira Software[4]

Treći dio je posvećen izdanjima rješenja. Pruža se pregled svih dosadašnjih izdanja rješenja te njihove statuse. Četvrti dio se odnosi na izvještaje koji pomažu u optimizaciji cjelokupnog proizvodnog procesa. Pravovremeni grafički izvještaji pomažu odgovornima za upravljanje procesom da donesu prave zaključke za trenutni ali i za buduće projekte.

[4] Izvor: https://www.atlassian.com/software/jira

3.3. Team Foundation Server

Team Foundation Server je alat za razvoj programskih rješenja prilagođen svim programskim jezicima i platformama. Sastoji se od skupa alata za razvoj softvera koji se integriraju na vlastitim platformama i na taj način omogućavaju timu da djeluje učinkovito na programerskim projektima svih veličina. Glavna funkcionalnost je praćenje promjena u kôdu i sinkronizacija kod svih korisnika uključenih u projekt. Posjeduje vlastito rješenje za tu funkcionalnost. Temeljem zahtjeva koje korisnik zadaje pružaju se najnovije verzije kôda. Na taj način se osigurava da nema preklapanja kôda u projektu. Nude se i opcije zaštite i kontrole kôda koje osiguravaju sigurnost u radu. Alat je orijentiran isključivo za izradu programskih projekata. Izrade prilagođenih kanban ploča su u skladu s agilnim razvojem aplikacija (Slika 3.5).



Slika 3.5 Kanban ploča Team Foundation Server[5]

Sve značajke kanban ploče su zastupljene, od prikupljanja zahtjeva od naručitelja do praćenja tijeka razvojnog procesa. Korisniku pruža kvalitetnu navigaciju i kontrolu nad alatom što značajno pridonosi preglednosti, praćenju i izradi izvještaja. Ovaj alat nudi i kontinuiranu integraciju rješenja. Rješenje prolazi kroz slijedeće faze: izrada, stvaranje paketa, testiranje, izdavanje i ponavljanje. Svi ovi dijelovi procesa mogu se podesiti da se izvode ručno ili automatski. Kontinuirana integracija nudi mogućnost ranog otkrivanja problema u programskom rješenju te njegov popravak dok nije prešao u drugu fazu. Na taj način se značajno smanjuju troškovi izrade rješenja.

[5] Izvor: https://visualstudio.microsoft.com/tfs/

4. Analiza arhitekture i odabira tehnologija

Proučavanjem i analizom već postojećih rješenja na tržištu za samoposlužne sustave zadataka došlo se do zaključka da je najbolje koristiti troslojnu arhitekturu aplikacije. Troslojna arhitektura garantira neovisnost svakog dijela ali i omogućuje njihovo povezivanje u jednu cjelinu. Rješenje se sastoji od: sloja spremišta podataka, poslovne logike i prezentacijskog sloja. Ovakva podjela omogućava brzu i efikasnu promjenu unutar određenog sloja a da drugi slojevi tu promjenu ne osjete, točnije da su potrebne minimalne promjene kako bi cjelokupno rješenje i dalje izvršavalo svoju svrhu.

Analizom arhitekture došlo se do zaključka koje tehnologije koristiti pri izradi aplikacije. Za sloj spremišta podataka odabran je Microsoft SQL Server 2014. Izrada sloja poslovne logike i prezentacijski sloja povjerena je alatu Microsoft Visual Studio 2015 u kojem će se koristiti ASP.NET MVC tehnologija.

4.1. Analiza arhitekture aplikacije

4.1.1. SQL baza podataka – sloj spremišta podataka

SQL baza podataka je jedna od najkorištenijih spremišta podataka. Svojim funkcionalnostima savršeno odgovara za izradu programskog rješenja kojeg ovaj rad prezentira. U stvarnosti postoje entiteti, cilj je pretvoriti entitete u podatke kako bi se isti mogli koristiti za rad nad njima. Svaki entitet ime atribute i vrijednosti tih atributa. Entitet, atribut i vrijednost atributa su vezani i na način da čine cjelinu. Da bi se takvi entiteti mogli pretvoriti u podatke koriste se relacijski modeli u bazama podataka. To je ujedno i najčešće korišteni model prikaza podataka u bazama podataka. Podaci se prikazuju u tablicama. Za rad se koristi SQL jezik.

4.1.2. Model i Controller - sloj poslovne logike

ASP.NET MVC tehnologija omogućava podjelu zadataka na tri glave skupine. Kratica MVC odnosi se na Model-View(pogled)-Controller(kontroler). Model je zamišljena klasa koja predstavlja podatak iz spremišta podataka. Kontroler je dio koji je zadužen za obradu i pripremu tih podataka za View – prezentacijski sloj. MVC tehnologija omogućuje veću kontrolu pri izradi i uvelike olakšava održavanje.

Kontroler se sastoji od niza javnih akcijskih metoda koje se poziva putem URL-ova. Akcijske metode vraćaju instancu klase *ActionResult*. Podaci koji se dobiju mogu biti: podaci bilo kojeg tipa, preusmjeravanja na druge URL-ove ili akcijske metode, iscrtavanje pogleda i dijela pogleda.

4.1.3. View – prezentacijski sloj

Za prezentacijski sloj u MVC arhitekturi predviđen je View(pogled). Pogledi se koriste na slijedeći način: nakon što akcijska metoda pronađe pogled predaje mu model kojeg pogled pretvara u HTML, CSS i JavaScript. Zadatak pogleda je prikazati model dok je strogo zabranjena bilo kakva obrada podataka ili komunikacija s bazom podataka unutar njega. Pogled može sadržavati HTML, CSS i JavaScript ali i dijelove programskog kôda koji služe isključivo za prikaz modela. Razor mehanizam pogleda služi za generiranje izlaznih podataka u obliku HTML-a, to je poveznica između C# kôda i HTML-a.

4.2. Analiza odabira tehnologija kod izrade aplikacije

4.2.1. Microsoft SQL Server 2014

Microsoft SQL Server je relacijska baza podataka koja se koristi T-SQL jezikom za upite. To znači da su omogućeni klasični upiti (SELECT naredba) kao i napredni upiti (IF naredba i slično). Glavna tehnologija za pohranu i obradu podataka je Database engine. Za pristup bazi podataka i obradu podataka korišten je SQL Server 2014 Managment Studio.

4.2.2. Microsoft Visual Studio 2015

Microsoft Visual Studio je interaktivno razvojno rješenje (IDE) koje se koristi za izradu programskih rješenja. Za izradu praktičnog dijela ovog rada korištena je mogućnost izrade WEB stranica. Ovaj alat podržava pregled, unos i ispravljanje kôda, otklanjanje neispravnosti u kôdu i izgradnju rješenja. Praktični dio rada izrađen je upravo ovim alatom zbog njegove jednostavnosti i brzine uporabe. Najbolji primjer je automatsko ispravljanje grešaka kod pisanja kôda. Na računalu na kojem je praktični rad izrađen kao najstabilnija i najbrža pokazala se verzija Microsoft Visual Studio 2015.

4.2.3. **ASP.NET – MVC**

ASP.NET je okvir (engl. *framework*) za razvoj web aplikacija koji se koristi HTML, CSS i JavaScript programskim jezicima. Programski okvir korišten za izradu praktičnog dijela ovog rada je ASP.NET MVC. ASP.NET MVC temelji se na odvajanju triju slojeva: modela, pogleda i kontrolera. Povezivanje modela, pogleda i kontrolera koristeći ugovore temeljene na sučelju, omogućuje testiranje svake komponente neovisno. Ovakva podjela odgovornosti na slojeve znači da svaki sloj ima vlastitu odgovornost. Kada stigne zahtjev na komponentu usmjeravanja on se usmjerava na točno određeni kontroler. Zadaća kontrolera je pribaviti odgovarajući model te stvoriti instancu modela prema podacima koje pribavlja komunikacijom s bazom podataka. Nakon obrade tih podataka prema odgovarajućem modela priprema podatke i prosljeđuje ih pogledu. Pogled služi kao prezentacijski sloj koji prikazuje model nakon što se odradi zahtjev za primanjem podataka na klijentskoj strani.

ASP.NET podržava još dva programska okvira za razvoj web aplikacija: Web Forms i ASP.NET Web Pages.

5. Praktično ostvarenje aplikacije samoposlužnih sustava zadataka

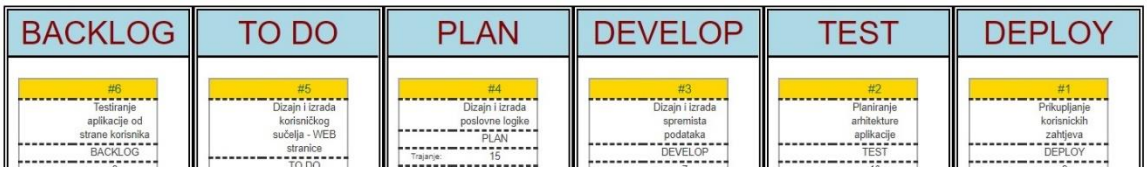
Cilj ovoga poglavlja je pojasniti svrhu aplikacije i način na koji je izrađena programskim alatima. Objasniti će se na koji način su se primjenjivale l*ean* metode te svaki pojedini dio njene troslojne arhitekture.

5.1. Svrha aplikacije

Naručitelj je izrazio želju za posjedovanjem jednostavne aplikacije praćenja rada na izradi budućih programskih rješenja. Analizom vlastitog proizvodnog procesa došao je do zaključka da se ne upravlja racionalno resursima prilikom izrade aplikacija. Postojeća rješenja na tržištu nisu zadovoljavala kriterije, naručitelju su bila previše kompliciranа za integraciju u postojeću organizaciju rada. Temeljem tih korisničkih zahtjeva preuzima se obaveza za izradom aplikacije koja će iste zadovoljiti.

Svrha ili cilj ove aplikacije je pokazati kako unaprijediti upravljanje jednim od najvažnijih resursa – vremenom izrade aplikacije. Samim time se povlači paralela s upravljanjem drugim važnim resursima, na primjer financijskim sredstvima koja su uložena u cjelokupni projekt. Ukoliko se racionalno koristi vremenom financijski troškovi će na kraju biti manji. Nije upravljanje vremenom jedini način kojim se osigurava da se cijeli projekt završi uspješno ali je svakako jedan od važnijih.

Upravljanje tijekom izvođenja pojedinih zadataka u aplikaciji je odrađeno na način da se prati njihov napredak kroz 6 faza u životnom vijeku zadatka. To su *backlog, to do, plan, develop, test* i *deploy* (Slika 5.1).



Slika 5.1 Faze životnog vijeka zadatka

Koristeći ovaj model praćenja zadataka dobiva se potpuna kontrola nad resursima. Na izvještaju koji se dobije može se kontrolirati zauzetost resursa po više kategorija. Detaljnim

pregledom istog može se dodatno provjeriti da li u sustavu postoje zastoji te ako postoje raspodijeliti zadatke i timove na odgovarajući način.

Kako bi naručitelj mogao pregledavati i testirati aplikaciju po fazama njenog razvoja i davati povratne informacije odlučeno je da će verzije aplikacije biti postavljane lokalno na jednom računalu kod naručitelja. Visual Studio pruža mogućnost izrade produkcijske verzije web aplikacije. Prilikom završetka rada u iteraciji izradila se produkcijska verzija – *release* – koja se zatim dostavljala naručitelju. Na računalu naručitelja postavila se verzija na IIS Manager koji omogućava da se na nekom unaprijed navedenom portu lokalno pokreće (*localhost*) aplikacija u web pregledniku. Za potrebe spremišta podataka instalirana je besplatna verzija SQL Managment Studio-a na kojoj se samo pokrenula SQL skripta koja je stvorila bazu podataka s inicijalnim vrijednostima u svakoj od stvorenih tablica.

5.2. Analiza aplikacije po korištenim *Lean* metodama

Samoposlužni sustavi zadataka su jedan od alata koji nam osiguravaju da nam projekt bude uspješan, točnije da se izvuče maksimum kako zadovoljstva naručitelja tako i zadovoljstva onoga koji ga je izradio. Kao što je prije napisano, nije garancija uspjeha koristiti sve *lean* metode u svome poslovanju već se prilagoditi i koristiti samo neke kako bi se napredovalo u cjelini. Tako se i u izradi ovoga projekta koristilo samo neke od nabrojanih *lean* metoda, i to one koje su nam se činile prikladnima.

5.2.1. Prepoznavanje otpada

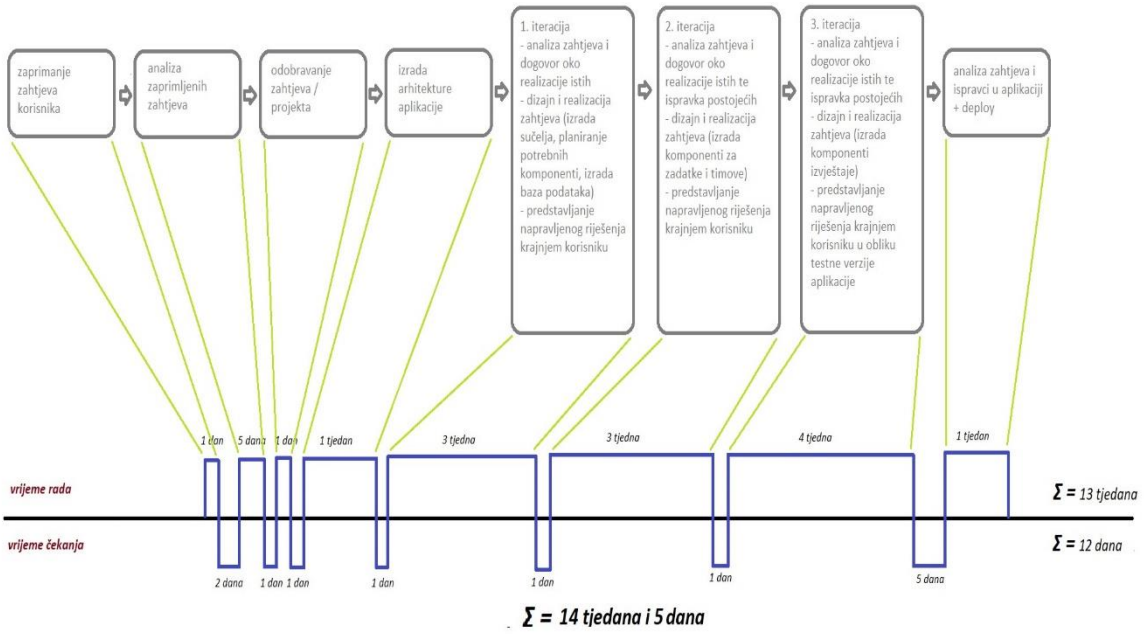
Prepoznavanje otpada u proizvodnom procesu aplikacije Samoposlužnih sustava zadataka svodi se na uočavanje nepotrebnih komponenti proizvodnog procesa koje oduzimaju resurse a nisu bitne za rad aplikacije ili nisu zadovoljile krajnjeg korisnika.

Najbolji primjer u vlastitom slučaju je komunikacija s krajnjim korisnikom. U prvim kontaktima s korisnikom dalo se naslutiti da je više osoba sa strane korisnika uključeno u proces odlučivanja funkcionalnosti aplikacije. Vrijeme odluke od strane korisnika je bilo prilično dugo što je usporavalo cijeli proces. Kako bi se umanjio taj utjecaj dogovoreno je da će se komunikacija između krajnjeg korisnika i izvođača odvijati preko samo dvije osobe. U korisničkom timu određena je osoba koja je zadužena za komunikaciju i usklađivanje svih zahtjeva te prijenos istih voditelju projekta sa strane izvođača. Time se postigla direktna veza, smanjilo se vrijeme odlučivanja i ubrzalo se cijeli proces.

Kvalitetno izrađena shema proizvodnog procesa i komunikacija s krajnjim korisnikom donijela je značajnu uštedu i priliku da se poboljšaju neke druge komponente a ne da se bavi stvarima koje nisu korisne krajnjem korisniku i ne dodaju vrijednost proizvodu.

5.2.2. Izrada sheme proizvodnog procesa

Prije same izrade aplikacije bilo je potrebno uskladiti želje krajnjeg korisnika o funkcionalnostima i izgledu aplikacije. Za kvalitetnu izradu aplikacije potrebno je bilo isplanirati tijek razvoja iste kako bi se odredilo koliko resursa je potrebno da bi se istu izradilo. Izrađena je shema proizvodnog procesa u skladu s iterativnim razvojem aplikacije kako bi se u potpunosti imala kontrola nad proizvodnim procesom. Slika 5.2 Shema proizvodnog procesaprikazuje konačnu verziju sheme proizvodnog procesa.



Slika 5.2 Shema proizvodnog procesa

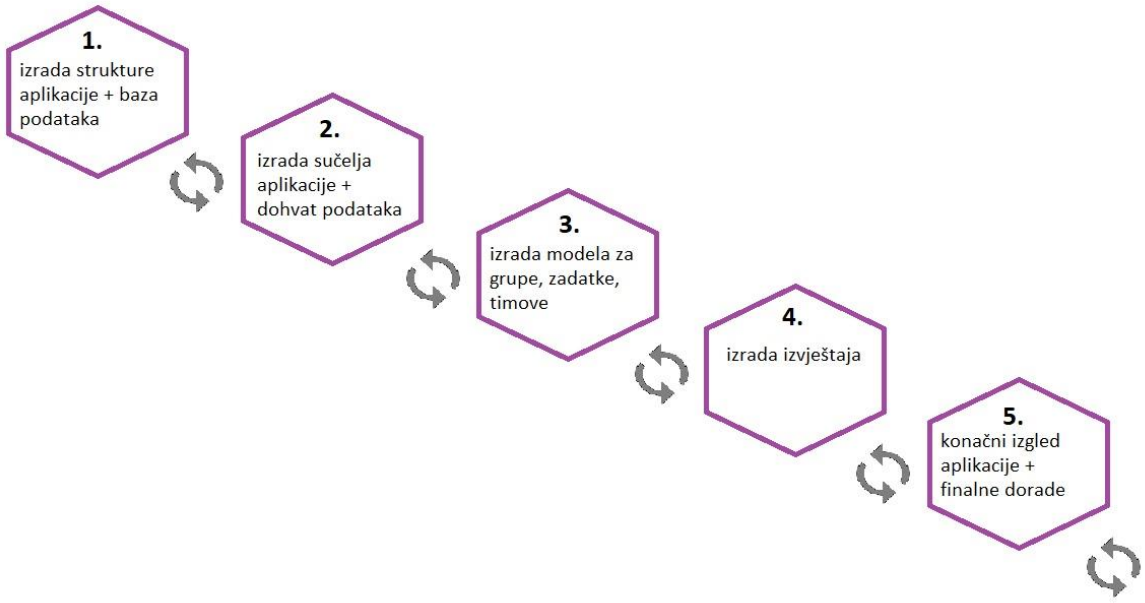
Shema proizvodnog procesa zamišljena je u skladu s agilnim razvojem. Skraćeno je maksimalno vrijeme odluke korisnika i s time se postigla bolja dinamika - više vremena za rad a manje vremena u iščekivanju povratnih informacija. Za razliku od tradicionalnog razmišljanja daje se krajnjem korisniku mogućnost provjera i izmjena funkcionalnosti u samom tijeku izrade aplikacije što smanjuje rizik od neuspjeha s obje strane.

Sam proizvodni proces se odvija u fazama. Krenulo se s zaprimanjem zahtjeva korisnika te analizom istih. Na temelju analize odobravani su zahtjevi i u konačnici cijeli projekt. Potom slijedi izrada arhitekture aplikacije koja će biti temelj na kojem će se izgraditi fleksibilna

aplikacija u potpunosti prilagođenu korisničkim zahtjevima. Zatim slijedi iterativni razvoj aplikacije. Na kraju svake iteracije krajnjem korisniku predstavlja se urađeno i traži se od njega povratne informacije kako bi se izvršili eventualne preinake na početku slijedeće iteracije. Kada se završe sve iteracije predstavlja se korisniku testna verzija aplikacije koju će pregledati i na kraju dati konačne informacije za završne izmjene. Nakon tih posljednjih ispravaka aplikacija se pušta u produkciju.

5.2.3. **Iteracije**

Kako bi se maksimalno iskoristilo resurse koji stoje na raspolaganju i zadovoljili se svi korisnički zahtjevi korištene su iteracije u razvoju programskog rješenja. Podjelu rada u iteracije prikazuje Slika 5.3 Pregled iteracija.



Slika 5.3 Pregled iteracija

Cilj je bio obuhvatiti jednom iteracijom smislene, povezane dijelove aplikacije kako bi se izradom istih završila cjelina koja bi nakon toga bila prezentirana korisniku. Na temelju urađenog korisnik pruža povratne informacije o svojem zadovoljstvu učinjenim. Ukoliko je korisnik zadovoljan prelazi se na novu iteraciju. Ukoliko nije razmatra se na temelju povratnih informacija da li treba ponoviti cijelu iteraciju ili je navedene promjene moguće uraditi na samom početku slijedeće iteracije. Proces se ponavlja sve dok korisnik nije u potpunosti zadovoljan proizvodom.

Korištenjem iteracija dobiva se mogućnost pravovremene reakcije na potrebe korisnika. Na taj način se postiže ušteda u resursima jer se u realnom vremenu reagira na korisničke zahtjeve koji se uvijek mijenjaju tijekom proizvodnog procesa aplikacije. Bili ti zahtjevi veći ili manji lakše će se prema istima prilagoditi aplikacija ukoliko se korisniku prezentiraju rješenja tijekom samog procesa izrade nego da mu se samo prezentiran konačni proizvod na kojem onda treba vršiti izmjene. Rješavanje poteškoća tijekom samog razvoja aplikacije uvelike pridonosi isporuci kvalitetnog proizvoda i zadanim vremenskim okvirima.

5.2.4. **Refaktoriranje**

Razvojem aplikacije neizbježno dolazi do promjena koje nije moguće predvidjeti prilikom izrade arhitekture i dizajna aplikacije. Kako bi se osiguralo stabilan rad i jednostavno održavanje aplikacije potrebno je bilo izvršiti refaktoriranje na pojedinim dijelovima. Samu strukturu aplikacije nije bilo potrebno mijenjati jer se odlučilo koristiti MVC strukturu aplikacije koja je fleksibilna ali i dovoljno jednostavna da omogućava maksimalne performanse aplikaciji. Prilikom izrade smisleno su se nazivale metode, varijable, modeli i ostale komponente kako bi se omogućilo jednostavnije korištenje i izmjena u budućnosti. Kôd 5.1 primjer jedne tako napisane metode.

```
public static List<SelectListItem> DohvatiTeamove()
{
    List<SelectListItem> kolekcija = new List<SelectListItem>();
    ds = SqlHelper.ExecuteDataset(cs, "DohvatiTeamove");
    foreach (DataRow row in ds.Tables[0].Rows)
    {
        kolekcija.Add(new SelectListItem
        {
            Value = row["IDTeam"].ToString(),
            Text = row["Naziv"].ToString()
        });
    }
    return kolekcija;
}
```

Kôd 5.1 Metoda za dohvat podataka o timovima

Aplikacija je napravljena bez ponavljanja koda, korištene su metode koje se pozivaju na više mjesta unutar programa kako bi se moglo na jednom mjestu vršiti izmjene koje će onda biti

vidljive u cijeloj aplikaciji. Kôd 5.2 i kôd 5.3 prikazuju primjere pozivanja iste metode `Repo.DohvatiTeamove()` na dva različita mjesta u aplikaciji.

```
public ActionResult Dodajzadatak()
{
   ViewBag.KolekcijaGrupa = Repo.DohvatiGrupe();
   ViewBag.KolekcijaTeamova = Repo.DohvatiTeamove();
   ViewBag.KolekcijaPosebnihOznaka = Repo.DohvatiPosebneOznake();
   return View();
}
```

Kôd 5.2 Poziv metode za dohvat podataka o timovima pri izradi novog zadatka

```
public ActionResult Izvjestaj()
{
    ViewBag.Message = "Izvjestaj";
    ViewBag.KolekcijaTeamova = Repo.DohvatiTimove();
    return View(Repo.GetIzvjestaj());
}
```

Kôd 5.3 Poziv metode za dohvat podataka o timovima pri izradi izvještaja

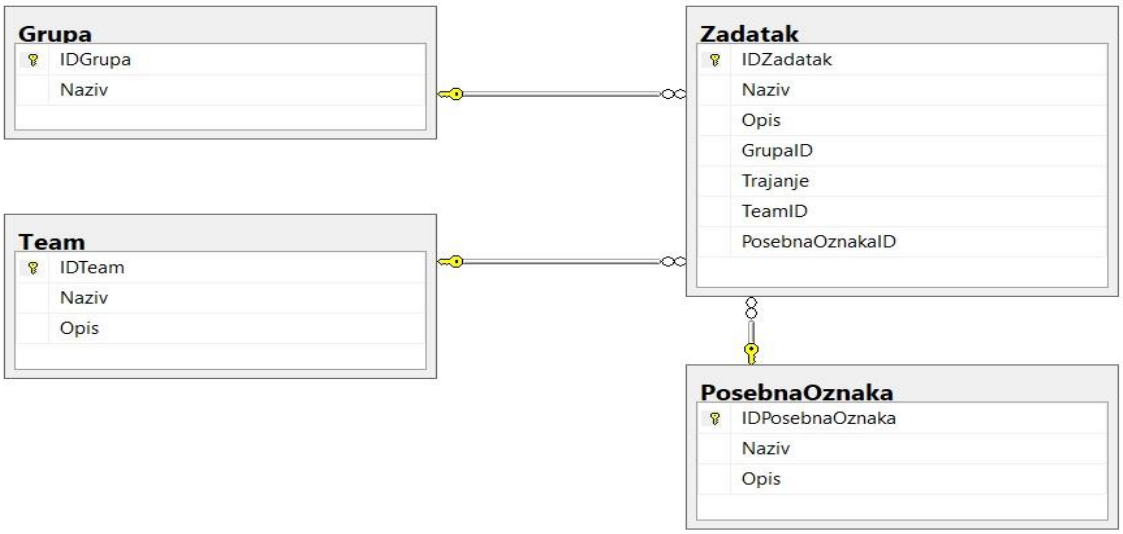
Smanjen je potencijalni otpad u aplikaciji na primjeru administratorskog sučelja. Naime nekom automatikom se pretpostavilo da će se aplikaciji pristupati kroz dvije grupe korisnika – normalni korisnik i administrator. Kada se započelo s programiranjem sučelja proučeni su ponovo zahtjevi korisnika u kojima se nije pronašlo korisnički zahtjev za više grupa korisnika. To je potvrđeno s krajnjim korisnikom i zaključeno je da administratorsko sučelje neće biti potrebno. Reakcijom u pravo vrijeme uštedjelo se na resursima i nije se uradilo nešto što bi bilo otpad u aplikaciji.

5.3. Sloj spremišta podataka

Spremište podataka se koristi za spremanje korisničkih podataka koja će se dalje obrađivati u sloju poslovne logike. Tablice koje se koriste su slijedeće:

- Zadatak
- Grupa
- Team
- PosebnaOznaka

Slika 5.4 pokazuje dijagram povezanosti tablica putem stranog ključa.



Slika 5.4 Dijagram povezanosti tablica

Tablica Grupa sadrži podatke o svim grupama/fazama kroz koje jedan zadatak prolazi u životnom vijeku. Povezana je stranim ključem s tablicom Zadatak na način `Grupa.IDGrupa = Zadatak.GrupaID`.

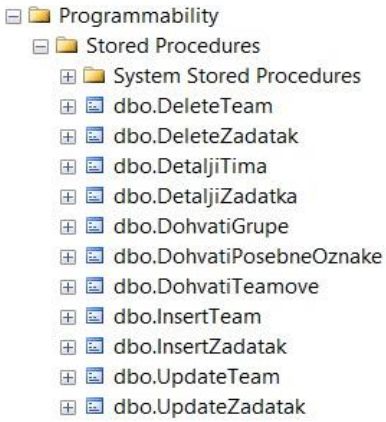
Tablica Team sadrži podatke o nazivu i kratkom opisu poslova timova koji sudjeluju u rješavanju zadataka. Povezana je stranim ključem s tablicom Zadatak na način `Team.IDTeam = Zadatak.TeamID`.

Tablica PosebnaOznaka sadrži podatke o nazivu i kratkom opisu za svaku pojedinu oznaku kojom se važnost zadatka. Povezana je stranim ključem s tablicom Zadatak na način `PosebnaOznaka.IDPosebnaOznaka = Zadatak.PosebnaOznakaID`.

Tablica Zadatak sadržava sve podatke o zadacima koji se rješavaju. Osim podataka koji su stranim ključevima vezani za druge tablice sadrži još informacije o nazivu, opisu i procijenjenom trajanju izvršavanja.

U spremištu podataka se još nalaze i procedure koje poziva sloj poslovne logike a služe za stvaranje novih, dohvat, izmjenu i brisanje postojećih podataka u prije navedenim tablicama.

Slika 5.5 prikazuje procedure koje se koriste iz aplikacije.



Slika 5.5 Procedure za stvaranje, dohvat, izmjenu i brisanje podataka

5.4. Sloj poslovne logike

Korištenjem MVC modela pri izradi ove aplikacije dobivena je jasna podjela u strukturi aplikacije. U prošlom poglavlju je opisana baza podataka a slijedećem će biti opisan prezentacijski sloj – sloj koji ih veže je sloj poslovne logike.

Za svaku tablicu iz baze podataka stvoren je model koji svojim atributima odgovara istoj. Prema tome modele koji se koriste za dohvat i rad s podacima iz baze podataka su: Zadatak, Team, Grupa i PosebnaOznaka. Kôd 5.4 prikazuje strukturu modela Team.

```
namespace LL_SSZ.Models
{
      public class Team
      {
            public       int IDTeam { get; set; }
                  [Required(AllowEmptyStrings = false, ErrorMessage
                                                = "Nije unesen naziv!")]
            public string Naziv { get; set; }
            public string Opis { get; set; }
      }
  }
```

Kôd 5.4 Struktura modela Team

Izvještaj je poseban model koji služi za prikaz obrađenih podataka u obliku izvještaja i za razliku od prije spomenutih modela nema tablicu u bazi čiju strukturu prikazuje. Svi navedeni modeli koriste se za inicijalizaciju instanci tipa podataka kako bi se s njima moglo koristiti. Nakon što su instance stvorene podaci u ubačeni u njih. Klasa Repo dohvaća sve relevantne podatke iz baze (*SELECT*) ali i vrši izmjene podataka (*UPDATE* i *DELETE*). Uz

metode za navedene funkcionalnosti posjeduje i metodu `GetIzvjestaj()`. Njen zadatak je priprema i obrada podataka potrebnih za izradu izvještaja.

Upravljanje aplikacijom dodijeljeno je HomeController-u koji sadrži popis svih akcija unutar aplikacije. Želi li se urediti neki podatak vezan uz određeni zadatak korisnik klikom miša na poveznicu zapravo poziva akcijsku metodu `Uredi(int id)`. Kôd 5.5 prikazuje navedenu akcijsku metodu.

```
[HttpGet]
public ActionResult Uredi(int id)
{
    ViewBag.KolekcijaGrupa = Repo.DohvatiGrupe();
    ViewBag.KolekcijaTeamova = Repo.DohvatiTeamove();
    ViewBag.KolekcijaPosebnihOznaka =
                                    Repo.DohvatiPosebneOznake();
    return View(Repo.DetaljiZadatka(id));
}
```

Kôd 5.5 Akcijska metoda Uredi (*HttpGet*)

`HttpGet` je oznaka da se koristi metoda za dohvat podataka. Prije samog poziva pogleda koji sadržava već postojeće podatke o odabranom zadatku (`return View(Repo.DetaljiZadatka(id))`) potrebno je pripremiti i podatke za padajuće izbornike na formi koja se koristi. Dohvaćanje podataka vrši se pozivanjem metode koja se nalazi u klasi Repo (npr. `Repo.DohvatiGrupe()`) a slanje dohvaćenih podataka u pogled preko rječnika ViewBag (npr. `ViewBag.KolekcijaTeamova = Repo.DohvatiTeamove()`). Na temelju pripremljenih podataka stvara se novi pogled koji korisniku omogućava pregled i izmjenu podataka vezanih za odabrani zadatak. Kada je završio s izmjenama promjene se mogu spremiti ili odbaciti. Ukoliko ih se želi spremiti poziva se akcijska metoda oznake `HttpPost` naziva `Uredi(Zadatak z)` koja će podatke spremiti u bazu i korisnika preusmjeriti na glavni prozor od kuda je i započeo radnju izmjene podataka. Kôd 5.6 prikazuje navedenu akcijsku metodu za spremanje izmjena.

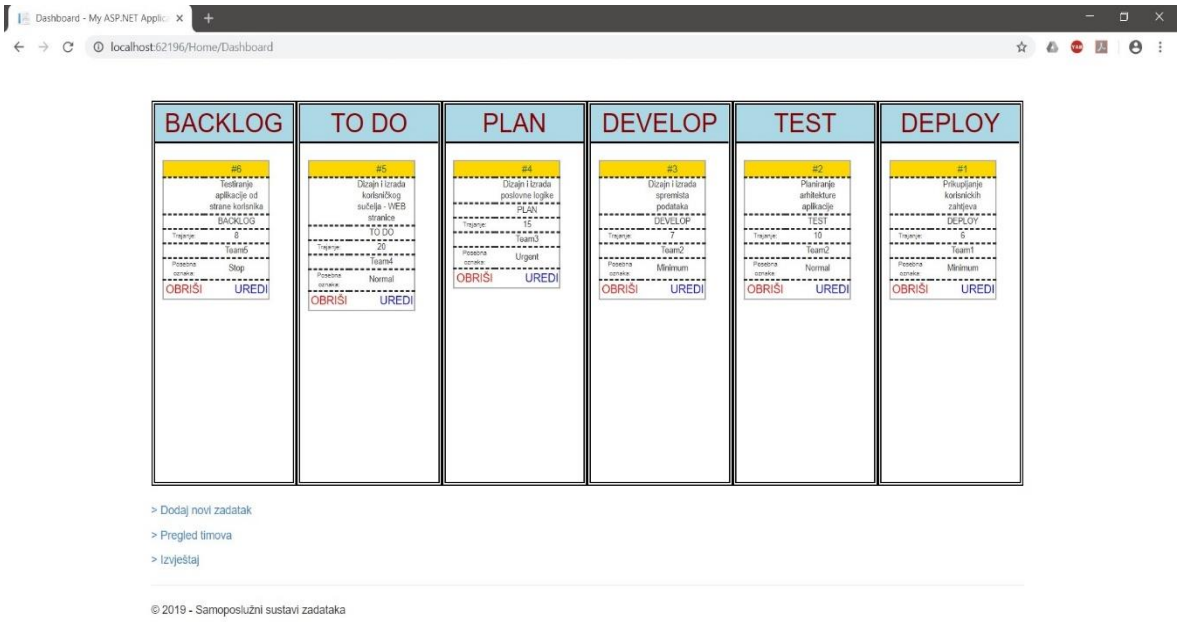
```
[HttpPost]
public ActionResult Uredi(Zadatak z)
{
    Repo.UpdateZadatak(z);
    return RedirectToAction("Dashboard");
}
```

Kôd 5.6 Akcijska metoda Uredi (*HttpPost*)

Ukoliko korisnik odustaje od promjena iste se ne spremaju u bazu i vraća se korisnika na mjesto od kuda je započeta sama radnja.

5.5. Prezentacijski sloj

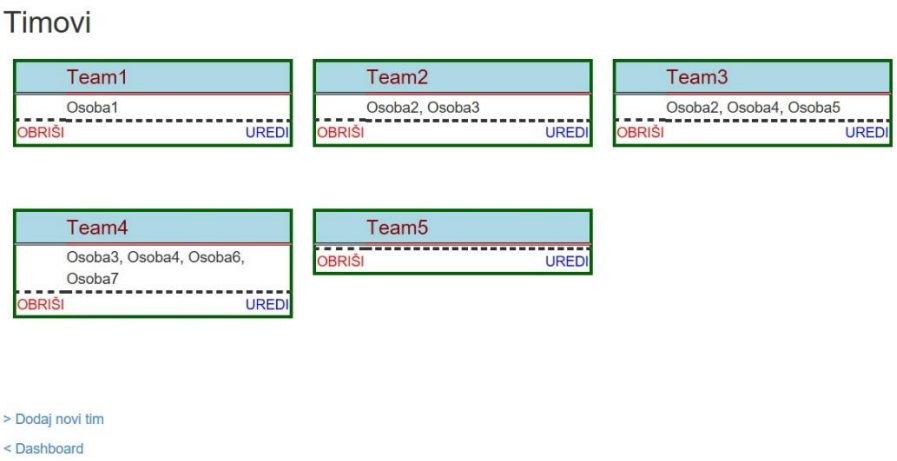
Prezentacijski sloj služi za prikaz podataka koja se u sloju poslovne logike dohvaćaju iz spremišta podataka i na njima vrši obradu. Prilikom pokretanja aplikacije pojavljuje se početni ekran koji pruža informacije o aplikaciji te nudi odabir nastavka rada u istoj. Glavna stranica aplikacije je ploča sa zadacima – *dashboard* koja nudi prikaz zadataka raspoređenih po grupama i timovima. U ovom pogledu postoje poveznice na nove poglede koji prikazuju, dodaju, ažuriraju i brišu podatke. Slika 5.6 prikazuje glavnu stranica Ploču s zadacima.



Slika 5.6 Ploča s zadacima – *dashboard*

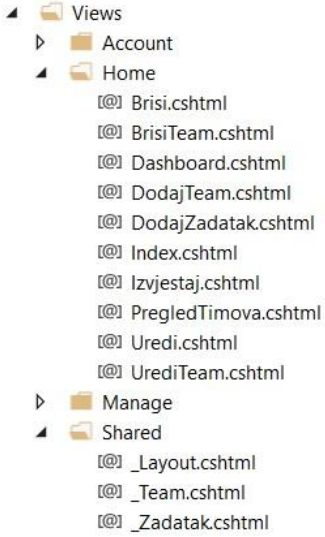
Za neke funkcionalnosti koriste se parcijalni pogledi koji omogućavaju jednake prikaze podataka istog tipa kada se dinamički kreiraju. Prilikom poziva akcijske metode `PregledTimova()` pokreće se pogled kojem se prosljeđuju podaci o naslovu pogleda i kolekcija podataka koja sadrži podatke za sve timove. S time se postiže da je izgled svih dinamički stvorenih instanci parcijalnog pogleda `_Team` je isti dok svaka ima različite podatke. Isti princip parcijalnih pogleda se koristi i na početnom ekranu kod dinamičkog stvaranja instanci postojećih zadataka.

Slika 5.7 prikazuje jedan takav pogled `PregledTimova`.



Slika 5.7 Pogled PregledTimova

Slika 5.8 prikazuje sve poglede i parcijalne poglede koji se koriste u aplikaciji.



Slika 5.8 Pregled pogleda i parcijalnih pogleda u aplikaciji

Zaključak

Korištenjem *Lean* metoda prilikom razvoja programskog rješenja osiguran je kvalitetan proizvod, isporučen u zadanom vremenu i u zadanim resursima.

Samo programsko rješenje je predviđeno za korištenje u malim timovima, ukoliko bi se javila potreba za korištenjem u većim timovima i većim projektima postoji mogućnost nadogradnje koja ne bi bila prezahtjevna jer su se koristili alati i struktura koji su pogodni za nadogradnje bez potrebe za strukturalnim promjenama. Tako bi se po potrebi moglo dodati administratorsko sučelje, povijest projekata, pohranjivanje dodatnog sadržaja vezanog uz projekt i slično. Ukoliko bi se u budućnosti krenulo s nadogradnjama postoji mogućnost da se naruši jednostavnost rada koju trenutno aplikacija pruža.

Programsko rješenje nije jedinstveno na tržištu i cilj ovog rada nije bio napraviti nešto što već ne postoji. Cilj ovog rada je bio pokazati da se korištenjem *Lean* metoda u razvoju aplikacije povećavaju šanse za uspješnost cijelog projekta, uzimajući u obzir i budućnost aplikacije u vidu održavanja i nadogradnje.

Prihvaćanje agilnog razvoja aplikacija te korištenje *Lean* metoda u razvoju budućnost je programiranja. Nekada nije lako, pogotovo u velikim tvrtkama, uvesti novine zbog inercije sustava ali primijeniti neke od metoda ne bi trebao biti problem. Svako umanjenje rizika od neuspjeha je pomak nabolje a *Lean* metode uvelike pridonose na tome planu.

Popis kratica

API	*Application programming interface*	programsko sučelje aplikacije
CSS	*Cascading Style Sheets*	stilski predlošci za vizualni prikaz
IDE	*Interactive Development Environment*	interaktivno razvojno rješenje
IIS	*Internet Information Services*	lokalni web servis
MVC	*Model-View-Controller*	model-pogled-kontoler
SQL	*Structured Query Language*	strukturirani jezik za upite
URL	*Uniform Resource Locator*	usklađeni lokator resursa
XML	*EXtensible Markup Language*	proširivi označni jezik

Popis slika

Popis kôdova

Literatura

[1] POPPENDIECK, M., POPPENDIECK, T. *Lean Software Development: An Agile Toolkit*. Addison-Wesley, 2003.

[2] KRIŽ, Z. Agilni razvoj. *Priručnik*. Visoko učilište Algebra u Zagrebu, 2014.

[3] LEAN ENTERPRISE INSTITUTE, INC. What is Lean?, https://www.lean.org/WhatsLean/, veljača 2018.

[4] C4MEDIA, INFOQ.COM The Three M's - The Lean Triad, https://www.infoq.com/articles/lean-muda-muri-mura, veljača 2018.

[5] TRELLO, ATALASSIAN, https://trello.com/tour, veljača 2019.

[6] JIRA SOFTWARE, ATLASSIAN, https://www.atlassian.com/software/jira, veljača 2019.

[7] TEAM FOUNDATION SERVER, MICROSOFT, https://visualstudio.microsoft.com/tfs/, veljača 2019.

Algebra
visoka škola za
primijenjeno računarstvo

PROGRAMSKI ALAT ZA SAMOPOSLUŽNE SUSTAVE ZADATAKA

Pristupnik: Luka Lukač, 0036386633

Mentor: Prof. Miroslav Popović