Ekstremno programiranje

Kovaček, Mateo

Sveučilište Jurja Dobrile u Puli

Odjel za informacijsko-komunikacijske znanosti

MATEO KOVAČEK

EKSTREMNO PROGRAMIRANJE

Završni rad

Pula, rujan 2016. godine

Sveučilište Jurja Dobrile u Puli

Odjel za informacijsko-komunikacijske znanosti

MATEO KOVAČEK

EKSTREMNO PROGRAMIRANJE

Završni rad

JMBAG: 0303046047, redoviti student

Studijski smjer: Informatika

Predmet: Softversko inženjerstvo

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijsko-komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Tihomir Orehovački

Pula, rujan 2016. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Mateo Kovaček, kandidat za prvostupnika informatike ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, rujan 2016. godine

IZJAVA

o korištenju autorskog djela

Ja, Mateo Kovaček dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom Ekstremno programiranje koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, rujan 2016.

Potpis

_____

SADRŽAJ

1. UVOD

Kada kažemo razvoj softvera obično mislimo na razvijanje nekakvih programa, jer se od nekoliko programa sastoji sam softver. Pojam programa može se definirati kao postupak za rješavanje određene vrste problema i to izvođenjem niza instrukcija određenim redom koji se pišu u nekom programskom jeziku. U današnje vrijeme postoji mnogo programskih jezika, današnji programeri su u nedoumici s kojim programom krenuti raditi. Treba odabrati pravi alat i pravu metodu razvoja softvera kako bi što uspješnije i kvalitetnije razvili softverski proizvod. Konkurencija razvoja softvera je velika. Današnji kupci imaju mnogo kriterija kada je u pitanju izbor proizvođača softvera. Neki od tih kriterija su: kreativnost, da sadrži što manje grešaka i normalnu brzinu rada bez zastajkivanja i uz to bi softver trebao biti isporučen u što bržem roku. Da bi sve to bilo omogućeno važno je odabrati pravu metodologiju razvoja softvera.

Metodologija razvoja softvera se kao pojam može definirati kao: „disciplina koja se bavi svim aspektima razvoja softvera, ona se bavi modelima, metodama i alatima koji su potrebni da bi na što jeftiniji način mogli proizvesti što kvalitetnije softverske produkte.“ (Manger, 2013., str. 9). Softversko inženjerstvo počelo se razvijati krajem 60-tih godina 20. stoljeća dolaskom treće generacije računala. Došlo je do potrebe za složenijim softverima koji bi mogli obavljati više zadataka u isto vrijeme, odnosno višezadaćnosti (eng. *multi-tasking*). Razvijali su se novi programski jezici, grafički jezici za modeliranje, bolji načini utvrđivanja zahtjeva, pouzdaniji način vođenja projekta i način procjene troškova, alati koji nastoje automatizirati proces razvoja softvera. „Zahvaljujući takvom razvoju softversko inženjerstvo uspjelo se etablirati kao važan dio tehnike i računarstva.“ (Manger, 2013., str. 10)

Tema ovog završnog rada je Ekstremno programiranje (XP). Cilj rada je dokazati zašto agilne metode ili točnije, zašto Ekstremno programiranje spada u jedne od najpoznatijih metoda za razvoj softvera u današnje vrijeme. To će biti dokazano kroz detaljno opisivanje svake aktivnosti i primjene XP-a u izradi softverskog produkta koja će biti odrađena kroz sljedeća tri poglavlja. Kroz ovaj rad korištena je deduktivna metoda istraživanja. Uz pomoć te metode došlo je do novih spoznaja i saznanja o čemu će biti više riječi u zaključku rada.

U prvom poglavlju objašnjava se pojam agilnih metoda te se navode i objašnjavaju neke druge agilne metode. Na kraju prvog poglavalja napravljena je usporedba s klasičnim metodama razvoja softvera. U drugom poglavlju objašnjava se pojam Ekstremnog programiranja. Dok se u četvrtom poglavlju detaljno objašnjava cjelokupni razvoj softvera od planiranja softvera, dizajna softvera, implementacije pa sve do testiranja softvera. U petom poglavlju rada nalazi se zaključak.

2. AGILNE METODE RAZVOJA SOFTVERA

Agilne metode razvoja softvera mogu se definirati kao pojam koji označava nekoliko iterativnih i postupnih metodologija za razvoj softvera. Ova metoda vuče korijene iz japanske filozofije koja je nastala još prije 20. godina, a tek danas se ta metoda počela intenzivnije koristiti (Zekić-Sušac, 2013). Agilne metode obilježavaju sljedeća svojstva:

1. Razvoj softvera promatra se kao kontinuirani niz iteracija čiji broj nije moguće unaprijed predvidjeti. Svaka iteracija usklađuje sustav s trenutnim zahtjevima. Zahtjevi se stalno mijenjaju, ne postoji cjelovita ili konačna specifikacija.
2. Izbjegavaju se svi oblici dokumentacije, osim onih koji se mogu automatski generirati iz programa. Smatra se da sam kôd u izvornom obliku predstavlja svoju najpouzdaniju dokumentaciju.
3. Ne postoji upravljanje projektom, aktivnosti se dogovaraju s korisnicima te se odvijaju uz njihovo aktivno sudjelovanje. Planiranje je kratkoročno, maksimalno dva tjedna unaprijed (Manger, 2013).

Iako svaka agilna metoda ima svoje specifičnosti, sve se drže određenih principa kao što su: zadovoljiti kupca brzom isporukom softvera, promjene zahtjeva čak i u kasnijim fazama razvoja su moguće, suradnja korisnika i članova projekta što češće kako bi se izbjegle sve nejasnoće i krajnji produkt bio što bolji, promoviranje održivog razvoja kroz održavanje stalne suradnje sponzora, programera i korisnika, stalna koncentracija na tehničku izvrsnost i dobar dizajn, jednostavnost kako bi korisnicima bilo što lakše koristiti novi softver.

Agilan način razvoja softvera danas je najčešće korišten način razvoja softvera. Naime, istraživanja iz 2011. godine pokazala su da 80% tvrtki koristi agilne metode u razvoju softvera. Rezultati su se također poboljšali, preko 70% tvrtki poboljšalo je svoju produktivnost, veća su zadovoljstva kupaca, radna atmosfera između radnika znatno se poboljšala. U današnjem svijetu, da bi kompanije bile uspješne i konkurente trebale bi usvojiti agilan razvoj jer praktične, jednostavne i uspješne agilne metode održavaju kompaniju živom među silnom konkurencijom koja se nalazi na današnjem tržištu (Algebra otvoreno učilište, 2014).

2.1. SCRUM

Scrum metoda razvoja softvera zasniva se na teoriji empirijske kontrole procesa, odnosno da znanje dolazi iz iskustva odlučivanja temeljenih na poznatom. Empirijska kontrola procesa temelji se na kontroli, transparentnosti i prilagodbi. Transparentnost se zasniva na tome da ključni aspekti moraju biti vidljivi onima na kojima leži odgovornost za proizvod. Korisnici Scrum-a često moraju kontrolirati liste zadataka projekta i tijek implementacije kako bi se na vrijeme uočili neki nedostatci ili eventualne pogreške, na čemu se temelji kontrola, a prilagodba se odnosi na korekciju proizvoda ako je to naravno potrebno.

2.1.1. KONTROLA I PRILAGODBA

Metoda Scrum propisuje četiri formalne prilike za kontrolu i prilagodbu, a to su:

- Sastanak planiranja perioda implementacije
- Dnevni Scrum sastanak
- Revizija perioda implementacije
- Retrospektiva perioda implementacije

Period implementacije može se smatrati kao jedan od glavnih dijelova i on predstavlja vremenski ograničen period od mjesec dana ili pak kraće, tijekom kojeg je proizvod završen upotrebljiv i potencijalno isporučiv. Dnevni Scrum je sastanak koji obično traje maksimalno 15-tak minuta, koji služi da razvojni tim dâ kratko izvješće o dosadašnjem tijeku razvoja i donese plan za idućih 24 sata. Revizija perioda implementacije odvija se pri samom završetku implementacije radi kontrole inkrementa i prilagođavanja zadataka projekta. Retrospektiva perioda je prilika da članovi projektnog tima analiziraju svoj odnos prema radu i donesu plan za unapređenje kojeg će provesti tijekom sljedećeg perioda implementacije (Radovan i sur., 2010).

2.1.2. ČLANOVI TIMA

Članovi tima čine vlasnik proizvoda, razvojni tim i Scrum majstor. Vlasnik proizvoda nadzire razvoj proizvoda i kontrolira rad razvojnog tima uz pomoć liste zadataka projekta. Lista zadataka projekta je artefakt Scrum-a koji sadrži sve važne informacije koje su potrebne za razvoj proizvoda i jedini je izvor zahtjeva za promjene koje se rade na proizvodu. Osim liste zadataka projekta, vlasnik proizvoda također može koristiti artefakt lista zadataka perioda implementacije, koji predstavlja skup stavka koje su odabrane za period implementacije zajedno s planom realizacije inkrementa i cilja perioda implementacije. Za analizu i procjenjivanje razvoja proizvoda koriste se grafovi sagorijevanja, oni prikazuju procjenu napretka perioda implementacije. Razvojni tim se obično sastoji od tri do devet članova, kako bi se izbjeglo premalo interakcije, nedostatak vještine, a i previše utrošenog vremena na interakciju. Scrum majstor odgovoran je da se svi drže načela i pravila Scrum metode razvoja.

2.1.3. IMPLEMNTACIJA SCRUM-A

Prva faza se sastoji od sastavljanja liste zadataka projekta koja se mora izvršiti do kraja projekta. Zadaci se moraju definirati tako da se mogu riješiti u roku jednoga ili dva dana i da se ti zadaci mogu raščlaniti na podzadatke kada se prikupi dovoljno informacija i znanja o zadanim zadacima. Na primjer, ako je potrebno implementirati web aplikaciju kao što je studomat u programskom jeziku Java koristeći MVC (eng. *Model View Controller*) arhitekturu s programskim okvirom Hibernate, lista zadataka projekta bi mogla ovako izgledati (tablica 1):

Tablica 1. Primjer popisa zadataka u inicijalnoj listi zadataka projekta (Radovan i sur., 2010)

R. broj	Lista zadataka projekta
1.	ER MODEL BAZE PODATAKA
2.	KREIRANJE TABLICA U BAZI PODATAKA
3.	KREIRANJE JAVA DOMENSKIH KLASA
4.	KREIRANJE HIBERNATE "MAPPING"-A ZA DOMENSKE KLASE
5.	JAVA "DAO" SLOJ ZA PRISTUPANJE BAZI PODATAKA
6.	JUNIT TESTOVI ZA TESTIRANJE "DAO" SLOJA

7.	JAVA "SERVISNI" SLOJ
8.	JUNIT TESTOVI ZA JAVA "SERVISNI" SLOJ
9.	DIZAJN TOKA EKRANA
10.	IMPLEMENTACIJA "CONTROLLER" KLASA
11.	JUNIT TESTOVI ZA METODE U "CONTROLLER" KLASAMA
12.	DIZAJN I IMPLEMENTACIJA EKRANA
13.	FUNKCIONALNO TESTIRANJE EKRANA
14.	PISANJE DOKUMENTACIJE ZA APLIKACIJU

2.2. KRISTALNA PORODICA METODOLOGIJA

Razvio ju je Alistar Cockburn 1991. godine. On smatra da se jedna metodologija za razvoj softvera ne može primjenjivati u svim situacijama. Iz koncepta da su eng. *people-centric* metodologije bolje od eng. *process-centric* metodologije, razvija se skup metodologija koja se zove Kristalna porodica metodologija, od 1998. godine taj skup metodologija dobiva naziv Kristal. Određene metodologije su dobile ime po geološkim kristalima:

- Čisti (eng. *clear*)
- Narančasti
- Žuti
- Crveni
- Smeđi (eng. *maroon*)

Najpoznatije i najkorištenije metode su: čisto-kristalna (*eng. crystal clear*), kristalno-narančasta (*eng. crystal orange)* i kristalno-narančasta web (*eng*. *crystal orange web*).

2.2.1. OBILJEŽJA METODE

Za uspješno izvršavanje metode potrebno je pridržavati se načela kao što su:

1. Učestala isporuka – kroz određeni period, korisnike se upoznaje s početnim verzijama produkta, dobiva se povratna informacija korisnika.
2. Kontinuirane povratne informacije – projektni tim raspravlja o aktivnostima, provodi se validacija dosadašnjeg projekta te se raspravlja o mogućim greškama i problemima.

3. Stalna komunikacija – veliki timovi su lokacijski povezani, mali timovi su svi smješteni u jednu sobu.
4. Sigurnost – svi članovi tima komuniciraju, rade bez represije, svi projekti nisu kritično isti.
5. Fokus – članovi tima se moraju upoznati s prioritetnim ciljevima, treba im dati mogućnost da ih samostalno rješavaju.
6. Raspoloživost korisnika – članovima tima se omogućuje konstantna komunikacija s klijentima.
7. Automatski testovi i integracija – konstanto verifikacija proizvoda radi manje mogućnosti grešaka u daljnjem radu.

Za uspješno izvršavanje projekta potrebno je da projektni tim učestalo komunicira, izbaci svu birokraciju, korisnik je praktički dio projektnog tima, testiranje mora biti učestalo, također je važno da bude omogućeno brzo i redovito stvaranje dijelova softvera ispravne funkcionalnosti (Ekonomski fakultet Zagreb, 2009).

2.3. ADAPTIVNI RAZVOJ SOFTVERA (ASD)

Adaptivan razvoj softvera nastao je 1992. godine kao posljedica brzog razvoja softvera. Osnovne karakteristike ASD su:

- Fokusiranje na misiju
- Zasnivanje na karakteristikama
- Vođenje rizikom
- Tolerantnost na promjene

Adaptivni razvoj zamjenjuje dosad tradicionalni vodopadni ciklus . Kada dođe do promjena u projektu, adaptivni tim se također mijenja. Problem nastaje kada projektni tim treba opisati buduće radnje. Obično se planiraju aktivnosti za najviše tjedan dana unaprijed. Izvještaji o aktivnosti za sljedećih šest mjeseci mogu sadržavati informacije samo o očekivanim rezultatima, što ovisi o težini samog zahtjeva, kao o datumu izdavanja nove verzije (Gavrić, 2014).

2.4. LEAN

Metoda razvoja Lean svoje korijene vuče još iz sustava proizvodnje koju je razvila tvrtka Toyota, čijom su metodom znatno povećali kvalitetu svojih vozila, postižući veliku briznu i učinkovitost u proizvodnji, od nabave materijala do isporuke kupcu. Tom i Mary Poppendieck su ta načela primijenili na sam razvoj softvera. Osnova svega je stalno poboljšavanje te stvaranje prvo ljudi, a zatim proizvoda.

Postoje određena Lean načela kojih se treba pridržavati da bi krajnji rezultat bio što uspješniji:

- Poboljšanje dizajna
- Male verzije
- Pridržavanje standarda kodiranja
- Kolektivno vlasništvo kôda
- Jednostavan dizajn
- Održivi korak

Sva ta načela čine jednu cjelinu i kao takve treba ih se sagledavati i primjenjivati. Također, postoje aktivnosti s kojima se samo koči Lean i trebalo bi ih se u što većem luku zaobilaziti, a to su: kašnjenje u izradi softvera, nejasni zahtjevi, nedovoljno testiranje, preopterećenje birokracijom, spora interna komunikacija. Da bi se takve aktivnosti znale izbjeći treba ih na vrijeme znati identificirati. Ukoliko se aktivnost može zaobići, znači da je višak. Procesi i funkcije koje klijenti ne koriste se također smatraju viškom i trebale bi se izbjegavati. Greške se sprječavaju testiranjem, najbolja praksa je odmah nakon što je kôd napisan. Nastoji se smanjiti pisanje prekomjerne dokumentacije ili da se ide u prevelike detalje prilikom planiranja, umjesto toga fokusira se više na pisanje kôda. Prilikom prikupljanja zahtjeva od klijenata pokazuju im se određeni primjeri, a na temelju tih primjera dobiva se povratna informacija kako bi budući produkt trebao izgledati.

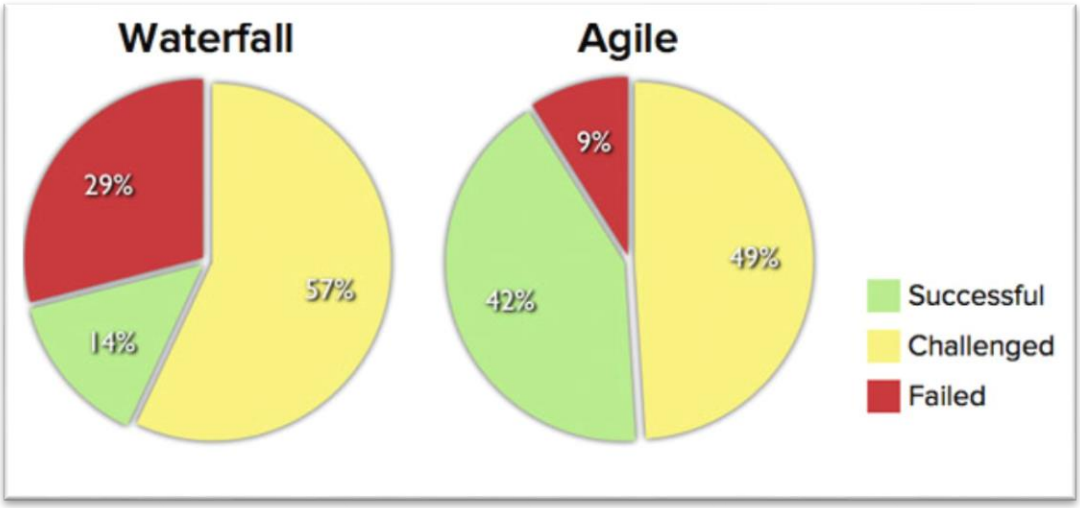
Bolji rezultati postižu se odgađanjem. Proizvod se pokazuje i izdaje tek kada je zasnovan na čvrstim temeljima, a ne na predviđanjima i pretpostavkama. Ako je neki softver složen, u njega treba ugraditi više mogućnosti za promjenama, omogućavajući odlaganje važnih i ključnih odluka. Nastoji se što brže proizvod isporučiti klijentu, da bi se brzo dobila povratna informacija i kako bi se moglo nastaviti dalje s radom. Klijenti

ne žele dugo čekati na svoje proizvode jer oni cijene brzu isporuku kvalitetnog proizvoda. U klasičnim metodama, menadžer upravlja projektom, dok u Lean metodi menadžeri slušaju programere, kako i što dalje raditi. Sam korisnik mora biti upućen u razvoj proizvoda. Tijek informacija je dvosmjeran, ide od programera do klijenta i obratno. Da bi Lean metoda bila uspješna, njena primijenjena mora biti razumljiva svim članovima tima prije samog korištenja (Gavrić, 2014).

2.5. RAZLIKA IZMEĐU AGILNIH I KLASIČNIH METODA RAZVOJA SOFTVERA

Agilne metode za razliku od ostalih metoda uvode nove uloge stručnjaka u projektni tim i drukčije organiziraju aktivnosti unutar razvoja softvera. Kod klasičnih metoda uloge su raspoređene tako da, na primjer, imamo stručnjaka za marketing, upravljanje projektom itd. koji zajedno dijele odgovornost, dok kod agilnih metoda postoji samo jedna osoba koja je odgovorna za sve, a ona se zove vlasnik produkta (eng. *Product owner)*. Kod agilnih metoda svi članovi skupa rade na projektu. Rade se kratki i česti sastanci projektnog tima te se iz dana u dan prati dinamika projekta, dok je kod klasičnih metoda na primjer stručnjak za marketing odvojen od ostataka tima i on radi sam. Analiza tržišta, utvrđivanje zahtjeva, oblikovanje i implementacija se kod klasičnih metoda određuje prije samog početaka rada, na samom proizvodu i ne smiju se kasnije mijenjati, a kod agilnih metoda aktivnosti se izvršavaju u dogovoru s korisnikom i smiju se mijenjati ili nadograđivati. Također, kod agilnih se metoda planira projekt u dijelovima, odnosno zadaju se kratkoročni ciljevi. U agilnim metodama nema dokumentacije, nego dokumentaciju čine testovi s kojima se provjerava radi li program po svojim specifikacijama. Kad god se neka nova funkcionalnost implemntira, ona se odmah testira da bi se pronašle moguće greške i odmah se otklonile. Kod klasičnih se pak metoda posebno sastavlja dokumentacija, a samim time se i obavlja dodatan posao. Kod klasičnih metoda povratne informacije od korisnika stižu u kasnijim fazama razvoja i testiranja produkta, odvijaju se nakon njegova lansiranja, dok se kod agilnih metoda ranije izdaju nove verzije proizvoda. Nakon toga se organiziraju sastanci s korisnicima kako bi oni analizirali proizvod i utvrdili odgovara li proizvod njihovim specifikacijama i zahtjevima. Prednost klasičnih metoda su viša uključenost menadžera u sami projekt. Dok je kod agilnih metoda teško pratiti projekt ako nema dovoljno znanja iz područja informatike i računarstva. Taj se problem pokušava riješiti

uz pomoć Project Management Institute (PMI) gdje će se menadžeri certificirati za upravljanje agilnim projektima (Radovan i sur., 2010). U nastavku teksta prikazana je razlika uspješnosti razvoja softvera između agilnih i Vodopad metode (slika 1).



Slika 1. Usporedba agilne metode i vodopad metode (Izvor: OneDesk , 2013)

Na prikazanim grafovima možemo vidjeti da su *agilne metode* uspješnije u razvoju softvera od *vodopad metode*. Čak 42% projekata je uspješno završeno, a 9% ih je propalo. Dok je kod *vodopad metode* samo 14% uspješno završeno, a čak 29% projekata je propalo.

3. EKSTREMNO PROGRAMIRANJE (XP)

Ekstremno programiranje je najčešće korištena agilna metoda razvoja softvera, koju je razvio Kent Beck skupa s Ward Cunningham sredinom 80-tih godina prošlog stoljeća (McBreen, 2002). Prvo izdanje svoje knjige „Extreme Programming Explained“ Kent je izdao 1999. godine, a drugo izdanje 2004. godine koja se razlikuje po nekim principima, neki su principi dodani, a neki su se stopili s postojećim (Pap, 2008).

XP je prvenstveno namijenjen malim timovima koji razvijaju softver, a koji su suočeni s neodređenim zahtjevima ili zahtjevima koji se stalno mijenjaju. XP je nastao iz problema dugoročnih razvojnih ciklusa koji su se odvijali kod tradicionalnih modela. Započelo je jednostavno, kao prilika da se završe poslovi s praksama koji su se smatrali dovoljno efikasnim u procesu razvoja softvera. Nakon što se uvidjelo da se XP pokazuje dobrom praksom, njezini ključni principi su bili dokumentirani.

3.1. UVOD U EKSTREMNO PROGRAMIRANJE

XP je disciplinirani pristup razvoja softvera koji spada pod agilne metode razvoja softvera. Ekstremno programiranje jedna je od novijih metoda koja se prvi put uporabila 6. ožujka 1996. godine. Jedan od temeljnih ciljeva XP-a je zadovoljavanje, odnosno zadovoljstvo kupca ili korisnika softvera. Zadovoljstvo korisnika je konačni cilj u ciklusu razvoja softvera jer se odnosi na ispunjenje dobivenih ulaznih zahtjeva. Ti zahtjevi su ispunjeni ako je korisnik zadovoljan s funkcionalnošću samog softvera. Metodologija XP-a je dizajnirana tako da se vrši implementacija točno onoga što je unaprijed dogovoreno. Također omogućuje da se promjene mogu implementirati i u najkasnijim fazama razvoja softvera. Timski rad dolazi do velikog isticanja. Kupci, menadžeri, programeri, svi su oni dio tima koji proizvodi softver. U XP se koristi efikasan način rada koji se temelji na timovima, najčešće u parovima (Maržić, 2005).

3.2. ŽIVOTNI CIKLUS

Životni ciklus ekstremnog programiranja se u većini razlikuje od klasičnih modela razvoja softvera. Karakteristika agilnih pa tako i ekstremnih programiranja je izbjegavanje pisanja formalnih dokumenta. XP koristi drukčiji pristup koji se zasniva na „verbalnim“ dokumentima. Također, on spaja sve svoje životne cikluse u jedno i primjenjuje ih svaki dan. Razlog tome je visok stupanj komunikacije koji se odvija između članova tima, a čiji se broj obično kreće oko deset osoba po projektu. Obično jedna iteracija traje od tjedan dana pa do tri tjedna, nekad i duže, što se nikako ne preporučuje. Kraće iteracije omogućuju timu brzo i lako prilagođavanje korisničkim zahtjevima.

3.2.1. TEMELJNE KARAKTERISTIKE EKSTREMNOG PROGRAMIRANJA

1. Igra planiranja (eng*. Planning Game*)
   Interakcija između korisnika i programera oko procjene implementacije pojedine funkcionalnosti.
2. Malene, ali česte isporuke (eng. *Small/short releases*)
   Manji dijelovi softvera, odnosno njegove funkcionalnosti, se često isporučuju svaka 2 do 3 mjeseca.
3. Organizacija sustava s metaforama (eng. *Metahaphor*)
   Metafora je pojednostavljena slika sustava u razvoju.
4. Jednostavan dizajn (eng. *Simple design*)
   Koristi se najjednostavnije rješenje koje postoji bez kompliciranog kôda i dodatne funkcionalnosti.
5. Testiranje (eng. *Testing*)
   Kontinuirano, nakon svake nove implementacije se softver testira.
6. Refaktoriranje kôda (eng. *Refactoring*)
   Micanje dvostrukog (redudantnog) kôda i korištenje jednostavnog kôda.
7. Programiranje u paru (eng. *Pair Programming*)
   Dva programera zajedno pišu određeni kôd, tako da jedan drugom mogu ispravljati greške.
8. Zajednički pristup kôdu (eng. *Collective Ownership*)

Bilo tko iz tima ima pristup tuđem kôdu, također ima dopuštenje za mijenjanje bilo čijeg kôda.

9. Kontinuirana integracija (eng. *Continuous intergration*)
   Novi kôd se integrira u sustav čim je testiran i spreman.
10. 40 satni radni tjedan (eng. 40 *hours week*)
    Maksimum rada tjedno je 40 sati. Nije preporučeno prekovremeno jer se na taj način uništava timski duh.
11. Na licu mjesta kupac (eng. *On-site customer*)
    Kupac je uvijek timu na raspolaganju.
12. Standardi kôdiranja (eng*. Coding standards*)
    Standardi kôdiranja se trebaju slijediti kako bi drugim programerima kôd bio što razumljivi i lakši za čitati (William, 2001).

3.2.2. TOK AKTIVNOSTI

U ovom djelu opisuje se uobičajeni tok aktivnosti tijekom jednog osmosatnog radnog dana kod ekstremnog programiranja. Aktivnost počinje desetominutnim sastankom prije početaka rada. Daju se informacije o prošlim aktivnostima i zadaju se zadaci za tekući dan. Svaki član tima bira svoje zadatke ili nastavlja s prošlim. Niti jedan zadatak ne smije trajati duže od tri dana. Formiraju se parovi, tko će prvi programirati, a tko će analizirati kôd. Nakon toga slijedi pisanje unit testova uz pomoć kojega se predviđaju budući problemi, korisničke akcije i tok te komponente, sve se to događa prije samog pisanja kôda. Sljedeća aktivnost je pisanje kôda uz poštivanje standarda kodiranja, pokušavaju se napisati što jednostavniji kôdovi tako da bi testiranje bilo uspješno, u slučaju nekih nejasnoća obraća se klijentu za pomoć. Nakon toga vrši se testiranje čiji se rezultat mora uklopiti u sve postojeće testove. Nakon završetaka zadataka, na redu je integracija programa koja sadrži dosad sve komponente, dosadašnja verzija programa ne smije biti narušena, a ako to nije moguće odbacuje se zadatak i sljedeći dan se zadaje novi zadatak (Tadić, 2005). Slika 2 prikazuje uobičajeni tok aktivnosti tijekom jednog radnog dana u XP-u (slika 2).

Slika 2. Tok aktivnosti u XP-u
(Izvor: Tadić, 2005)

3.3. ULOGE UNUTAR EKSTREMNOG PROGRAMIRANJA

Svaki član unutar razvojnog tima ima svoju ulogu. Uloge unutra XP mogu se raspodijeliti u 4 skupine. Te skupine su: programeri, klijenti, test-osobe i treneri.

Prva skupina uloga su programeri. Zadatak programera je pronalaženje najefikasnijeg načina za implementaciju korisničkih zahtjeva, međutim najvažniji zadatak im je ipak programiranje. Najčešće se programira u paru koristeći razvoj usredotočen prema testiranju (eng. TDD- *Test-driven development*), pišu testove za testiranje softvera, ali bave se i dizajnom softvera. Ako naiđu na neke probleme ili imaju neka pitanja, korisnik im je uvijek na raspolaganju. Razvojni tim bi se uglavnom trebao sastojati od parnog broja programera, tako da svi mogu raditi u paru. Uglavnom se razvojni tim sastoji od 4 do 6 programera.

Druga skupina uloga su klijenti (eng. *On-Site Clients)*. Njihov zadatak je definiranje kako će softverski proizvod izgledati i funkcionirati. Oni ne moraju biti ni pravi klijenti, nego je samo važno da znaju dobro definirati softverski proizvod. Najvažnija funkcija im je planiranje, odnosno određivanje vizije projekta, identifikacija zahtjeva kao i određivanje njihovog prioriteta, upravljanje rizikom itd. Još jedna vrlo važna funkcija je pružanje programerima informacije oko detalja zahtjeva. Razlog tome je što XP nema posebnu listu specifikacija zahtjeva pa im klijenti služe kao „živa“ specifikacija i klijenti moraju biti dostupni programerima u svakom trenutku. Oni

također sudjeluju u testiranju. Klijenti su zaduženi za kreiranje klijentskih testova da bi se moglo provjeriti rade li funkcionalnosti onako kako su si to klijenti zamislili. Sve to ne bi bilo moguće bez kvalitetne komunikacije, zato je važno da klijenti prisustvuju svakom sastanku razvojnog tima. Klijenti se mogu podijeliti na: menadžere proizvoda, dizajnere interakcije, stručnjake domena i poslovne analitičare.

- Menadžer proizvoda (eng. *Product Maneger*) ima vrlo važan zadatak, odnosi se na održavanje i promociju vizije proizvoda, dokumentiranje vizije te predavanje vizije proizvoda tvrtci, definiranje zahtjeva, određivanje njihovog prioriteta, analiza napretka softverskog proizvoda, vođenje prezentacija na krajevima iteracija, reklamiranje, promoviranje i prezentiranje krajnjeg softverskog proizvoda. Kvalitetan menadžer proizvoda dobro poznaje tržište, zna što je potrebnu tržištu, prepoznaje konkurenciju i ima znanja i vještine da donese kvalitetne odluke. Svakom projektu je potreban jedan menadžer proizvoda.
- Dizajner interakcije (eng. *Interaction Designers*) ima za zadatak definiranje i kreiranje korisničkog sučelja. Njihova uloga je procjenjivanje kako bi korisnici željeli koristiti taj proizvod i to na način da ispituje korisnike, rade ankete s korisnicima kako bi saznali njihovo viziju proizvoda. Njihova najčešća metoda je promatranje korisnika kako bi na taj način saznali njihove ukuse i ideje o proizvodu.
- Stručnjak domene (eng*. Domain Experts*) ima za zadatak određivanje detalja kod zahtjeva te pisanje klijentskih testova. Da bi softver imao vrijednosti, mora se dokazati da je dovoljno profesionalan u toj domeni. Ali, problem je što u većini slučajeva programeri nemaju dovoljno znanja o tom području. Tu dolazi uloga stručnjaka domene.
- Poslovni analitičar (eng*. Business Analyst*) služi kao pomoćni klijent čiji je zadatak poboljšanje korisničkog zahtjeva tako što će kroz razgovor prikupiti zaboravljene podatke.

Optimalan razmjer između programera i korisnika trebao bi biti 3:2 (tri programera nasuprot dva korisnika). Korisnici uvijek trebaju biti na raspolaganju ostatku tima, oni uvijek moraju biti jedan korak ispred programera, ali treba naglasiti da uvijek jedan od korisnika mora biti menadžer proizvoda.

Treća skupina uloga su test osobe (eng. *Testers*). Njihov zadatak je da se softverski proizvod isporuči korisnicima bez greške. Oni pišu istraživačke testove i uz pomoć njih rade testiranje na softverskom proizvodu. Također testiraju performanse softvera. Njihova uloga je samo da pronađu greške, ali na njima nije da ih ispravljaju, to je posao programera. Kada se pronađe greška test-osobe pomažu ostatku tima popraviti grešku tako da sljedeći put ne dođe do takve greške. Razmjer između programera i test-osoba je 4:1 (četiri programera nasuprot jedan tester). To je dovoljan broj test-osoba s obzirom da se svi u timu bave testiranjem.

Četvrtu skupinu uloga čine treneri (eng. *Coaches*). Njegova uloga je uloga lidera u timu. Iako se u XP-u smatra da se timovi sami organiziraju, tj. da svaka osoba obavlja posao kako misli da će najbolje pomoći timu, ipak mora postojati osoba koje će sve to regulirati. Njegove funkcionalnosti nisu iste kao kod ostalih metoda, da podjeli zadatke i nadgleda rad, njegova uloga je pomoći timu da što bolje ostvari svoje ciljeve. Trener će poduzeti sve moguće mjere da timu osigura što kvalitetniju radnu atmosferu, odrediti će sastav tima, njegova uloga je, također, biti posrednik između tvrtke koja naručuje proizvod i tima. Njegovi poslovi još uključuju planiranje i pružanje pomoći oko programiranja. Treneri se mogu podijeliti na trenere-programere i menadžere projekta.

- Treneri-programeri (eng. *Programmer-Coaches*) su iskusni programeri koji pomažu drugim programerima u timu. Njihov primarni cilj je promicanje agilnih metoda u programiranju, kao što je programiranje u paru.
- Menadžer projekta (eng. *Project Manager*) ima za zadatak da tim i uprava tvrtke pronađu zajedničku riječ. Menadžer projekta je potreban jer još mnogo uprava ne zna kako točno funkcionira ekstremno programiranje. Osim posla komuniciranja s upravom, menadžer projekta sudjeluje u poslovima vezanim za planiranje.

Preporučeno je da svaki tim ima barem jednog trener-programera i ako je potrebno jednog menadžera projekta (Pap, 2008).

4. RAZVOJNA METODOLOGIJA EKTREMNOG PROGRAMIRANJA

U ovom odjeljku će detaljnije biti opisan cijeli postupak razvoja softvera, koji se sastoji od planiranja razvoja softvera, dizajna softvera, kodiranja softvera i implementacije te testiranja softvera.

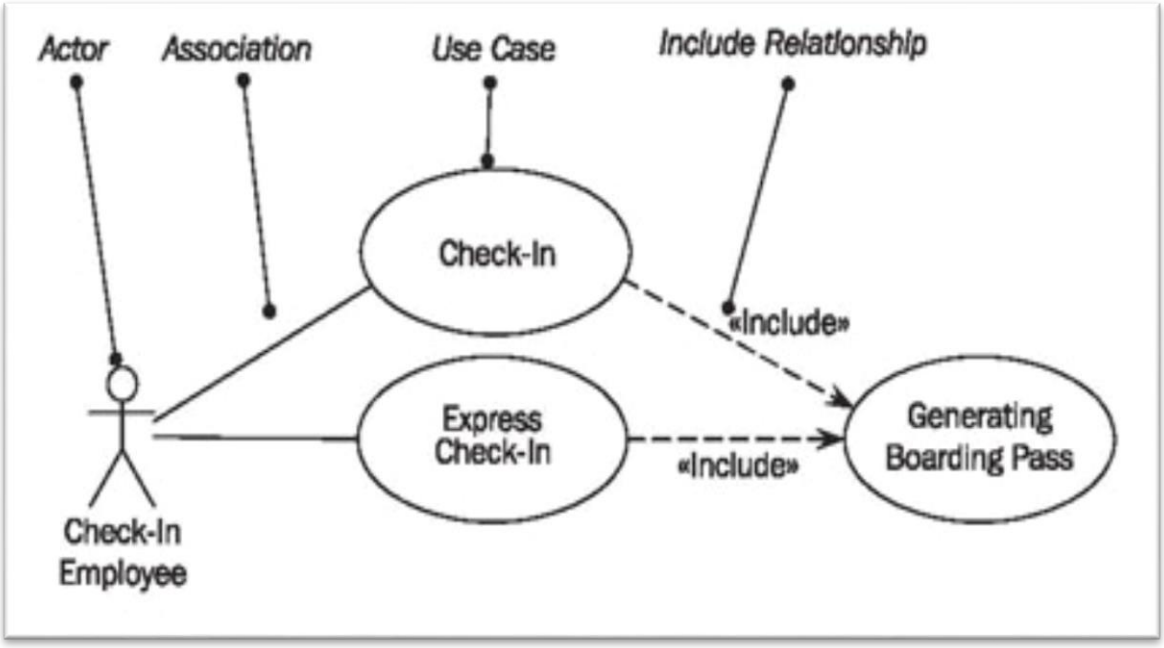
4.1. PLANIRANJE RAZVOJA SOFTVERA

Planiranje razvoja softvera uključuje prikupljanje zahtjeva za budući softver koji se oblikuje u obliku korisničke priče, nakon toga ide planiranje isporuke te kreiranje plana iteracije za svaku pojedinu iteraciju za razvoj softvera. Vrši se česta isporuka malih dijelova projekta i ističe se važnost malih dnevnih sastanaka te kretanje ljudi u projektu kao mehanizam sprječavanja gubitka znanja.

4.1.1. KORISNIČKE PRIČE

Korisničke priče (eng. *User Stories*) koriste se za kreiranje vremenske procjene isporuke softvera. Oni zamjenjuju svu papirologiju za zahtjevima koji softver mora zadovoljavati, kao što je kod klasičnih metoda razvoja softvera. Najveća razlika je u količini detalja. Korisničke priče moraju osigurati dovoljno detalja kako bi se prikazalo stvarno vrijeme potrebno za implementaciju korisničke priče. Druga razlika je u razini detalja. Kod XP-a pokušava se izbjeći velika razina detalja, kao npr. koji točno algoritam koristiti ili koju bazu podataka koristiti.

Slučajevi uporabe koriste se u grafičkom jeziku za modeliranje UML. U najjednostavnijem obliku, jedan slučaj uporabe bilježi jednu interakciju između sustava i njegove okoline, s njime se određuju akteri (npr. ljudi, sustavi) koji su uključeni u tu interakciju te joj se daje ime. Skup od više slučajeva dokumentira se uporabom jednog dijagrama slučaja uporabe (*eng.use case*)(Manger, 2013). Primjer slučaja uporabe vidimo na sljedećoj slici (slika 3).



Slika 3. Primjer Use Case dijagrama (Izvor: Sourcemaking, 2013)

Slučajevi uporabe predstavljaju skup scenarija koji sustav izvodi kako bi došao do cilja. Scenarij se može definirati kao skup koraka koje se izvode za vrijeme interakcije između sustava i samog korisnika.

Zahtjevi koje sustav mora ispunjavati mogu se podijeliti u dvije skupine: zahtjevi koji se odnose na kvalitetu usluge i funkcijski zahtjevi. Zahtjevi koji se odnose na kvalitetu usluge su pouzdanost, performansi i sigurnost samog sustava, dok funkcijski zahtjevi definiraju što će točno sustav raditi za korisnika. Kada se definira sustav on obično predstavlja crnu kutiju (eng. *Black box*) jer se samo promatra ponašanje sustava izvana.

Korisničke priče piše sam naručitelj softvera kao zahtjeve koje bi budući sustav trebao zadovoljiti. Najčešće se pišu u formatu od tri rečenice. Korisničke priče vode k stvaranju „testa prihvaćenosti“ softvera koji potvrđuje naručitelj softvera. Potrebno je najmanje jedan ili više takvih testova kako bi se uspjela dobiti potvrda da su korisničke priče na kraju ispravno implementirane. Programer preuzima od korisnika korisničke priče i na temelju njih procjenjuje koliko će trajati implementacija svake korisničke priče. Obično jedna korisnička priča zauzima dva do tri tjedna posla, ovisno o težini zahtjeva koju nosi. Ako bi implementacija trajala duže od tri tjedna za jednu korisničku

priču, trebalo bi ju razdvojiti u više manjih dijelova. Dok bi trajanje implementacije manje od jednog tjedna značilo premalu razinu detalja.

4.1.2. PLANIRANJE ISPORUKE

Plan isporuke (eng. *Release Planning*) softvera je idealna procjena trajanja svake korisničke priče tijekom jednog tjedna. Prema procjenama vremenski period od tjedan dana je dosta za implementaciju jedne korisničke priče, ako se osim toga ništa drugo ne radi. Osim implementacije testova, korisnik odlučuje koje korisničke priče su najbolje, znajući koja je važnost pojedine priče za kupca. Korisničke priče najčešće se pišu na papirima. Programeri i korisnici skupa određuju koje će korisničke priče biti implementirane u prvoj ili sljedećim isporukama. Krajnji cilj je, naravno, stabilno upotrebljiv sustav koji će biti isporučen što je ranije moguće. Brzina projekta služi da bi se otkrilo koliko se korisničkih priča može implementirati prije krajnjeg roka. Izračun vremenskog planiranja vrši se na način da se pomnoži broj iteracija s brzinom projekta da bi se otkrilo koliko se korisničkih priča može završiti odnosno implementirati. Kada se vrši planiranje po dosegu, dijeli se ukupan broj tjedna, procijenjenih za završavanje korisničkih priča, s brzinom projekta kako bi se otkrilo ukupan broj iteracija do kraja. Individualne iteracije planiraju se prije svake iteracije, a ne unaprijed. Na kraju, kada se dovrši plan isporuke, on se prosljeđuje menadžeru. Nije preporučljivo da se mijenja plan isporuke jer se s promjenom plana može uzročiti problemi u kasnijim fazama razvoja softvera.

4.1.3. MALE ISPORUKE

Nastoji se što češće isporučivati manje dijelove projekta naručitelju. Treba se dobro definirati koji dijelovi projekta bi bili pogodni za isporuku. Važno je da povratna informacija od naručitelja bude brza i kvalitetna kako bi se na vrijeme ispravile greške. Ako se softver mijenja u kasnijim fazama razvoja, cijena njegovog popravka će biti veća.

4.1.4. BRZINA PROJEKTA

Brzina projekta označava koliko se brzo posao na projektu obavlja. Faktor radnog učinka je mjera koja se koristi kao mjera brzine u projektima. On služi kako bi se kreirala početna procjena brzine projekta, nakon toga koristi se brzina projekta. On mjeri koliko je korisničkih priča implementirano ili koliko je programskih rješenja napravljeno u jednom iteraciji.

Brzina projekta može se povećati tako što programeri pitaju naručitelja da im dostavi drugu korisničku priču ako su završili ranije s jednom. Prilikom određivanja brzine projekta očekuje se da će doći do smanjenja, ali i do povećanja brzine tijekom trajanja projekta. Ako dođe do velikih oscilacija između predviđene i realne brzine trajanja projekta, potrebno je mijenjati plan isporuke. Problem kod svakog projekta je kreiranje inicijalnih procjena. Sakupljanjem detalja ne čini inicijalnu procjenu ništa drugo nego čisto nagađanje. Važnije je dobro procijeniti cijeli projekt nego napraviti opsežnu dokumentaciju.

4.1.5. ITERATIVNI RAZVOJ

Iterativni razvoj (eng. *Iterative Development*) povećava brzinu razvoja softvera. Preporučljivo je da se projekt raspodijeli u deset iteracija, jedna iteracija bi u prosjeku trebala trajati od dva do tri tjedna. Nije preporučljivo da se programski zadaci unaprijed zadaju. Potrebno je napraviti raspored iteracija i na početku svake iteracije odrediti što će se raditi u toj iteraciji. Planiranje u trenutku je korisno kako bi se mogli pratiti korisnički zahtjevi ako dođe do nekih promjena. Ne preporuča se ići unaprijed i implementirati nešto što nije isplanirano u toj iteraciji.

Trebalo bi se pridržavati planiranih rokova, ako se vidi da se iteracija ne može dovršiti u roku, trebala bi se izvršiti nova procjena iteracije, ponoviti procjene te smanjiti opseg zadataka u toj iteraciji. Primarni cilj svake iteracije je dovršavanje najvažnijih zadataka koje je korisnik procijenio. Ako se funkcionalnosti implementiraju samo onda kada je to isplanirano te ako se pomno prati planiranje, ne bi trebalo biti problema s promjenama korisničkih zahtjeva.

4.1.6. PLANIRANJE ITERACIJE

Planiranje iteracije (eng. *Planning Iteration*) izvršava se na samom početku svake iteracije. Planiraju se svi zadaci koji se moraju izvršiti unutar jedne iteracije. Korisničke priče korisnik sam određuje za trenutnu iteraciju prema planu isporuke u poretku kojeg on određuje. Korisničke priče i testovi koji nisu prošli provjeru ispravnosti ubacuju se kao programske zadaće novih korisničkih priča. Programeri popisuju nove zadatke i procjenjuju koliko je potrebno da bi se svi zadaci izvršili. Programeri trebaju procijeniti kraj zadataka. Svaka zadaća bi u prosjeku trebala trajati od dva do tri idealna programska dana. Idealan programski dan znači koliko bi vremena trebalo da se izvrši zadatak bez da ima ikakvih smetnji. Zadaci koji se mogu riješiti unutar jednog dana mogu se skupa grupirati, a zadaci kojima treba više od tri dana da se riješe trebali bi se razdijeliti u nekoliko manjih zadataka.

Brzina projekta služi kako bi se procijenilo je li iteracija preopterećena ili nije. Trajanje jedne iteracije ne bi trebalo premašiti vremensko trajanje druge iteracije. Ako postoji više korisničkih priča, korisnik treba odlučiti koje priče prebaciti za sljedeću iteraciju. Ali, ako postoji malo korisničkih priča, onda članovi tima mogu dodati dodatne korisničke priče u iteraciju. Nije poželjno dodavanje dodatnih funkcionalnosti u iteraciju jer takve stvari samo usporavaju projekte i time dolazi do zakašnjenja projekta. Ponekad je potrebno načiniti ponovnu procjenu i pregovaranje oko plana isporuke svake tri do pet iteracije.

4.1.7. KRETANJE LJUDI

Kretanjem ljudi (eng*. Move people around*) može se spriječiti ozbiljan gubitak znanja u kodiranju. Ako samo jedna osoba iz tima može jedina raditi na određenom području i ta osoba bude nedostupna na određeno vrijeme, na taj način može usporiti rad samog projekta. Kretanje ljudskih resursa za obavljanje različitih zadataka u kombinaciji s programiranjem u paru ima efekt dobrog, može pružiti dobar trening. U timu ne bi smjela samo jedna osoba sve znati ili jedna osoba biti specijalizirana samo za jedno područje nego bi sve osobe u timu trebale znati sve o projektu ili o nekim dijelovima kôda itd. Takav način rada jako je izražen u malim grupama, jer na taj način jedna osoba ne bi imala previše posla nego bi svi imali podjednako posla. Tako bi se veći broj programera mogao posvetiti važnijem dijelu implementiranja sustava.

4.1.8. DNEVNI SASTANCI

Na uobičajenim sastancima kao kod npr. klasičnih metoda, većina članova tima uopće ne sudjeluje aktivno na sastancima, nego uglavnom slušaju samo zaključke. Zato se kod ekstremnog programiranja provode tzv. „stojeći sastanci“ (eng. *Stand up Meeting*). Sastanak je prva stvar ujutro što se odvija. Na njemu se diskutiraju i rješavaju problemi. Sastanak se odvija na način da svi članovi tima stoje u krugu kako bi se izbjegle duge rasprave. Maksimalna dužina sastanka je oko deset minuta. Tako dolazi do veće efikasnosti jer su sastanci kratki i svi prisustvuju, a ne dugi sastanci na kojima ne budu svi članovi tima. Dnevni stojeći sastanci mogu zamijeniti mnogo drugih sastanaka, zbog štednje vremena koju pruža dužinom svojih sastanaka.

4.1.9. DEBUGIRANJE SOFTVERA

Normalno je za očekivati da će se tijekom razvoja softvera naći na manje, ali i na veće greške. Potrebno je slijediti postupke ekstremnog programiranja kako bi se na vrijeme započelo s debugiranjem softvera. Pravila trebaju biti slijedna, sve dok ih projektni tim ne odluči mijenjati. Programeri trebaju znati što očekivati jedan od drugih imajući već prije definiran skup pravila koji je pak jedini način za postavljanje tih očekivanja.

4.2. DIZAJN SOFTVERA

U ovom dijelu rada govoriti će se o dizajnu softvera kod ekstremnog programiranja, koji uključuje jednostavan dizajn, metafore kao pojednostavljene slike sustava, korištenje CRC karata za timski dizajn sustava, korištenje testova za isprobavanje funkcionalnosti. Preporučeno je da se ne implementiraju dodatne funkcionalnosti bez suglasnosti naručitelja.

4.2.1. JEDNOSTAVNOST

Kod metoda ekstremnoga programiranja uvijek je potrebno činiti najjednostavniju stvar koja može funkcionirati. Jednostavan dizajn je lakše za izraditi, nego neki kompleksan. Ako se tijekom implementacije naiđe na kompleksan dio, potrebno je taj dio zamijeniti nekim jednostavnim jer je uvijek brže, a i jednostavnije zamijeniti

kompleksan dio sa jednostavnim, prije nego što se izgubi dosta vremena na njemu. Time je dosta olakšano održavanje samog kôda, a samim time lakše je vršiti implementaciju novih funkcionalnosti. Potrebno je držati dizajn što duže jednostavnijim, tako da se nova funkcionalnost ne dodaje dok to nije određeno iteracijom.

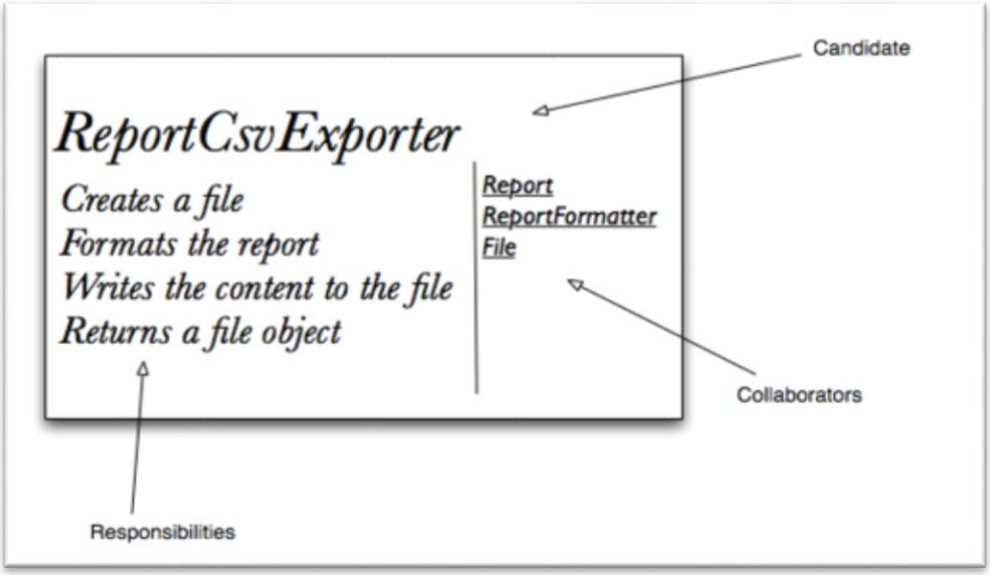
4.2.2. METAFORA SUSTAVA

Metafora sustava (eng. *System Metaphor*) odnosi se na pojednostavljenu sliku sustava koja je još u razvoju. Potrebno je sliku što prije pojednostaviti, tako da je svi članovi tima mogu razumjeti. Glavna funkcija metafore sustava je pokazati programerima jasnu sliku gdje se točno nalazi njihov dio u sustavu. To znači da programeri imaju čitav pregled pojednostavljenog sustava te im je pojednostavljena slika cijele funkcionalnosti sustava.

Metafora predstavlja zajedničku terminologiju koja se koristiti na projektu kako bi je svi mogli razumjeti. Potrebno je odabrati metaforu sustava kako bi programeri mogli konzistentno davati imena klasama, objektima, metodama itd. Na taj način štedi se veliki dio vremena ne razmišljajući koja imena davati klasama i objektima. Potrebno je davati imena koji će svi članovi tima moći povezati sa specifičnim dijelom sustava.

4.2.3. CRC KARTE

CRC karte (eng. *Class, Responsibilities, Collaboration cards*) je skraćenica za kartice klasa, odgovornosti i suradnje, a služe za timski dizajn sustava. One omogućavaju cijelom projektnom timu da se fokusira na dizajn softvera. Na slici se nalazi primjer jedne CRC karte (slika 4).



Slika 4. CRC karta (Izvor: DZone / Agile Zone, 2012)

Individualne CRC karte predstavljaju objekte. Klasa objekta može biti predstavljena na vrhu karte, odgovornosti se prikazuju u donjem lijevom kutu dok se suradnja među klasama prikazuje u desnom kutu.

CRC sesija odvija se s osobom koja simulira sustav, na način da govori sustavu koji objekti šalju kojim objektima poruku. Prolazeći kroz sustav rano se otkrivaju problemi i slabosti sustava. Jedan od najvećih mana CRC karti je nedostatak napisanog dizajna. To ne predstavlja problem ako CRC karte čine dizajn očitim, a ako je potreban zapis, treba biti dokumentirana jedna CRC karta za svaku klasu i treba je sačuvati kao dokumentaciju. CRC karte smatraju se strateškim nivoima dizajna.

4.2.4. ŠILJAK RJEŠENJE

Šiljak rješenje (eng. *Spike Solution*) je mali eksperiment ili testiranje kako bi se istražio odgovor na pitanje. Taj test je obično vrlo jednostavan program koji pomaže u prolaženju rješenja za problem s kojim se susreće razvojni tim. Izrađuje se skup jednostavnih programa koji samo adresiraju problem koji se istražuje uz ignoriranje svih ostalih radnji. Glavna funkcionalnost je samo testiranje da se vidi da li rješenja rade, a ne rješenje za cijeli sustav. Glavni cilj im je povećanje pouzdanosti korisničkih priča i smanjenje rizika tehničkog problema (Shore, 2010).

4.2.5. IZBJEGAVANJE IMPLEMENTACIJE DODATNE FUNKCIONLANOSTI

Dodatna funkcionalnost će samo usporiti implementaciju te potrošiti previše vremena. Projektni tim mora biti usmjeren na sadašnje funkcionalnosti i zahtjeve, koncentrirati se na samo ono što mu je zadano.

4.2.6. REFAKTORIRANJE

Mnogo programera koristi praksu upotrebe ponovnog kôda jer se na taj način štedi vrijeme. Umjesto pisanja taj dio kôda samo se prekopira. Zato služi refaktoriranje. Naime, refaktoriranje je tehnika poboljšavanja kôda bez da se mijenja njegova funkcionalnost. To je disciplinaran način čišćenja kôda koji minimizira šanse za uvođenjem neke nepravilnosti. Pod refaktoriranje spada uklanjanje kôda, brisanje nekorištene funkcionalnosti, te redizajn zastarjelog dizajna, kako bi kôd bio što razumljivi. Na taj način štedimo vrijeme i povećavamo kvalitetu našeg softvera.

Refaktoriranje se koristi kako bi se dizajn sustava održao čistim te da bi se izbjegla nepotrebna složenost. Ako se kôd održava čistim, omogućava se njegovo lakše razumijevanje i proširivanje. Refaktoriranje se provodi tijekom cijelog životnog ciklusa softvera, to podrazumijeva prije i poslije implementacije novog kôda, prije implementacije nove funkcionalnosti radi se kako bi kôd funkcionalnosti bio što jednostavniji. Poslije implementacije funkcionalnosti radi se kako bi se pojednostavio napisani kôd. Kako refaktoriranje ne bi narušilo ispravnu radnju novo implementirane funkcionalnosti, potrebno je svako malo provoditi testove kako bi se osigurali da nova funkcionalnost nije narušena. Refaktoriranje se više primjenjuje u kasnijim fazama razvoja jer korisnici zahtijevaju nove funkcionalnosti. Zato je potrebno razviti arhitekturu koja ima bolju mogućnost prihvaćanja novih funkcionalnosti. Tu se često javlja potreba za refaktoriranjem kako bi se uklonio dvostruki kôd koji je nastao implementacijom nove funkcionalnosti. (Folwer i sur., 1999).

Velika mana refaktoriranja je to što je vezana uz baze podataka jer se baze podataka ne mogu tako lako mijenjati. Druga mana je migracija podataka. Bez obzira što je sustav dizajniran kako bi se minimizirale ovisnosti između baza podataka i objektnog modela, promjena sheme označava migraciju podataka, što je vrlo problematičan i dug proces. Treća mana je promjena korisničkog sučelja. Unutrašnjost

objekta može se redizajnirati, a s promjenom sučelja treba biti jako oprezan, pogotovo ako korisnik koristi sučelje.

4.3. KODIRANJE I IMPLEMENTACIJA

U ekstremnom programiranju veliku ulogu igra naručitelj softvera koji mora biti što više dostupan razvojnom timu. U XP-u postoje standardi kodiranja kojeg se programeri trebaju obavezno pridržavati. Testiranje se obavezno odvija prije implementacije kôda, ali i nakon implementacije. To su samo neke od aktivnosti koje se odvijaju unutar kodiranja i implementacije kod ekstremnog programiranja. Detaljniji opis navedenih aktivnosti biti će opisani u nastavku.

4.3.1. PRISUTNOST NARUČITELJA

Kod XP-a uloga naručitelja nije samo da pomaže razvojnom timu, nego je cilj da on postane dio razvojnog tima. Kroz cijeli životni ciklus XP-a odvija se neprestana komunikacija između naručitelja i razvojnog tima. Postoje dva načina kako naručitelj može sudjelovati u razvoju softvera:

- Naručitelj radi na projektu, dodijeljeni su mi konkretni zadaci.
- Naručitelj pomaže timu kako da dođe do nekih problema, odgovarajući na nejasna pitanja ili pitanja vezana uz zahtjeve.

Stalna prisutnost naručitelja može dovesti do problema u razvoju. Ako ne može točno opisati što je sve potrebno prije nego što se kôd uopće počne pisati.

Korisničke priče stvara naručitelj uz pomoć razvojnog tima s čime nastaju vremenske procjene pri čemu im se dodjeljuju prioriteti. Svako određuje svoje obveze u planiranju:

- Naručitelj određuje datum izdavanja te koliko je važna koja korisnička priča. Na osnovnu toga određuje se prioritet korisničkih priča unutar pojedine iteracije.
- Programer procjenjuje koliko je potrebno za implementaciju pojedine korisničke priče. Na osnovu toga određuje se raspored implementacije korisničke priče unutar pojedine iteracije.

Zahtijevaju se konstante isporuke malih dijelova sustava kako bi korisnik na vrijeme vidio odgovaraju li implementirane funkcionalnosti njegovim vizijama i željama. Također je potrebna pravovremena povratna informacija od korisnika kako bi se na vrijeme ispravile greške.

Testovi prihvaćanosti provjeravaju je li sustav, ili samo dio sustava, spreman za isporuku. Naručitelj izravno sudjeluje u izradi testova prihvaćanosti, koji provjeravaju funkcionalnost sustava. Ako naiđe na tehničke probleme pri izradi, može zatražiti pomoć od programera. Može se dogoditi da sustav neće proći sve testove prihvaćenosti prije isporuke. Naručitelj treba odlučiti da će sustav ići u isporuku ili će se isporuka odgoditi dok se ne otklone sve greške i sustav prođe test prihvaćenosti.

4.3.2. STANDRADI PISANJA KÔDA

Problem može nastati što svaki programer ima svoj vlastiti stil kodiranja. Zbog toga postoje standardi kodiranja kako se ne bi previše vremena gubilo na razumijevanje tuđeg kôda. Ako se programeri u timu ne mogu dogovoriti oko stila pisanja, potrebno je preuzeti standard propisan od neutralne organizacije.

4.3.3. KODIRANJE TESTOVA PRIJE IMPLEMENTACIJE

Prije implementacije funkcionalnosti vrši se implementacija testova jer se na taj način kreiranje kôda vrši puno brže i lakše. Implementacija testa ne uzima nikakvo dodatno vrijeme, no čini implementaciju funkcionalnosti, za koju se piše test, dosta lakšom.

Uz pomoć jediničnog testa programerima se olakšava da bolje uvide što zapravo treba biti implementirano bez implementacije onog što nije potrebno ili nužno. Testovi prate zahtjeve sustava. Jedinični testovi nam odmah daju povratnu informaciju za vrijeme implementacije kôda. Čime se pokrene jedinični test zna se da je implementacija funkcionalnosti uspješno izvršena.

Ponekad je teško vršiti testiranje nekih softverskih sustava. Slučaj kod takvih sustava je da se prvo čini implementacija funkcionalnosti pa se tek onda implementiraju svi testovi. Implementacija testova prije kôda naziva se TDD (eng. *Test Driven Development*). Prvo se implementira jedan test koji definira jedan problem. Zatim se za taj test implementira jednostavan kôd kako bi prethodno implementirani

test prošao bez greške. Kada se implementira nova funkcionalnost, treba se refaktorirati prethodni kôd. Test se opet provodi da bi se vidjelo radi li nova funkcionalnost ispravno. Postupak se ponavlja dok svi zahtjevi nisu zadovoljeni ili prebačeni u kôd. Programer koji je pisao kôd ne bi trebao pisati testove, jer postoji mogućnost da je programer napravio neke greške, a testovi neće otkriti te greške. Korištenjem TDD tehnike, kôd se pojednostavljuje i brže nastaje. To se događa jer je implementirano samo ono što je potrebno i nužno. Čim se test pokrene odmah se dobiva povrtna informacija radi li novi kôd ispravno. Ujedno se ti testovi mogu koristiti kao dokumentacija projekta.

4.3.4. PROGRAMIRANJE U PARU

Svaki kôd napravljen tijekom životnog ciklusa softvera je napisan od dva programera. Programiranjem u paru povećava se kvaliteta softvera bez da se gubi dodatno vrijeme. Broj grešaka je puno manji, nego kada radi samo jedan programer jer se brže uočavaju greške na taj način. Dva programera koja rade za istim računalom mogu implementirati dvostruko više funkcionalnosti, nego dva programera koji rade sa dva računala. Na taj se način povećava sama produktivnost. Postoje dvije uloge programera u paru:

a) programer koji piše kôd
b) programer koji provjera kôd.

Jedan programer tipka i razmišlja taktički o kôdu, dok drugi programer strateški razmišlja o kôdu i kako se on uklapa u pojedinu klasu.

Ne programiraju uvijek iste osobe u paru, nego članovi tima kruže kako bi proširile svoje iskustvo. Poželjno je da osobe u parovima budu sličnih znanja i vještina kako bi se izbjeglo da jedna osoba radi dok se druga odmara.

4.3.5. KONTINUIRANA INTERGRACIJA KÔDA

Ako se izvorni kôd ne kontrolira, programeri ne mogu znati da kôd nema grešaka. Zbog paralelne integracije izvornog kôda, može postojati kombinacija kôda koja još nije bila testirana. Zbog toga može doći do velikih problema bez pravovremenog reagiranja. Drugi problem je ispravnost jediničnih testova. Ako se ne mogu osigurati

kompletni i ispravni testovi, mogu postojati i zabilježiti se lažne neispravnosti i propustiti prave neispravnosti u kôdu.

Za rješavanje problema koristi se tzv. jednonitna integracija (eng. *Single Theaded*) koju vrši programer, sam u kombinaciji s zajedničkim vlasništvom nad kôdom. Kôd se prebacuje u zajednički repozitoriji izvornog kôda koji omogućuje samo jednom paru programera da testira i radi promjene na izvornom kontrolnom repozitoriju kôda.

Obično se to radi nekoliko puta u danu. Kontinuiranom integracijom sprječavaju se greške ako su promjene u kôdu značajne. Nekada zna doći problema u komunikaciji između programera o tome koji se dijelovi kôda mogu ponovno upotrijebiti, a koji dijelovi kôda se mogu dijeliti. Svaki programer mora biti u stanju raditi s posljednjom verzijom. Također, promjene bi se smjele raditi samo u zadnjoj verziji. Zadnja verzija kôda se stavlja i dohvaća sa zajedničkog repozitorija izvornog kôda. Mogu se koristiti repozitorij kao što su:

a) Rational ClearCase
b) Microsoft Visual Source Safe
c) Concurrent Versions System

4.3.6. ZAJEDNIČKO VLASNIŠTVO NAD KÔDOM

Zajedničko vlasništvo nad kôdom znači da su svi članovi tima odgovorni za taj kôd. Cijeli tim je odgovaran za arhitekturu sustava. Arhitektura sustava je raspodijeljena u projektnom timu. Svaki član ima određene odgovornosti nad arhitekturnim odlukama. Svaki programer je dužan kreirati jedinične testove uz kôd koji je napisao jer na takav način funkcionira zajedničko vlasništvo nad kôdom. Testovi znaju biti velika pomoć drugim programerima jer pokazuju kako se određeni dio kôda koristi i kada se kôd mijenja, izvršavanje testova ukazuje na promjenu funkcionalnosti.

Svi kôdovi i testovi trebaju biti pohranjeni na zajedničkom repozitoriju. Programeri koji su pisali kôd često kasnije ni ne primijete da je kôd promijenjen. Učinkovitije je zajedničko vlasništvo nad kôdom nego dodjeljivanje odgovornosti nad pojedinim klasama određenim osobama u timu.

4.3.7. OPTIMIZACIJA KÔDA

Optimizacija kôda obavlja se na samom kraju razvojnog projekta. Ne preporučuje se raditi optimizaciju za vrijeme trajanje implementacije kôda. Optimizacija kôda radi se kako bi se određeni dijelovi kôda zamijenili naprednijim dijelovima koji su jednostavniji i vremenski se brže izvode ili zamijeniti pojedine algoritme bržim algoritama. Ali, optimizacija se tek radi ako se ustanovi da sustav prije toga normalno funkcionira.

4.3.8. 40 SATNI RADNI TJEDAN

Mnogo projektnih timova danas rade greške, umjesto da pomno isplaniraju isporuku kako bi je uskladili s vremenskim zahtjevima, mnogo timova se odluči na prekovremene kako bi projekt bio završen u roku, što im mnogo puta neće poći za rukom. Ekstremno programiranje ima svoju radnu politiku koja se zove 40 satni radni tjedan koji zahtjeva od članova tima da budu odmorni ujutro i puni novih ideja, a na kraju radnog dana da budu umorni, ali zadovoljni sa svojim radom. Menadžeri bi tu svoje prste trebali imati da radnicima osiguraju dobru radnu okolinu, ne zamarati ih s administrativnim poslovima i ne dopustiti prekovremeni rad (Maržić, 2005).

4.4. TESTIRANJE SOFTVERA

Kod ekstremnog programiranja testovi se mogu podijeliti u dvije skupine:

1. Jedinični testovi – provjeravaju da li sustav radi. Daju sigurnost programeru u vlastiti kôd te objašnjavaju kako se taj kôd koristi. Jedinični testovi moraju biti automatizirani kako bi se ubrzalo testiranje kôda.
2. Testovi prihvaćenosti – provjerava se funkcionalnost čitavog softvera koji se temelji na korisničkim pričama. Određuje zadovoljava li softver kriteriji prihvaćenosti i omogućuje naručitelju da odluči prihvaća li softver kao takav. Kreira ga naručitelj uz pomoć programera.

Testiranje se provodi skoro kroz sve faze razvoja softvera. Ako programer napravi i najmanju promjenu na kôdu, odmah se provodi testiranje kako bi se potvrdilo

radi li program ispravno. Zbog toga je važno da testovi budu automatizirani kako bi se omogućilo njihovo stalno ponavljanje.

4.4.1. TESTIRANJE JEDINICE PROGRAMA

Jedinični testovi su jedan od najvažnijih stvari kod ekstremnog programiranja. Da bi se jedinični testovi uspješno primijenili, potrebno je kreirati ili koristiti gotova rješenja koja se nalaze na tržištu. Kent Beck i Erich Gamma napravili su JUnit za J2SE i J2EE platformu. Uz pomoć ovih alata, omogućeno je automatizirano izvršavanje jediničnih testova iz IDE (integrirano razvojno okruženje).

Potrebno je testirati sve klase, ali nije potrebno testirati sve metode u objektnom sustavu. Nije potrebno testirati metode s kojima se postavlja ili dohvaća neka vrijednost objekta ili varijable. Testovi se trebaju kreirati prije pisanja samog kôda. Jedinični testovi stavljaju se u zajednički repozitoriji skupa s datotekama izvornog kôda. Ako se otkrije da nema testa za neku metodu ili klasu, jedinični test treba obavezno biti naknadno napisan. Automatizirani testovi su najbolje rješenje protiv evidentiranih neispravnosti. Naime, čim je test teži za implementirati, biti će potrebniji, jer će donositi veće uštede. Veća je korist od automatiziranih testova nego troškovi njihovih izrada.

Jedinični testovi omogućuju zajedničko vlasništvo nad kôdom. Jer oni zahtijevaju da sva funkcionalnost radi ispravno prije nego što se kôd premjesti u zajednički repozitoriji. Vlasništvo nad kôdom nije potrebno ako su sve klase pokrivene kôdom. Oni također omogućuju refaktoriranje. Nakon svake promjene kôda, jedinični testovi provjeravaju da nova struktura nije narušila nikakvu funkcionalnost. Kreiranjem univerzalnih jediničnih testova za validaciju i regresijsko testiranje omogućuje se česta integracija. Moguće je brzo integrirati bilo kakvu nedavnu promjenu pokrećući niz testova. Popravljanje malih greška je lakše i brže svako malo raditi, nego popravljanje velikih grešaka u zadnji tren. Automatiziranjem jediničnih testova moguće je sjediniti skup promjena sa zadnjom verzijom u kratkom vremenu.

Dodavanjem novih funkcionalnosti, automatski se dolazi do mijenjanja jediničnih testova kako bi testovi odgovarali funkcionalnostima. Ponekad se zna dogoditi da kôd bude ispravan, a testovi sadrže neke nepravilnosti, ali to se lako otkrije pokretanjem testova. Kreiranje testova koji nisu zavisni o kôdu prije samog nastajanja kôda postavlja veliku vjerojatnost da će kôd raditi kada bude napisan.

4.4.2. PRONALAŽENJE NEISPRAVNOSTI

Čim se naiđe na neispravnost sustava, kreiraju se testovi kako bi se spriječilo dolaženja do takve neispravnost. Evidentirana neispravnost zahtjeva da bude napisan test prihvaćenosti softvera kojeg potvrđuje naručitelj. Ako se dobije test prihvaćenosti koji nije prošao, programeri moraju napisati jediničan test kako bi sagledali problem iz kuta specifičan izvornom kôdu. Kada svi jedinični testovi prođu, znači da je greška otklonjena i može se ponovno pokrenuti test prihvaćenosti koji bi trebao pokazati ako je sve uredu, da je neispravnost popravljena.

4.4.3. PONAVLJANJE TESTA PRIHVAĆENOSTI

Testovi prihvaćenosti softvera, koje potvrđuje naručitelj, nastali su tj. kreirani iz korisničkih priča. Tijekom iteracije, korisničke priče koje su odabrane tijekom isporuke biti će prevedene u testove prihvaćenosti. Korisničke priče mogu imati jedan ili više testova prihvaćenosti, s obzirom koliko je potrebno da bi se provjerilo radi li funkcionalnost. Testovi prihvaćenosti testiraju sustav kao crnu kutiju. Apstrakcija crne kutije odnosi se na vidljivost implementacije sustava iza sučelja sustava. Naručitelji ne moraju znati što se točno nalazi iza sučelja ili specifikacije. Svaki test prihvaćenosti predstavlja neki očekivani rezultat sustava. Naručitelj je odgovoran za kontrolu i ispravnost testova prihvaćenosti te je zadužen za ocjenjivanje rezultata testiranja.

Korisnička priča smatra se završenom tek onda kada je prošla test prihvaćenosti. Test prihvaćenosti mora biti kreiran u svakoj iteraciji. Također bi svi testovi prihvaćenosti trebali biti automatizirani tako da se što češće pokreću. Rezultati testova dostavljaju se razvojnom timu. Na razvojnom timu je da osmisli vremenski raspored u svakoj iteraciji kako bi popravili sve moguće greške (Hunt, Thomas, 2003).

5. ZAKLJUČAK

U današnjem svijetu tehnologije teško je izabrati kojim putem, kao programer krenuti, od izbora programerskog jezika pa sve do metode razvoja softvera. Konkurencija je sve veća i jača pa treba pažljivo odabrati odgovarajuće alate i metode potrebne za izradu softvera.

U ovom radu prikazana je metodologija ekstremnog programiranja. Opisuju se potrebni koraci kao i redoslijed kojim se ti koraci moraju izvršiti kako bi se stvorio uspješan softverski proizvod.

Prije samog početka rada potrebno je osigurati prava sredstva i ljude za uspješno provođenje XP metode. Potrebno je imati na raspolaganju stručne i iskusne programere koji znaju raditi u timovima. Potreban je iskusan menadžer koji se također treba razumjeti u kodiranje, koji će znati objasniti i predstaviti proizvod kupcima te koji će moći uputiti svoje nadređene o tijeku i načinu razvitka softvera. Potreban je vođa tima koji mora biti iskusan programer jer on pomaže ostalim članovima tima ako oni zapnu bilo gdje. Njegova glavna zadaća je da projektom timu osigura da što kvalitetniju radnu atmosferu.

Programiranjem u paru smanjuje se ovisnost o pojedincu i njegovom znanju, ali povećava šansa za ispravljanjem grešaka, što znatno smanjuje troškove poslovanja. Programeri su usredotočeni na kodiranje, a ne gube vrijeme na izradu dokumentacije i održavanje dugih sastanaka. No, u timu mogu postojati osobe koje nisu naviknule na timski rad i ne mogu funkcionirati u takvom okruženju. Uvođenjem jednostavnosti i jediničnih testova ubrzava se izrada softvera, uz što manji broj grešaka zbog stalnog testiranja, nakon svake nove promjena na softveru. Ako dođe do promjena zahtjeva od strane korisnika, to ne predstavlja problem za XP zbog svoje kontinuirane integracije. Zato je potrebno je detaljno planiranje od početaka projekta pa sve do najsitnijih promjena koje se rade na softveru kako bi se unaprijed mogli procijeniti troškovi poslovanja.

Zbog svojih istaknutih prednosti, zaključio sam da bi ekstremno programiranje moglo koristiti sve više malih kompanija jer XP način rada omogućava niske troškove poslovanja. Projektni tim sastoji se maksimalno od 10 ljudi zbog čega je više prilagođen malim tvrtkama jer za korištenje takve metode nije potrebno mnogo ljudi.

Ciljevi rada su dokazani, definirani su pojmovi agilnih metoda razvoja softvera i ekstremnog programiranja, objašnjena je cijela metodologija ekstremnog programiranja, od prikupljanja zahtjeva, dizajna pa sve do implementacije i testiranja softverskog proizvoda.

LITERATURA

[1] Algebra otvoreno učilište, 2014., Agilni razvoj, Dostupno na: http://www.algebra.hr/agilni-razvoj. [Pristupljeno: 23.07.2016.]

[2] Ekonomski fakultet u Zagrebu, 2009., Trendovi u programiranju, Dostupno na: http://web.efzg.hr/dok//inf/pozgaj/pisani%20materijali/T07%20Trendovi%20u%20programiranju.pdf. [Pristupljeno: 23.07.2016.]

[3] Folwer M i sur., lipanj 1999., Improving the Design of Excisting Code, Addison Wesley Professional

[4] Gavrić Ž., 2012., Pregled metodologija i tehnika za razvoj softvera za generisanje asp.net web formi kao primjera dana-driven programiranja, Dostupno na: http://fit.spu.ba/test/wp-content/uploads/2014/09/DiplomskiRadZeljkoGavric.pdf. [Pristupljeno: 29.07.2016.]

[5] Hunt A. i Thomas D., rujan 2003., Pragmatic Unit Testing, The Pragmatic Programmers,

[6] Manger R., 2013., Softversko inženjerstvo, Prirodoslovno matematički fakultet u Zagrebu, Zagreb

[7] Maržić K., 2005., Prilagodba Ekstremnog programiranja za projekt razvoja javne električne usluge, Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, Zagreb

[8] McBreen P., srpanj 2002., Questioning Extreme Programming, Addison Wesley Professional

[9] Pap R., 2008., Ekstremno programiranje kao metod agilnog razvoja softvera, Univerzitet u Novom Sadu, Prirodno-matematički fakultet, Novi Sad

[10] Radovan A. i sur., 2010., Prijedlog podučavanja agilnih metoda razvoja softvera korištenjem alata Microsoft Project Veleučilište Velika Gorica, Velika Gorica

[11] Shore J., 2010., The Art of Agile. Dostupno na: http://www.jamesshore.com/Agile-Book/spike_solutions.html. [Pristupljeno: 03.09.2016.]

[12] Tadić B., 2005., Ekstremno programiranje i primjena na Balkanu, Dostupno na: http://www.quality.unze.ba/zbornici/QUALITY%202005/034-Q05-028.pdf.

[13] Zekić-Sušac M., 2013., Pristup izradi poslovnih aplikacija - Agilne metode razvoja aplikacija. Dostupno na: http://www.efos.unios.hr/razvoj-poslovnih-aplikacija/wp-content/uploads/sites/228/2013/04/RPA_P1_Agilne-metode1.pdf. [Pristupljeno: 23.07.2016.]

POPIS SLIKA

POPIS TABLICA

SAŽETAK

Agilne metode razvoja softvera kao pojam označavaju nekoliko iterativnih i postupnih metodologija za razvoj softvera. Neke od njih su Scrum, Lean i Adaptivni razvoj. Jedna od najpoznatijih metoda je Ekstremno programiranje (XP). Temeljni cilj XP je zadovoljavanje ili zadovoljstvo kupca ili korisnika softvera. Razvojna metodologija obuhvaća prikupljanje zahtjeva, dizajn softvera, implementaciju i testiranje softvera. Karakteristike XP su; malene, ali česte isporuke, svaka dva do tri mjeseca, kako bi se na vrijeme mogle ispraviti sve greške, korištenje najjednostavnih rješenja bez kompliciranog kôdiranja i dodatnih funkcionalnosti. Kontinuirano testiranje nakon svake nove implementacije. Dva programera pišu kôd, tako da jedno drugog mogu ispravljati, 40 satni radni tjedan i korisnik je uvijek mora biti na raspolaganju razvojnom timu. Dokumentacija se ne piše nego testovi služe kao dokumentacija softvera. Timski rad dolazi do velikog isticanja. Kupci, menadžeri, programeri, oni su svi dio tima koji proizvodi softver.

Ključne riječi: Agilan razvoj softvera, Ekstremno programiranje, programiranje u paru, testiranje

SUMMARY

Agile methods of software development as a term can be defined as several iterative and incremental methodologys for software development. Some of them are Scrum, Lean and Adaptive development. One of the most popular methods is Extreme Programming (XP). One of the fundamental goals of XP is the satisfaction of customer's needs. Development methodology includes gathering requirements, software design, implementation and testing of software. Characteristics of XP are; small, but frequent deliveries, every two to three months, in order to correct all errors in time, the use of the simplest solutions without complicated the code or adding additional functionality. Continuous testing after each new deployment. Two programmers write the code, so they can correct each other, 40 hour work week and the user must always be available to the team. The test's of the software are used as documentation. Teamwork leads to big highlight. Customers, managers, developers, they are all part of the team that produces the software.

Key words: Agile methods of software development, Extreme Programming, pair programming, softver testing