POSLOVNI I PRAVNI FAKULTET - BEOGRAD

DIZAJN SOFTVERA U EKSTREMNOM PROGRAMIRANJU

- PREDAVANJA 2 -

Predavač: Prof. dr Borivoje M. Milošević

2022

Osnovni koncepti i metodologije

Planiranje razvoja softvera uključuje akcije prikupljanja zahteva koji se oblikuju u:

- ✓ korisničke priče (engl. *User Stories*),
- ✓ planiranje isporuke na nivou čitavog projekta,
- ✓ kreiranje plana iteracijâ za svaku pojedinu iteraciju u iterativnom razvoju softvera

Ovde je naglašena i uključena važnost čestih i malenih isporuka i važnost merenja brzine projekta (engl. *Project Velocity*) za vršenje planiranja

Takodje, ističe se važnost dnevnih stojećih sastanaka (engl. *Daily Stand Up Meeting*) i spomenuta praksa kretanja ljudi (engl. *Move people around*) u projektu kao mehanizam sprečavanja gubitka znanja.

Osnovni koncepti i metodologije

Korisničke priče:

Korisničke priče (engl. *User Stories*) služe istom cilju kao i slučajevi upotrebe (engl. *Use Cases*), ali zapravo nisu isto. Korišćene su, u prvom redu, kako bi se kreirale vremenske procene za planiranje isporuke softvera (engl. *Release Planing*). Takodje, njihovo značenje je i zamena velikih dokumenata sa zahtevima koje softver treba zadovoljavati.

Slučajevi upotrebe koriste se za modeliranje zahteva koje sistem mora ispunjavati. Zahtevi koje sistem mora ispunjavati se mogu podeliti u dva glavna skupa:

- ✓ funkcijski zahtevi (engl. *functional requirements*) i
- ✓ zahtevi koji se odnose na kvalitet usluge (engl. *quality of service requirements*).

Osnovni koncepti i metodologije

Korisničke priče:

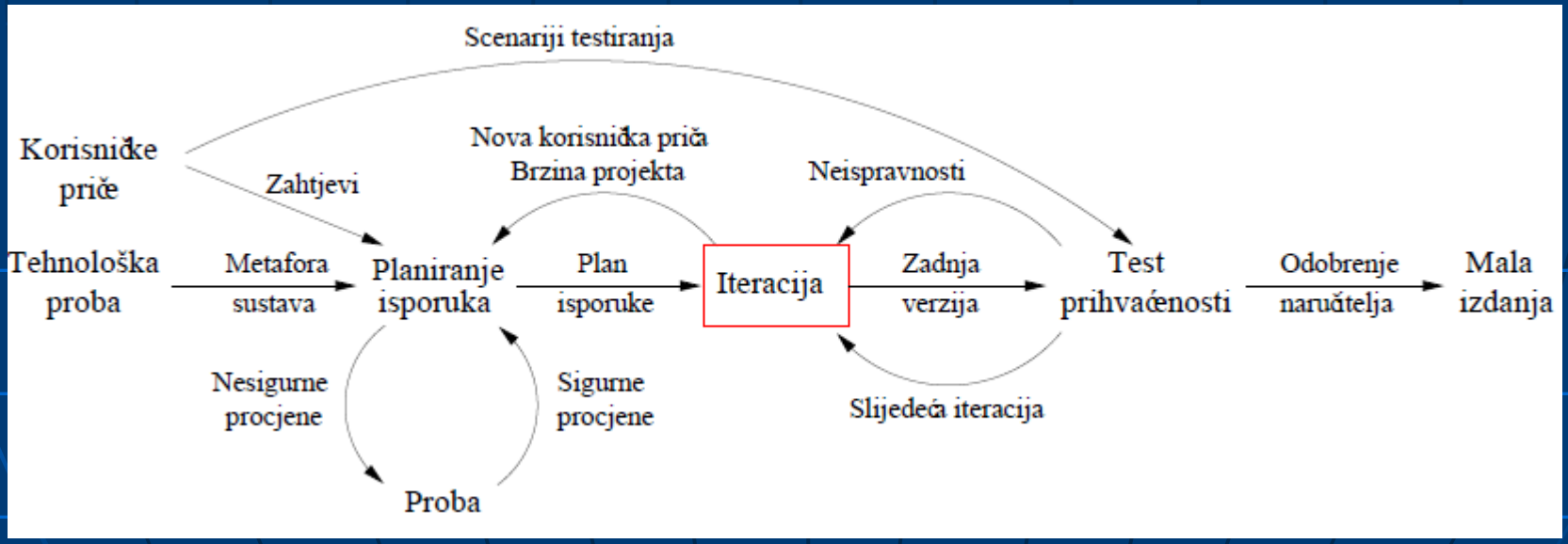
Slučajevi upotrebe definišu ponašanje sistema ili dela sistem i predstavljaju skup scenarija koji sistem izvodi kako bi postigao neki cilj. Scenarij je skup koraka koji se izvode za vreme interakcije izmedju korisnika i sistema. Funkcijski zahevi definišu šta će sistem raditi za korisnika. Kada se definišu, sistem se obično predstavlja kao crna kutija (engl. *black box*) jer se samo gleda ponašanje sistema spolja.

Zahtevi koji se odnose na kvalitet usluge definišu performanse, pouzdanost i sigurnost sistema. Primena slučajeva upotrebe ima za cilj modeliranje željenog ponašanja sistema, bez potrebe specificiranja načina implementacije takvog ponašanja. Tipičan proces modeliranja nekog sistema počinje detaljnom specifikacijom slučaja upotrebe, a nakon toga se prelazi na realizaciju (definisanje klasâ od kojih se sastoji i potrebnih interakcijskih dijagrama).

Osnovni koncepti i metodologije

Korisničke priče piše naručilac (engl. *customer*) softvera kao zahteve koje sistem (softverski sistem) treba zadovoljiti. Slične su korisničkom scenariju, jedino što nisu limitirane na opis korisničkog interfejsa (engl. *user interface*). Obično su u formatu od oko tri rečenice teksta u terminologiji korisnika bez tehničke sintakse. Dakle, po svojim karakteristikama, drugačije su od slučajeva upotrebe.

Korisničke priče vode kreiranju tzv. testa prihvaćenosti softvera kojeg potvrdjuje korisnik (engl. *Acceptance Test*). Potrebno je kreiranje jednog ili više automatiziranih testova prihvaćenosti softvera kako bi se verifikovale korisničke priče, tj. proverilo da li su korisničke priče ispravno implementirane.



Osnovni koncepti i metodologije

Jedno od najvećih nerazumevanja sa korisničkim pričama je kako se priče razlikuju od tradicionalnih specifikacija zahteva, odnosno slučajeva upotrebe. Najveća razlika je u stepenu detalja. Korisničke priče trebaju osigurati dovoljno detalja kako bi se načinila razumna procena relativno niskog rizika koja prikazuje koliko dugo će trajati implementacija te korisničke priče. Kada dodje vreme za implementaciju priče, programeri će doći do naručioca i primiti detaljan opis zahteva.

Programer procenjuje koliko dugo (vremenski) će trajati implementacija pojedine korisničke priče. Svaka korisnička priča će oduzeti jednu, dve ili tri nedelje procene u "idealnom vremenu" razvoja. To idealno vreme razvoja zapravo znači koliko dugo treba za implementaciju korisničke priče u kôd, ukoliko nema nekih prepreka, drugih obveza pa je tačno poznato šta i kako treba činiti.

Druga razlika izmedju korisničkih priča i dokumenta zahteva, odnosno slučajeva upotrebe je fokus na potrebe naručioca. Poželjno je pokušati izbeći detalje specifične tehnologije, osnove baze podataka i algoritme. Poželjno je pokušati držati korisničke priče fokusirane na potrebama naručioca kao suprotnost specifikaciji izgleda grafičkog interfejsa.

Osnovni koncepti i metodologije

Planiranje isporuke:

Korisničke priče služe kako bi se kreirao plan isporuke koji vredi za čitav projekt. Plan isporuke se, kada je donešen, koristi za kreiranje plana iteracijâ za svaku pojedinu iteraciju

Planiranje isporuke (engl. *Release Planning*) softvera sadrži skup pravila koje omogućuju svima uključenim u projekat da naprave vlastite odluke. Skup pravila definše metodu

oko pregovora vremenskog plana koga svako iz tima može ispuniti.

Suština planiranja isporuke za razvojni tim je idealna procena trajanja svake korisničke priče u nedeljama. Idealno, nedelja je vremensko trajanje implementacije korisničkih priča ako se apsolutno ništa drugo ne čini, jedino uključujući implementaciju jediničnih testova. Naručilac tada odlučuje koje korisničke priče su najvažnije ili imaju najviši prioritet da bude završene, znajući koja je važnost pojedine priče za kupca.

Osnovni koncepti i metodologije

Krajnji cilj je upotrebljiv, testabilan sistem koji će biti rano isporučen (čim je moguće ranije). Brzina projekta je kreirana kako bi se odredilo koliko korisničkih priča može biti implementirano pre datog datuma (roka) ili koliko dugo traje implementacija skupa korisničkih priča.
Kad se vrši vremensko planiranje, treba pomnožiti broj iteracija sa brzinom projekta da bi se odredilo koliko korisničkih priča može biti završeno. Kad se vrši planiranje po dosegu, potrebno je podeliti ukupan broj nedelja procenjenih korisničkih priča sa brzinom projekta kako bi se odredio ukupan broj iteracija do kraja.

Na postupak nastajanja planiranja isporuke utiču zahtevi i metafora sistema (engl. *System Metaphor*). Nepouzdane i nesigurne procene (engl. *Uncertain Estimates*) mogu biti popravljenje odredjenim tehnološkim probama (engl. *Spike*). Sigurne, tj. Pouzdane procene (engl. *Confident Estimates*) ponovo utiču na planiranje isporuke.

Planiranje isporuke rezultuje planom isporuke (engl. *Release Plan*) koji vredi za čitav projekat pa se koristi za kreiranje plana iteracijâ za svaku pojedinu iteraciju. Kreiranje novih korisničkih priča (engl. *New User Stories*) unutar pojedinih iteracija ponovo utiče na planiranje isporuke i donošenja novog plana isporuke.

Osnovni koncepti i metodologije

Kvantifikovanje projekta sa četiri promenljive:

Osnovna filozofija planiranja isporuke je da projekat može biti kvantifikovan sa četiri Promenljive:

- ✓ domet,
- ✓ resursi,
- ✓ vreme i
- ✓ kvalitet.

Promenljive predstavljaju dimenziju projekta.

- ✓ Domet se odnosi na kvantitet; tj. koliko toga treba načiniti.
- ✓ Resursi su u značenju broja dostupnih ljudi.
- ✓ Vreme se odnosi na trenutak kad su projekat ili isporuka gotovi.
- ✓ Kvalitet je pokazatelj koliko je softver dobar i koliko dobro će biti testiran.

Gledajući sveukupno, potrebno je kontrolisati barem dve od četiri promenljive u bilo kom projektu, kako bi se projekt odvijao pod nadzorom.
Menadžment može samo odabrati tri od četiri projektne promenljive kako bi upravljao i planirao, dok razvoj uzima ostalu četvrtu promenljivu. Smanjenje kvaliteta manje od izvanredan (engl. *excellent*) ima nepredvidljivi uticaj na ostale tri promenljive. Zbog toga, u osnovi, postoje samo tri promenljive koje se zapravo žele menjati, izuzeti kvalitet.

Osnovni koncepti i metodologije

Male isporuke:

Zadatak razvojnog tima je često isporučivati iterativne verzije sistemaa naručioca. Kod planiranja isporuke je potrebno definisati malene jedinice funkcionalnosti koje su pogodne za isporuku i koje mogu biti isporučene u okruženje naručioca rano u projektu (u ranim fazama projekta).

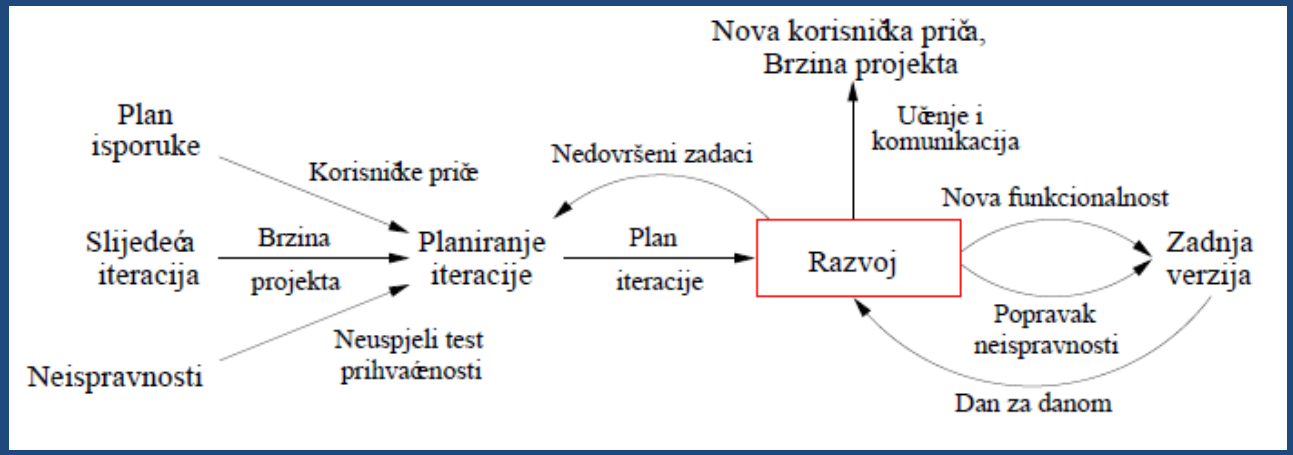
Kritična potreba je dobijanje vredne i kvalitetne povratne informacije od korisnika kako bi se na vreme imao bolji uticaj na dalji razvoj sistema. Cena promene softvera je veća u kasnijim fazama projekta (implementaciji, testiranju i isporuka) nego u početnim fazama projekta kada je malo toga implementirano. Što se više čeka isporuka važnih obeležja korisnicima, biće manje vremena za njihov popravak.

Osnovni koncepti i metodologije

Brzina projekta:

Brzina projekta (engl. *Project Velocity*) je mera koliko brzo posao može biti napravljen na projektu. Faktor radnog učinka (engl. *load factor*) je korišćen kao mera brzine u projektima sve do nedavno. Ali, brzina projekta je jednostavnija mera za korišćenje od faktora radnog učinka. Ako pomaže, može se koristiti faktor radnog učinka kako bi se kreirala početna procena brzine projekta. Nakon toga se može koristiti brzina projekta umesto faktora radnog učinka.

Za merenje brzine projekta, može se brojiti koliko korisničkih priča ili koliko programerskih zadataka je završeno u iteraciji. Za vreme kreiranja plana isporuke, brzina projekta u završenim korisnčckim pričama može biti korišćena kako bi se procenilo koliko još korisničkih priča može biti dovršeno do nekog vremena. Za vreme kreiranja plana iteracijâ, dozvoljeno je da programeri potpisuju isti broj procenjenih dana za obavljanje programerskih zadataka koji je jednak brzini projekta merenoj u prethodnoj iteraciji.

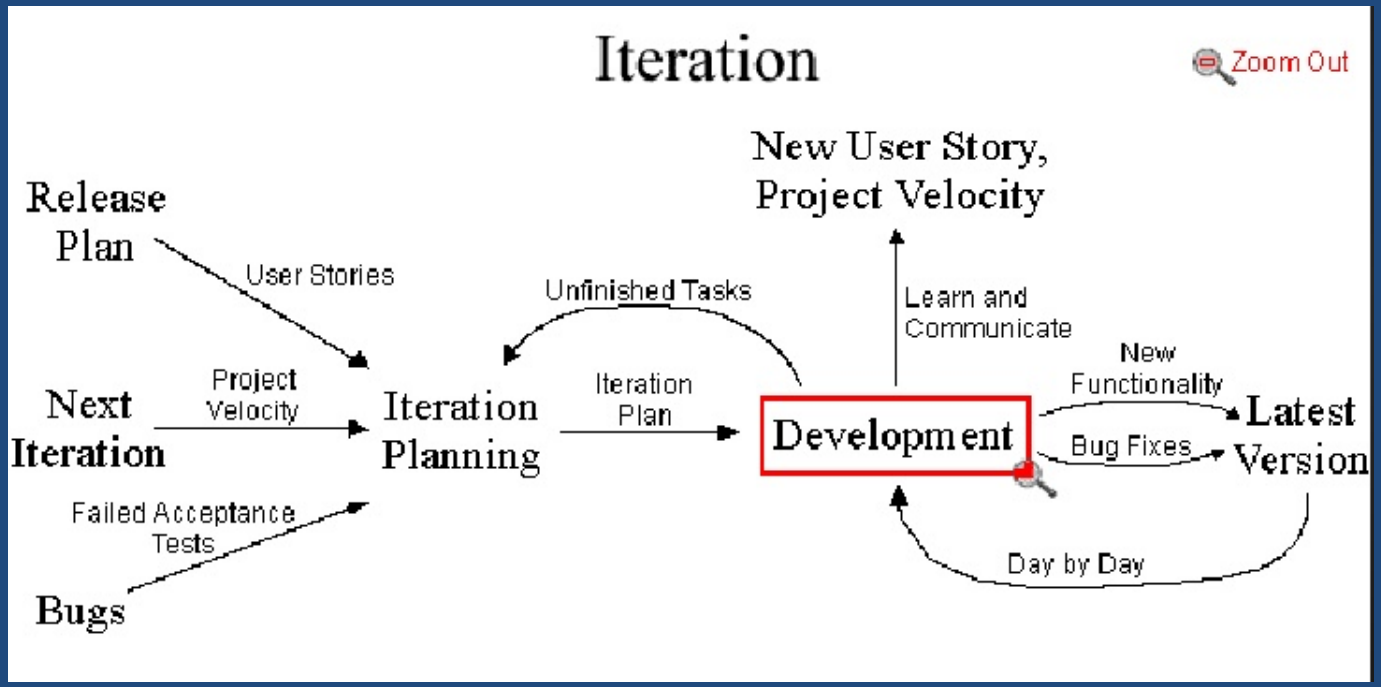


Osnovni koncepti i metodologije

Iterativni razvoj:

Iterativni razvoj (engl. *Iterative Development*) povećava brzinu razvoja. Razvoj softvera kod XP metodologije se može podeliti u raspored u oko desetak iteracija, a iteracija traje u proseku od jednne do tri nedelje. Poželjno je držati dužinu iteracija konstantnom tokom čitavog projekta jer su iteracije zapravo žila kucavica projekta.

XP adresira i paralelni razvoj. Ako je razvojni projekt razmerno veliki, tj. u njemu sudeluje veći broj osoba (npr. 30 do 40 programera), poželjno je podeliti tim u dva ili više manjih timova.



Osnovni koncepti i metodologije

Iterativni razvoj:

Svaki manji podtim (grupa) dobija vlastite korisničke priče od naručioca. Budući da je osnovna ideja podela korisničkih priča na manje timove, u pravilu ne bi trebalo biti većih problema u sinhronizaciji medju grupama.

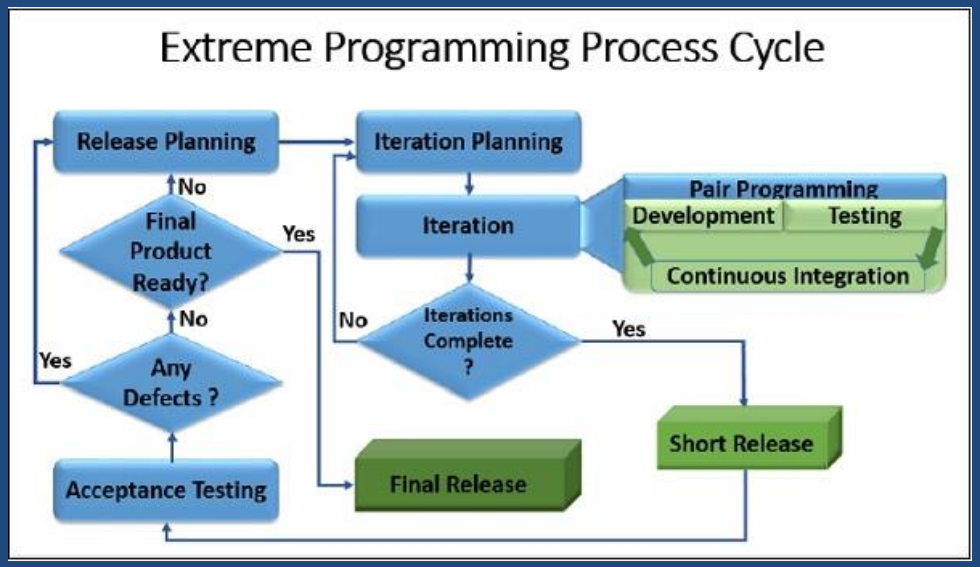
Nije dobro podeliti programerske zadatke (kodiranje) unapred. Umesto toga, potrebno je načiniti plan iteracijâ i planirati iteracije na početku svake od njih, kako bi se odredilo šta će se činiti u trenutnoj iteraciji. Planiranje u trenutku (engl. *just-in-time planning*) je jednostavan način za praćenje promena korisničkih zahteva u projektu. Takodje, nije preporučeno gledanje unapred i implementiranje onoga što nije predvidjeno u trenutnoj iteraciji. Biće dosta vremena za implementiranje i funkcionalnosti kada ona postane deo jedne od sledećih korisničkih priča u nekoj od sledećih iteracija.

Osnovni koncepti i metodologije

Planiranje iteracija:

Planiranje iteracije (engl. *Iteration Planning*) se vrši na početku svake iteracije i ima cilj proizvesti plan izvršavanja zadataka koje će se izvršavati u toj iteraciji. Obično je svaka iteracija u dužini od jedne do tri nedelje. Korisničke priče naručilac odabira za trenutnu iteraciju skladno planu isporuke u poretku koga sâm vrednuje. Neispravnost testa prihvaćenosti softvera koga potvrdjuje narucilac, a koji treba biti popravljen, je takodje povezan sa planiranjem iteracije. Narucilac odabere korisničke priče zajedno sa procenama o brzini projekta iz prethodne iteracije.

Korisničke priče i testovi koji nisu prošli proveru ispravnosti se ubacuju kao programerski zadaci u obliku novih korisničkih priča. Dok su korisničke priče u jeziku naručioca, zadaci su u jeziku programera. Duplirani zadaci mogu biti maknuti iz iteracije.



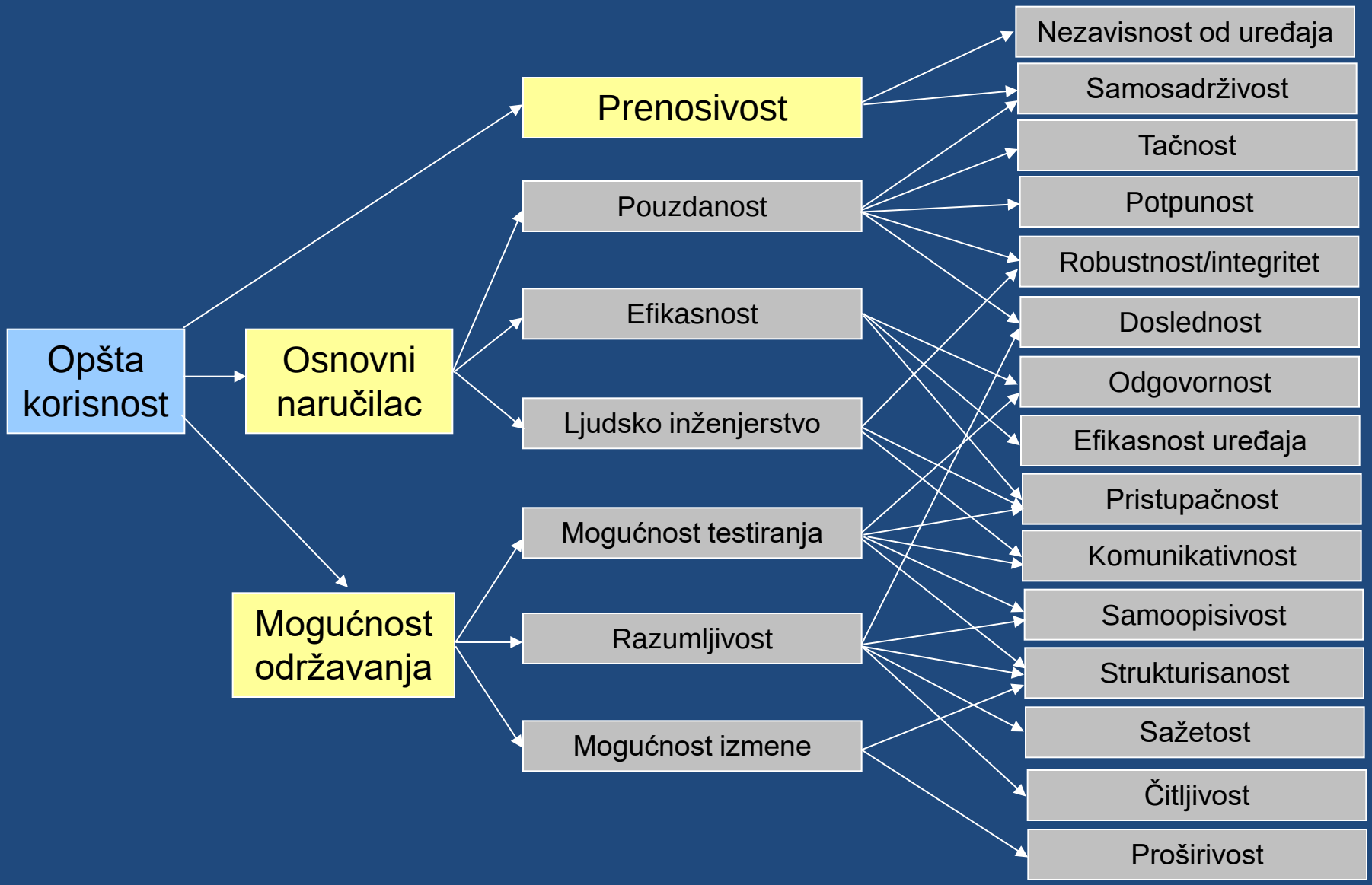
Osnovni koncepti i metodologije

Planiranje iteracija:

Razvijatelji projekta potpisuju zadatke i procenjuju koliko je potrebno da bi se ti zadaci kompletirali. Važno je za razvijatelje koji prihvataju zadatke da oni budu ti koji će proceniti kraj zadatka. Unutar XP-a ne preporučuje se zamena ljudi koji sudeluju na razvoju. Odredjena osoba koja će implementirati zadatak mora proceniti koliko dugo će trajati implementacija.

Brzina projekta je korišćena kako bi se odredilo da li je iteracija preopterećena i rezervisana ili nije. Celokupna vremenska procena u idealnim programerskim danima ne bi smela premašiti brzinu projekta iz prethodne iteracije. Ako iteracija sadrži previše korisničkih priča, naručilac treba odabrati priče (na osnovi prioriteta) koje moraju biti maknute sve do sledeće iteracije.

Boehm-ov model kvaliteta



Osnovni koncepti i metodologije

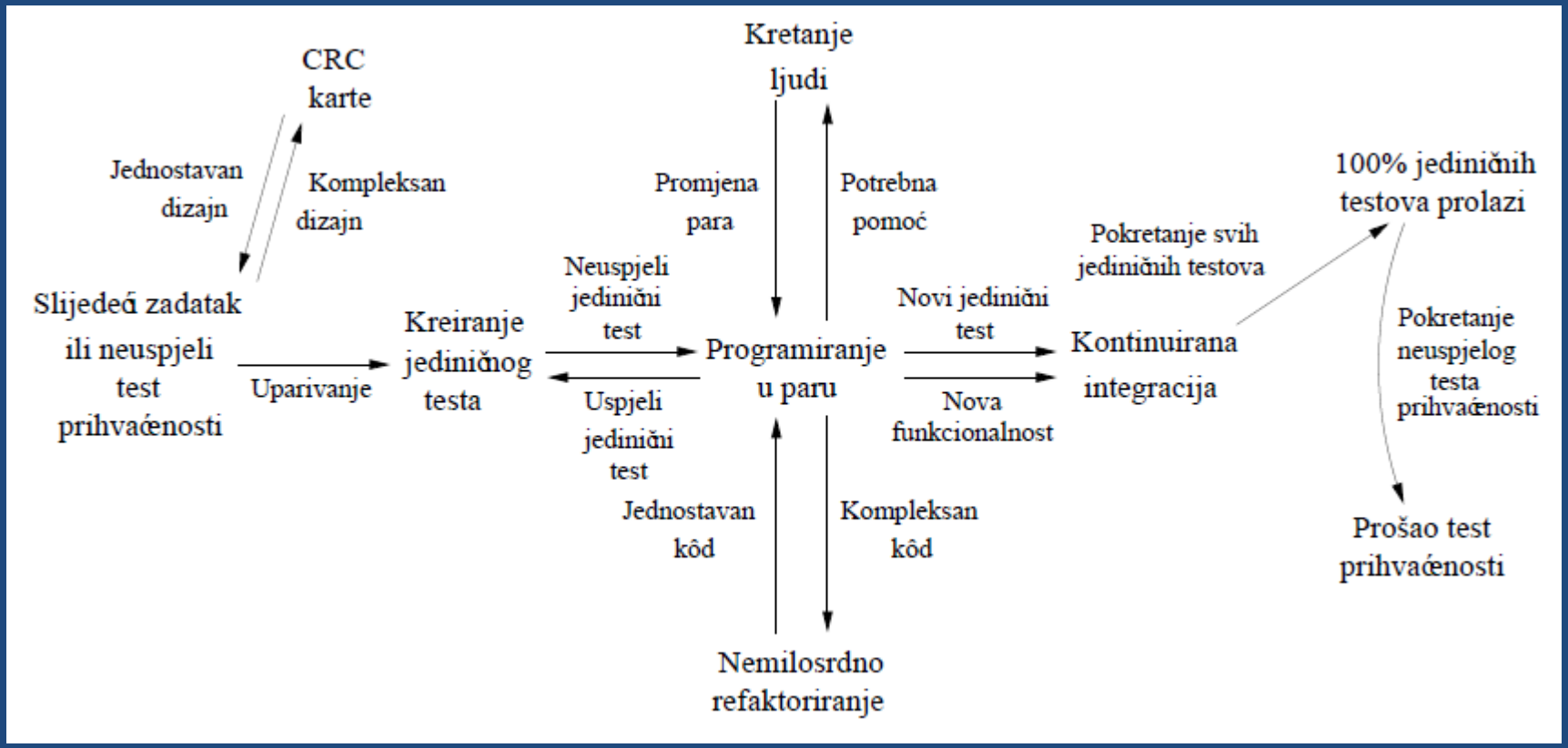
Kretanje ljudi:

Kretanjem ljudi (engl. *Move people around*) se može sprečiti ozbiljan gubitak znanja i uska grla u kodiranju. Ako samo jedna osoba u projektnom razvojnom timu može raditi na zadatom području i ta osoba je nedostupna, napredak projekta će biti veoma spor. Medjusobno treniranje ljudi je ćesto važna činjenica za razmatranje u kompanijama koje nastoje izbjeći tzv. "ostrva znanja" koji su osetljivi na gubljenje. Kretanje ljudskih resursa u obavljanju različitih zadataka u kombinaciji sa programiranjem u pâru (engl. *Pair Programming*) ima efekat dodatnog treninga. Umesto jedne osobe koja zna sve oko odredjenog dela kôda ili tehnologije, svi u timu znaju mnogo o svim delovima.

Tîm je fleksibilniji ako svako zna dovoljno raditi na svakom delu sistema. To je naročito izraženo u malim grupama. Umesto da postoji nekoliko ljudi prezauzetih sa poslom dok ostali članovi tima imaju manje posla, celi tim može biti produktivan. Bilo koji broj programera može biti dodeljen trenutno najvažnijim delovima sistema.

Dobra praksa je jednostavno ohrabrivati svakog pojedinca da pokušava delovati na novom delu sistemaa i to na barem nekom delu svake iteracije. Tehnika programiranja u pâru to čini mogućim bez gubitka produktivnosti i omogućuje kontinuiranost. Jedna osoba iz pâra može biti zamenjena dok druga nastavlja sa novim partnerom. Ovo je izvrstan način deljenja znanja u timu.

Osnovni koncepti i metodologije



Slika prikazuje zajedničko vlasništvo kôda i kretanje ljudi koje je neposredno povezano sa tehnikom programiranja u pâru. U slučaju da je potrebna pomoć, dolazi do promene pâra i stvaranja novog.

Osnovni koncepti i metodologije

Dnevni stojeći sastanci:

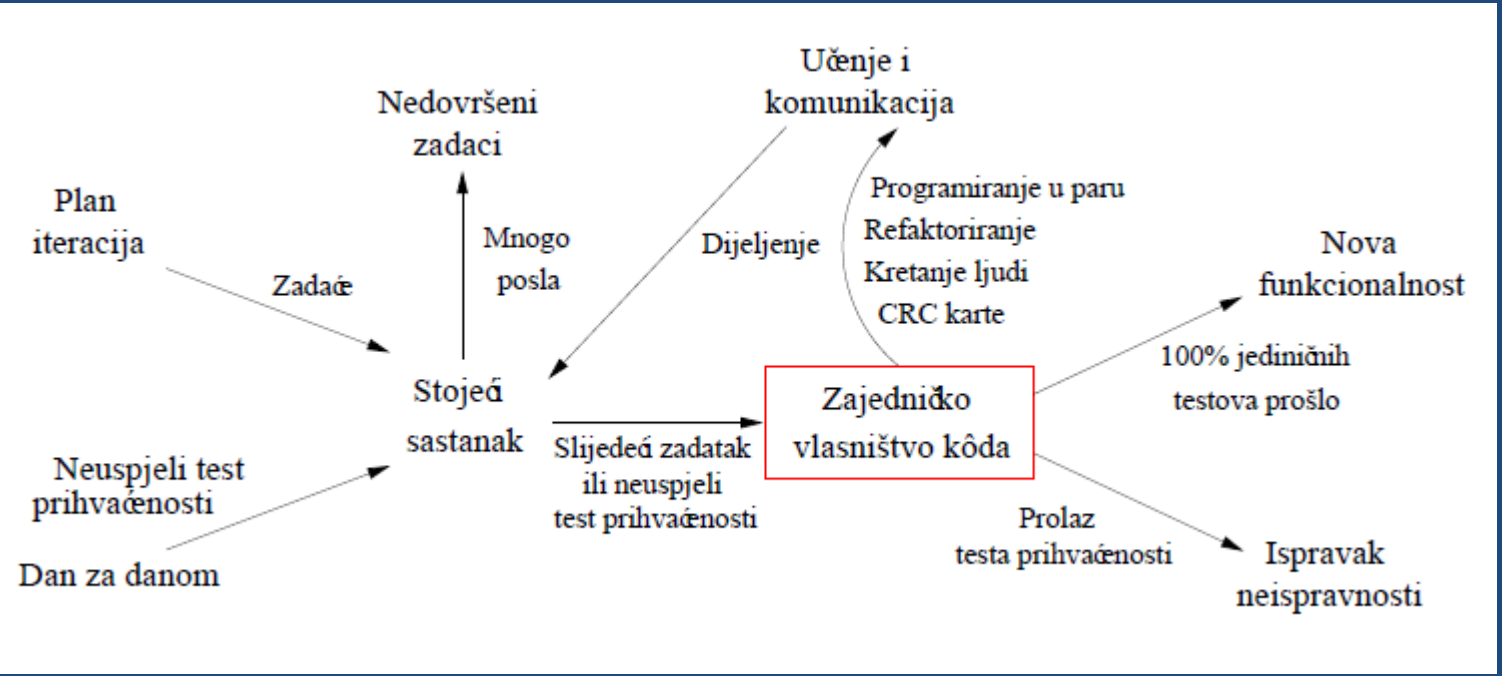
Na tipičnom projektnom sastanku većina osoba obično aktivno ne sudeluje već samo sluša zaključke. Veliki deo vremena računarskih programera je potrošen na korišćenje trivijalne količine komunikacije. U slučaju da mnogo osoba prisustvuje svim sastancima, to ima za uzrok iscrpljivanje projektnih resursa i stvaranje projektne "noćne more".

Komunikacija medju celim timom je cilj stojecćh sastanaka. Stojeći sastanci (engl. *Stand Up Meeting*) svakog jutra služe kako bi se komunicirali problemi i rešenja i unapredio timski fokus. Svi stoje "u krugu" kako bi se izbegle duge diskusije. Sastanak se i zove "stojeći" kako bi se osigurala njegova kratkoća (desetak minuta) a da sastanak ne preraste u neku dugačku raspravu. Efikasnije je da postoji kratak sastanak kome su svi obvezni prisustvovati, nego mnogo sastanaka sa delom ljudi.

Kada se koriste dnevni stojeći sastanci, auditorij bilo kojeg drugog sastanka može biti odabran prema tome ko treba sudelovati. Uz održavanje dnevnih stojećih sastanaka, moguće je ne koristiti čak većinu ostalih sastanaka.

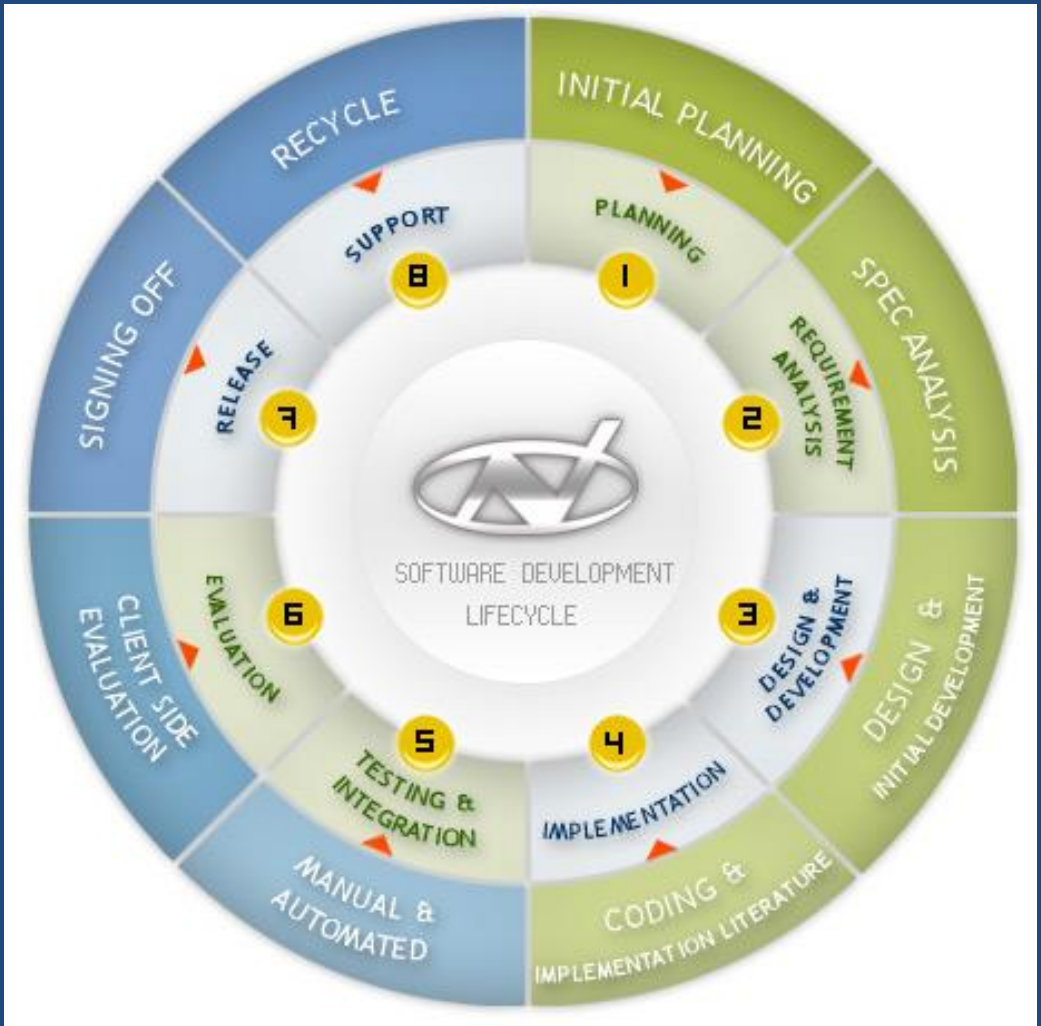
Sa limitiranim auditorijem, većina sastanaka može biti spontano zamenjena sa računarima gde se može pregledavati kôd i rešavati problemi. Dnevni stojeći sastanak može zameniti mnogo drugih sastanaka, pružajući čistu štednju nekoliko puta svojom vlastitom dužinom.

Osnovni koncepti i metodologije



Slika prikazuje tipične aktivnosti dnevnih stojećih sastanaka. Prema planu iteracija, najavljuju se zadaci za tekući dan. Dobijaju se izvještaj o testovima prihvaćenosti softvera, a koji nisu prošli. Takodje se izveštava ako je od prethodnog dana ostalo nezavršenih zadataka i ako je bilo previše toga za testirati, implementirati i refaktorisati. Tim dobija nove zadatke ili popravak stvari iz prethodnog dana. Vredi pravilo zajednčkog vlasništva nad kôdom. Zajedničko vlasništvo nad kôdom obuhvata programiranje u pâru, "nemilosrdno" refaktorisanje i kretanje ljudi unutar tima, čime se postiže učenje i komunikacija.

Dizajn softvera u ekstremnom programiranju

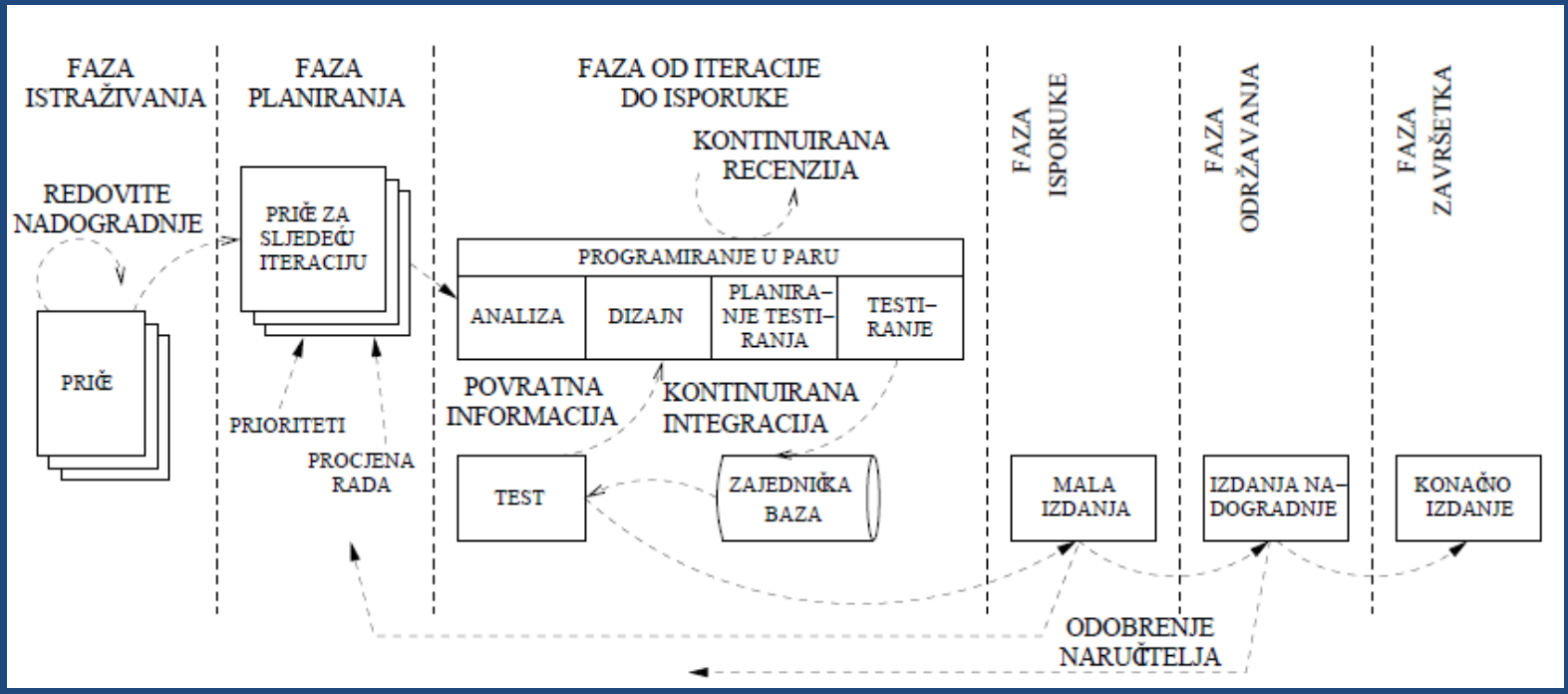


Dizajn softvera u ekstremnom programiranju

Dizajn softvera u XP-u uključuje korišćenje jednostavnog dizajna, korišćenje metafore kao pojednostavljene slike sistema koje je u razvoju, korištenje CRC algoritama za timski dizajn sistema, korišćenje rešenja "šiljka (tj. tehnoloških proba).

Naglašena je važnost izbegavanja dodavanja nepotrebne funkcionalnosti pre nego

što je ta funkcionalnost zaista nužna ili dogovorena za implementaciju pa i važnost upotrebe prakse refaktorisanja kao metode koja obuhvata akcije poboljšanja kôda. Životni ciklus XP metode prikazan je na slici.



Dizajn softvera u ekstremnom programiranju

Jednostavnost:

Jednostavan dizajn uvek zahteva manje vremena za završetak nego kompleksni. Tako je uvek potrebno činiti najjednostavniju stvar (tj. najjednostavniju moguću) koja može funkcionirati. Ako se nadje neka kompleksna materija u implementaciji, poželjno je da se ona zameni sa jednostavnijom. Jednostavan dizajn je jedan opšti princip koji vredi i za ostale razvojne metodologije.

Uvek je brže i jeftinije zameniti kompleksan kôd jednostavnijim pre nego što se izgubi previše vremena na njega. Razlog tome je lakše održavanje sâmog kôda, članovi projekta se lakše snalaze i lakše je vršiti implementaciju novih funkcionalnosti.

Stvari je potrebno držati jednostavnijim što je duže moguće, tako da se nova funkcionalnost ne dodaje dok to nije propisano iteracijom. Potrebno je biti svestan da je vodjenje jednostavnog dizajna zapravo težak posao.

Dizajn softvera u ekstremnom programiranju

Metafora sistema:

Metafora sistema (engl. *System Metaphor*) predstavlja pojednostavljenu sliku sistema koji je u razvoju, tj. u implementaciji. Važno je da ta slika sustava bude čim više pojednostavljena kako bi je svi razumeli.

Glavna ideja ove pojednostavljene slike je da programeri koji sudeluju u razvoju imaju jasnu sliku gde se njihov deo nalazi u sistemu, tj. kako se njihov deo uklapa u sistem. Metafora pojednostavljenom slikom pomaže modeliranje sistema. Korišćenjem metafore, programeri imaju i pojednostavljen pregled čitavog sistema pa im je moguće imati pojednostavljenu sliku celokupne funkcionalnosti.

Metafora definiše zajednčki jezik, tj. terminologiju koja se upotrebljava na projektu i koju svi razumeju. Potrebno je odabrati metaforu sistema kako bi tîm konzistentno davao imena klasama i metodama. Davanje imena objektima je vrlo važno za razumevanje celokupnog dizajna sistema i tehniku ponovnog iskorišćavanja kôda. Velika ušteda vremena je znati kako bi nešto moglo biti imenovano u sistemu. Nužno je odabrati imena objektima koji svi u timu mogu povezati sa specifčcnim delom sistema.

Dizajn softvera u ekstremnom programiranju

CRC karte:

CRC karte (engl. *Class, Responsibilities, Collaboration cards*) (karte klasâ, odgovornosti i saradnje) je potrebno koristiti za timski dizajn sistemaa. CRC karte omogućuju celokupnom projektnom timu da se usredsredi na dizajn. Što je više ljudi koji mogu pomoći u dizajnu sistema, biće veći broj uključenih dobrih ideja.

Individualne CRC karte se koriste kako bi predstavljale objekte. Klasa objekta može biti predstavljena na vrhu karte, odgovornosti prikazane na donjoj levoj strani a saradnja medju klasama desno od svake odgovornosti.

CRC sesija se odvija sa nekom osobom koja simulira sistem, govoreći koji objekti šalju poruke kojim objektima. Prolazeći kroz sistem, rano se otkrivaju slabosti i problemi. Alternative dizajna mogu biti brzo istražene simulirajući predloženi dizajn.

Jedna od najvećih kritika CRC karti je nedostatak napisanog dizajna. To obično nije potrebno ako CRC karte čine dizajn očitim. Ako je potreban zapis, treba biti dokumentirana jedna karta za svaku klasu i sačuvati je kao dokumentaciju. CRC karte mogu biti smatrane strateškim nivoom dizajna, programiranje u pâru taktičnim nivoom dok refaktorisanje služi i kao strateški i kao taktički nivo dizajna.

Dizajn softvera u ekstremnom programiranju

Rešenje tehnoloških proba:

Rešenje tehnoloških proba (engl. *Spike Solution*) pomaže u pronalaženju odgovora na teške probleme iz tehnike ili dizajna. To rešenje je zapravo vrlo jednostavan program koji pomaže u otkrivanju potencijalnog rešenja za problem sa kojim se razvojni tim ili pâr susrće.

Potrebno je izgraditi sistem, tj. skup jednostavnih programa koji samo adresiraju problem koji se istražuje i ignorišju sve ostale poslove.

Većina tehnoloških proba nisu dovoljno dobre kako bi se zadržale u sistemu, tako da je opšte ocekivanje da se odbace.

Njihova osnovna namena je samo proba da li odredjeno rešenje radi, a ne celokupno rešenje dizajna. Cilj je, dakle, ili smanjenje rizika tehničkog problema ili povećanje pouzdanosti procena korisničkih priča.

Kada tehnička poteškoća preti razvoju sistemaa, poželjno je uključiti pâr programera da rešavaju problem.

Dizajn softvera u ekstremnom programiranju

Refaktorisanje:

Programeri računara se zadržavaju na dizajnu softvera i dugo nakon što je softver stvoren. Praksa je da se nastavlja sa korišćenjem i sa ponovnim korišćenjem kôda (engl. *Code Reuse*) koji dugo nije održavan jer još uvek taj kôd nekako radi i strah ga je menjati kako se ne bi nešto poremetilo. Postavlja se pitanje da li je ekonomski isplativo tako činiti. Ekstremno programiranje metodologija razvoja softvera daje odgovor na ovu problematiku.

Refaktorisanje (engl. *Refactoring*) je tehnika poboljšavanja kôda bez promene funkcionalnosti, tj. spoljnjeg ponašanja kôda. To je discipliniran način čišćenja kôda koji minimizira šanse uvodjenja neispravnosti.

Tehnika refaktorisanja uključuje uklanjanje redundancija kôda, elimininiranje nekorišćene funkcionalnosti i pomladjivanje zastarelog dizajna što kôd čini lakšim za razumevanje. Refaktorisanje tokom čitavog životnog ciklusa razvojnog projekta štedi vreme i povećava kvalitet. Refaktorisanje nije pisanje kôda iz početka. Moguće su situacije kada je sigurnije i bolje početi iz početka; bolje je poboljšati postojeći kôd nego ući u rizik ponovnog pisanja kôda.

Potreba za refaktorisanjem raste sa starošću arhitekture ali i sa novim funkcionalnostima koje korisnici traže. Nove funkcionalnosti mogu biti dodate bilo kada u kôd.

Kodiranje softvera u ekstremnom programiranju

Kodiranje softvera u XP-u naglašava:

- ✓ važnost prisutnosti i dostupnosti naručioca razvojnom timu
- ✓ važnost postojanja standarda (tj. dogovora) pisanja kôda
- ✓ važnost implementacije testova pre implementacije sâmog kôda,
- ✓ opis tehnike programiranja u pâru i
- ✓ provodkenje kontinualne integracije kôda.

Ističe se politika zajedničkog vlasništva nad kôdom, provodjenje optimizacije kôda na kraju (engl. *Optimize Last*) i praktikovanje 40-satne radne nedelje, bez prekovremenog rada (engl. *40 Hours Week, No Overtime)*.

Kodiranje softvera u ekstremnom programiranju

Stalna prisutnost naručioca u razvoju:

Jedan od nekoliko zahteva XP-a je prisutnost, odnosno dostupnost naručioca, ne samo kako bi pomogao razvojnom timu u radu, već kako bi on bio deo razvojnog tima.

Sve faze XP projekta zahtevaju komunikaciju sa naručiocem, često licem u lice. Najlakše je stoga stalna prisutnost, odnosno dostupnost naručioca razvojnom timu, pogotovo ako je projekt zametne veličine i složenosti.

Postoje dva osnovna načina kako naručioc može sudelovati u timu:

• naručilac nešto radi u projektu (ima dodeljene zadatke)
• naručiolac pomaže timu u izradi softvera odgovarajćci na nejasnoće u trenutku kada su nastupili problemi.

Stalna prisutnost naručioca u projektu može rezultirati odredjenim problemima u projektu, naročito ako je naručioc uključen u tehnički deo implementacije funkcionalnosti ili nije u mogućnosti verno opisati šta je sve potrebno pre nego što je kôd napisan.

Kodiranje softvera u ekstremnom programiranju

Standardi pisanja kôda:

Programeri u razvojnom timu su često u prilici da gledaju i proučavaju kôd koji su drugi iz tima napisali. Potencijalni problem je što svaki programer ima svoj vlastiti stil kodiranja.

Važno je da postoji sporazum, odnosno pravila ili standard kako pisati kôd (engl. *Coding Standars*). U protivnom, troši se previše vremena na razumevanje tudjeg kôda.

Standardi kodiranja ne smeju oduzimati previše programerskog vremena. Ako se članovi XP tima ne mogu dogovoriti oko pravila pisanja kôda, najbolje je preuzeti neki standard propisan od neutralne organizacije.

Takav standard može biti u vidu dogovora ili u obliku pisanog dokumenta ako je većeg opsega.

Kodiranje softvera u ekstremnom programiranju

Kodiranje testova za jedinice programa prije implementacije funkcionalnosti:

Kada se prvo vrši implementacija testova (pre implementacije odred jene funkcionalnosti), kreiranje sâmog kôda će biti mnogo lakše i brže. Vreme koje je potrebno da se implementira prvo test a zatim kôd koji zadovoljava test je slično vremenu koje je potrebno za implementiranje funkcionalnosti bez implementacije testa. Implementacij testa, dakle, ne odnosi neko dodatno vreme, no čini implementaciju funkcionalnosti za koju se piše test dosta lakšom. Ako već postoji jedinični test, nije ga potrebno kreirati nakon implementacije funkcionalnosti, čime se štedi vreme.

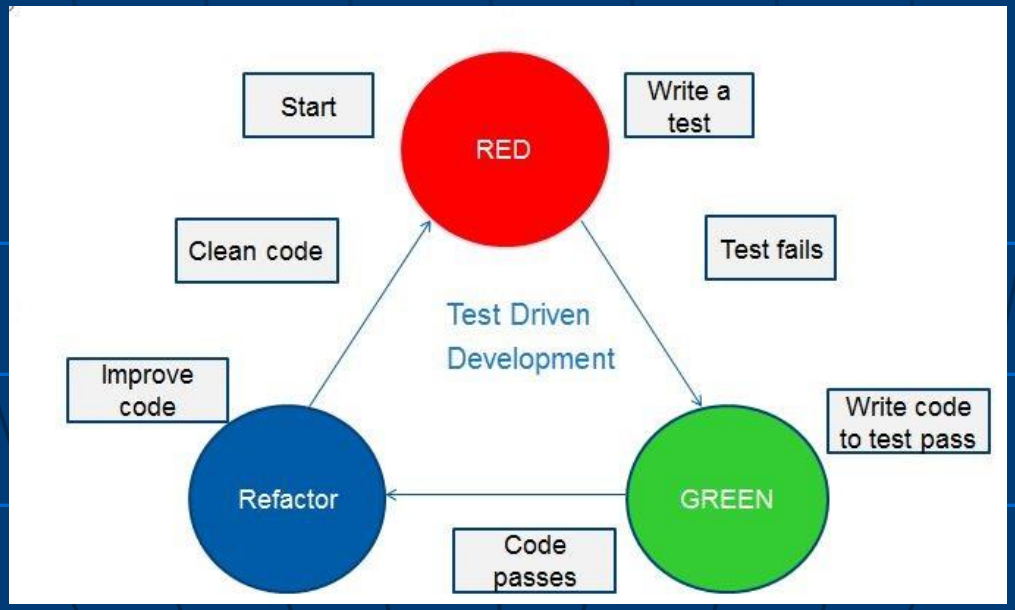
Kreiranje jediničnog testa pomaže programerima da zaista razmotre što zapravo treba biti implementirano bez implementacije onoga što nije nužno ili što nije izričiti zahtev. Testovi prate zahteve sistema. Korišćenjem jediničnih testova dobija se odmah povratna informacija za vreme rada, tj. implementacije kôda. Često nije jasno kada je programer završio, tj. implementirao svu zahtevanu funkcionalnost. Kada se kreiraju jedinični testovi, tačno se zna trenutak kada je implementacija funkcionalnosti završena time što je pokretanje testa bilo uspešno, tj. svi jedinični testovi su prošli.
Pristupom implementacije testova pre sâmog kôda vrši se veliki utecaj na dizajn sistema.

Često je vrlo teško vršiti testiranje nekih softverskih sistema. Kod takvih sistema se prvo čini klasična implementacija funkcionalnosti pa se tek onda implementiraju testovi u vidu jediničnih testova.

Kodiranje softvera u ekstremnom programiranju

Postoji ritam u razvoju softvera koji nalaže da se vrši implementacija jediničnih testova pre implementacije sâmog kôda. Taj pristup je poznat kao razvoj diktiran testiranjem (engl. *Test Driven Development*, TDD).

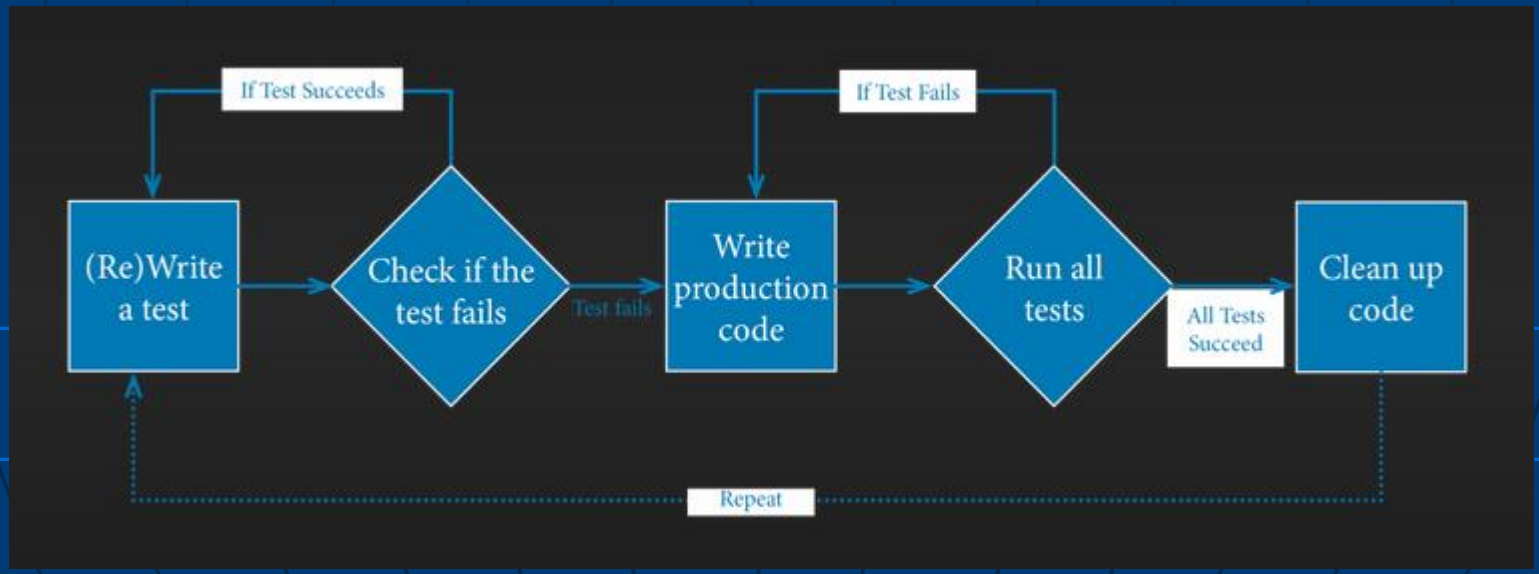
Prvo se implementira jedan test (jedna ili više testnih metoda) koji definiše jedan mali aspekt problema. Za taj test (ili testove) se implementira najjednostavniji kôd koji će omogućiti da prethodno implementirani test prodje bez grešaka. Prilikom dodavanja nove funkcionalnosti, prethodno se prema potrebi refaktoriše prethodni kôd. Test se neprestano izvodi kako bi se proverilo da ta nova funkcionalnost radi, i da se refaktorisanjem nije narušila neka prethodna funkcionalnost. Ciklički se ponavlja dodavanje testnih metoda, refaktorisanje kôda i dodavanje nove funkcionalnosti sve dok svi zahtevi nisu pretočeni u testove i kôd.



Kodiranje softvera u ekstremnom programiranju

Mogući su odredjeni problemi kada isti programeri koji pišu kôd pišu i jedinične testove za taj kôd. Moguća je situacija da programeri proizvedu odredjene neispravnosti i pri tome sa jediničnim testovima ne detektuju te neispravnosti.

Korišćenjem TDD tehnike obično brže nastaje kôd. Zabeleženo je da je takav kôd često jednostavniji nego kada se ne primenjuje ova tehnika. Implementirano je samo ono što je nužno i specificirano zahtevima. Pokretanjem testova se odmah dobiva povratna informacija da novi kôd radi.



Kodiranje softvera u ekstremnom programiranju

Programiranje u pâru:

Sav kôd koji će biti uključen u produkciju je kreiran od dva programera koji rade u pâru za jednim računartem (engl. *Pair Programming*).

Programiranjem u pâru se povećava kvalitet softvera bez uticaja na vreme isporuke. Neka istraživanja su pokazala da kôd koji nastaje od pâra programera je znatno višeg kvaliteta od kôda koji ne radi pâr, dakle nastaje od pojedinca Broj evidentiranih neispravnosti je obično manji kod programiranja u pâru jer dva programera obično brže uoče pogreške nego što ih uoči samo jedan programer.

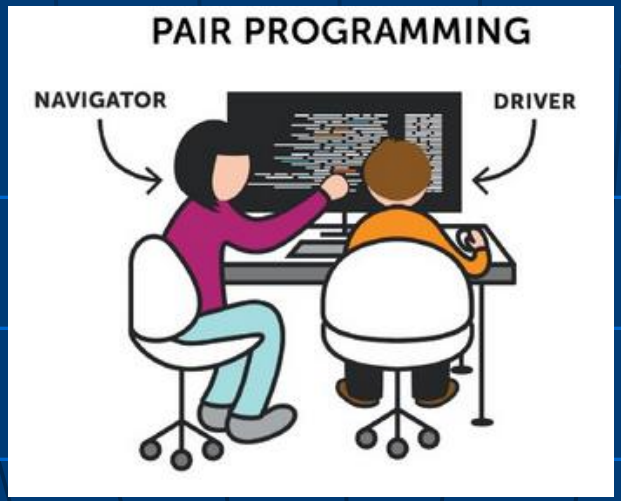
Martin Fowler u svojoj knjizi navodi značaj pregledanja tuđeg koda: ***"… treba da imate male grupe za pregledanje koda. Iz iskustva savetujem da jedan konsultant i autor originalnog koda zajedno rade na kodu. Konsultant predlaže izmene, pa obojica odlučuju da li se izmene lako mogu uneti refaktorisanjem. Ako je tako, prave izmene... Ova ideja o pregledanju koda je dovedena do krajnjih granica u slučaju ekstremnog programiranja, praktičnom primenom programiranja u paru."***

Kodiranje softvera u ekstremnom programiranju

Programiranje u pâru:

Dva programera koji rade u pâru za jednim računarem mogu implementirati isto toliko (ili čak više) funkcionalnosti kao i dva programera koji rade svaki za sebe. Dakle, produktivnost pâra je ista ili veća nego produktivnost svakog programera pojedinačno. Oba programera u pâru povećavaju svoju kompetenciju čime se omogućuje proces deljenja znanja. Oba programera su uključena u trenutnu problematiku, odlučivanje i jedinično testiranje.
Razlikujemo dve uloge, odnosno role koje se medjusobno izmenjuju:

1. Programer koji koristi tastaturu računara i piše kôd (engl. *Driver*).
2. Programer koji proverava napisani kôd (engl. *Navigator*).



Kodiranje softvera u ekstremnom programiranju

Programiranje u pâru:

Prednosti:

• *Povećana disciplina.* Prave se manje pauze u radu, jer se partneri smjenjuju za računarom.

• *Bolji kod.* Partneri zajednički retko greše u dizajnu i sintaksi i teže kvalitetu

• *Fleksibilni tokovi posla.* Mnogo se fleksibilnije prihvataju ometanja rada jer se jedan programer bavi rešenjem ometanja dok drugi programira. Informacije protiču i među partnerima i prema ostatku tima.

• *Više programera doprinosi razvoju rešenja.* Ako je česta rotacija parova, više programera će biti uključeno u razvoj neke osobine i tako dati svoj doprinos.

• *Poboljšan moral*. Može biti i interesantnije od individualnog programiranja

• *Kolektivno vlasništvo nad kodom*. Kada svi u timu praktikuju ovu praksu i parovi često rotiraju, svi stiču pojam o celom kodu, ne samo nekom delu.

• *Mentorstvo*. Ovo je najbezbolniji način da se pojedinačna znanja izmešaju.

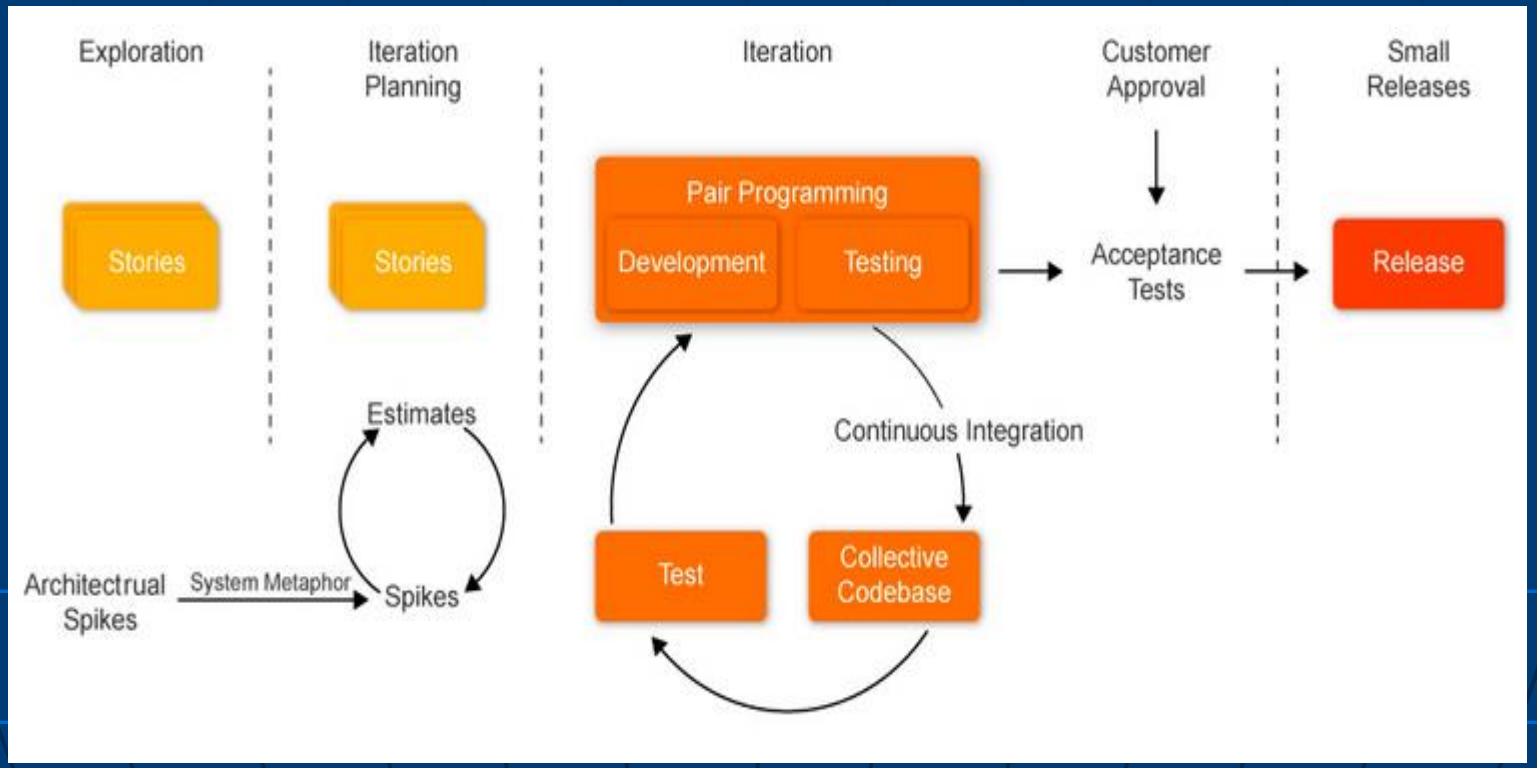
• *Manje ometanja*. Primećeno je da je ljudima lakše da ometaju pojedinca u radu, nego kada ih je dvojica.

• *Jedna radna stanica manje*. Pošto dvojica rade za jednim računarom, računar koji je preostao se može upotrijebiti u druge svrhe.

• *Manja osjetljivost na izostanke*. Praksom programiranja u paru se homoenizuju znanja o sistemu, te je izostanak nekog iz tima bezbolniji.

Kodiranje softvera u ekstremnom programiranju

Programiranje u pâru:



Kodiranje softvera u ekstremnom programiranju

Kontinuirana integracija kôda:

Bez kontrolisanja izvornog kôda, programeri koji vrše integraciju veruju da je sve u redu. Zbog paralelne integracije izvornog kôda modula, može postojati kombinacija kôda koja nije pre medjusobno testirana. Zbog toga često nastaje veliiki broj problema bez pravovremene detekcije. Slijedeći problemi se pojavljuju kada ne postoji jasan rez poslednje verzije koja se isporučuje naručiocu. To se ne odnosi samo na izvorni kôd, već i na kolekciju jediničnih testova koji moraju verifikovati ispravnost izvornog kôda. Ako se ne mogu osigurati kompletni, ispravni i konzistentni testovi, mogu se evidentirati lažne neispravnosti i propustiti prave neispravnosti u kôdu.

Neki projekti pokušavaju imati programere koji su zaduženi za odredjene dijelove kôda, npr. odredjene klase. Vlasnici klasâ se tada uveravaju da je kôd svake klase integrisan i ispravno isporučen. To umanjuje probleme kod integracije, ali zavisnost medju klasama uvek može biti problem.

Takav pristup zaduženja programera po klasama takodje ne rešava problem. Drugi način rešavanja problema integracije je uspostavljanje integratora ili tima integratora u projektu. Integrisanje kôda više programera često ne može obaviti samo jedna osoba. Tim ljudi je preveliki resurs za ntegrisanje više od jednom u nedelji. U takvom okružju, programeri koriste starije verzije kôda koji je prošao integraciju.

Kodiranje softvera u ekstremnom programiranju

Zajedničko vlasništvo nad kôdom:

Zajedničko vlasništvo nad kôdom (engl. *Collective Code Ownership*) ohrabruje sve koji sudeluju na projektu da doprinose novim idejama u svim segmentima projekta. Bilo koji programer može promeniti bilo koju liniju kôda kako bi dodao novu funkcionalnost, ispravio evidentirane neispravnosti ili refaktorisao.

Zajedničko vlasništvo nad kôdom znači da su i svi odgovorni za taj kôd. Celokupan tim je odgovoran za arhitekturu sistemaa i ˇcesto se ne izdvaja posebna projektna uloga arhitekte.

Arhitektura sistema je distribuirana u XP timu; svako iz tima ima odredjene odgovornosti nad arhitekturalnim odlukama. Način na koji funkcioniše zajedničko vlasništvo nad kôdom nalaže da svaki programer kreira jedinične testove uz kôd koji je napisao. Testovi su velika pomoć drugim programerima koji će se susresti sa tim kôdom jer:

- ✓ pokazuju kako se odredjeni deo kôda koristi
- ✓ kad se kôd menja (ispravljaju evidentirane neispravnosti ili se vrši refaktorisanje), izvršavanje testova ukazuje na promenu funkcionalnosti.

Kodiranje softvera u ekstremnom programiranju

Optimizacija kôda na kraju:

Optimizaciju kôda treba ostaviti za kraj razvojnog projekta, nikako nije dobro raditi optimizaciju za vreme trajanja implementacije.

Proces optimizacije je različit od procesa refaktoriranja. Refaktorisanjem se poboljšava kôd bez promene funkcionalnosti, čime se eliminišu viškovi i redundancija. Uglavnom, kôd se želi učiniti lakšim za razumevanje i bolje strukturiranim.

Optimizacija kôda se vrši kako bi se odredjeni delovi kôda zamenili sa delovima koji su tehnički napredniji: manjeg su nivoa složenosti, jednostavniji su za korišćenje i vremenski se brže izvode.

Ponekad je potrebno zameniti korišćene algoritme sa bržim algoritmima ili primeniti neki od poznatih obrazaca dizajna (engl. *design patterns*).

Najvažnije je da sistem ispravno radi. Tek nakon toga se može posvetiti optimizaciji kôda i brzini rada sistema.

Testiranje softvera u ekstremnom programiranju

Testovi se mogu podeliti u dva osnovna skupa:

1. Jedinični testovi
Proveravaju ispravnost rada sistema i uvek moraju raditi. Daju sigurnost programerima u vlastiti kôd i ujedno objašnjavaju kako se kôd koristi. Moraju biti automatizovani kako bi se omogućila jednostavna ponovljivost.

2. Testovi prihvaćenosti
Proveravaju funkcionalnost čitavog sistema koji je opisan korisničkim pričama. Odredjuju da li sistem zadovoljava kriterijum prihvaćenosti i omogućuju naručiocu da odluči da li može prihvatiti sistem. Pišu ga naručitelj uz pomoć programera. Testiranje je jedna od najvećih i najjačih značajki XP metodologije razvoja softvera. Svaki put kada programer promeni kôd iz bilo kog razloga, potrebno je ponoviti jedinične testove kako bi se uverilo da promene nisu narušile prethodno mplementiranu i testiranu funkcionalnost.

Iz tog razloga je važno da testovi budu automatizovani kako bi se omogućila njihova ponovljivost. Programer je ujedno i jedinični tester svog kôda.

Testiranje softvera u ekstremnom programiranju

Testiranje jedinice programa:

Jedinični testovi su jedna od najvažnijih stvari u ekstremnom programiranju. Za uspešno korišćenje jedinićnih testova, potrebno je kreirati ili koristiti neki od gotovih test okruženja koja su dostupna na tržištu i kojima je moguće vršiti jedinićno testiranje.

Kent Beck je napisao *SUnit* testno okružje za *Smalltalk* programski jezik. Posle toga su *Kent Beck* i *Erich Gamma* napravili *JUnit* za J2SE i J2EE platformu. Pomoću ovih alata, moguće je automatizovano izvršavati jedinične testove iz IDE razvojnog okružja.

Potrebno je testirati većinu klasa i metoda u objektnom sistemu. Može se za svaku klasu koja se želi testirati generisati testna klasa i za svaku metodu testna metoda. Nije nužno potrebno testirati sve metode. Obično je nepotrebno testiranje metoda kojima se postavljaju ili dohvataju vrednosti varijable ili objekta.

Poželjno je kreirati test pre pisanja kôda ako je to moguće. Jedinićni testovi se stavljaju u zajednički repozitorij zajedno sa datotekama izvornog kôda. Kôd bez pripadnog testa ne bi se uopšte trebao stavljati u zajednički repozitorij. Ako je otkriveno da nedostaje jedinični test (npr. za neku metodu), test treba biti naknadno kreiran. Tokom života projekta, automatizovani testovi mogu smanjiti troškove stotinu puta jer su najbolje oružje protiv evidentiranih neispravnosti. Čim je test teži za implementaciju, biće potrebniji jer će doneti i veće uštede.

Testiranje softvera u ekstremnom programiranju

Pronalaženje neispravnosti:

Kad se pronadje neispravnost, kreiraju se testovi kako bi se spriječilo da se ta neispravnost neprimećeno opet pojavi. Evidentirana neispravnost zahteva da bude napisan test prihvaćenosti softvera kojeg potvrdjuje naručitelj. Kreiranje testova prihvaćenosti pre suočavanja programerâ sa kôdom pomaže naručitelju koncizno definisanje problema i komuniciranje problema razvojnom timu.

Programeri u timu imaju test koji nije prošao i mogu fokusirati nastojanja da čim pre otkriju i reše problem. Kada je dobijen test prihvaćenosti koji nije prošao, programeri mogu kreirati jedinični test da pokažu neispravnost iz ugla gledanja koji je specifičan izvornom kôdu. Jedinični testovi koji nisu prošli daju odmah povratnu informaciju programerima da je neispravnost popravljena (pre su prolazili testovi koji nisu otkrivali neispravnost).

Kada svi jedinini testovi opet prolaze, može se opet pokrenuti test prihvaćenosti (koji nije prolazio) koji pokazuje da je neispravnost popravljena.

Neispravnost u sistemu takodje zahteva kreiranje specifičnog testa prihvaćenosti. Kada se dogodi greška u isporuci, tj. otkrije se neispravnost, programeri popravljaju grešku i isporučuju zadnju i ispravljenu verziju sistema. Testom prihvaćenosti se može proveriti da li je greška ispravljena.

Testiranje softvera u ekstremnom programiranju

Faza isporuke:

Faza isporuke (engl. *Productionizing Phase*) zahteva posebno testiranje i proveru performansi sistema pre nego što sistem može biti isporučen kupcu. U toj fazi se mogu dodavati nove promene uz odluku da li će biti uključene u trenutnu isporuku ili u neku od sledećih isporuka. Za vreme faze isporuke, iteracije trebaju biti skraćene od tri nedelje na jednu nedelju uz obavezno održavanje dnevnih stojećih sastanaka. Odložene, tj. odgodjene predložene ideje su dokumentirane za kasniju. Takodjr, u fazi isporuke se može pozabaviti sa pitanjima performansi sistema ali tek nakon što sistem provereno radi.

Faza održavanja:

Nakon prve isporuke, XP tim mora održavati sistem u radu zajedno sa implementacijom novih iteracija (nove funkcionalnosti). Kako bi se to postiglo, faza održavanja (engl. *Maintenance Phase*) zahteva napor za pružanje podrške kupcu. Razvoj sistema koji je isporučen u radu nije isti kao i razvoj sustava koji još nije došao u fazu isporuke. Treba biti oprezniji sa promenama koje se rade pa i biti spreman prekinuti trenutni dalji razvoj kako bi se rešili problemi u radu jer su oni najvišeg prioriteta.

Razvojna brzina se može deklarisati nakon što je sistem isporučen i u radu. Faza održavanja može zahtevati inkorporiranje novih ljudi u projektni tim i odredjene promene strukture tima.

Testiranje softvera u ekstremnom programiranju

Faza isporuke:

Faza isporuke (engl. *Productionizing Phase*) zahteva posebno testiranje i proveru performansi sistema pre nego što sistem može biti isporučen kupcu. U toj fazi se mogu dodavati nove promene uz odluku da li će biti uključene u trenutnu isporuku ili u neku od sledećih isporuka. Za vreme faze isporuke, iteracije trebaju biti skraćene od tri nedelje na jednu nedelju uz obavezno održavanje dnevnih stojećih sastanaka. Odložene, tj. odgodjene predložene ideje su dokumentirane za kasniju. Takodjr, u fazi isporuke se može pozabaviti sa pitanjima performansi sistema ali tek nakon što sistem provereno radi.

Faza održavanja:

Nakon prve isporuke, XP tim mora održavati sistem u radu zajedno sa implementacijom novih iteracija (nove funkcionalnosti). Kako bi se to postiglo, faza održavanja (engl. *Maintenance Phase*) zahteva napor za pružanje podrške kupcu. Razvoj sistema koji je isporučen u radu nije isti kao i razvoj sustava koji još nije došao u fazu isporuke. Treba biti oprezniji sa promenama koje se rade pa i biti spreman prekinuti trenutni dalji razvoj kako bi se rešili problemi u radu jer su oni najvišeg prioriteta.

Razvojna brzina se može deklarisati nakon što je sistem isporučen i u radu. Faza održavanja može zahtevati inkorporiranje novih ljudi u projektni tim i odredjene promene strukture tima.

XP projektne uloge

Postoje različite projektne uloge u XP-u koje su se profilirale za različite zadatke i svrhe tokom procesa programiranja i razvoja softvera. Mogu se izdvojiti sledeće projektne uloge:

- ✓ programer (engl. *Programmer*) (odjeljak 5.6.1)
- ✓ naručilac (engl. *Customer*) (odjeljak 5.6.2)
- ✓ tester (engl. *Tester*) (odjeljak 5.6.3)
- ✓ tragač (engl. *Tracker*) (odjeljak 5.6.4)
- ✓ trener (engl. *Coach*) (odjeljak 5.6.5)
- ✓ savetnik (engl. *Consultant*) (odjeljak 5.6.6)
- ✓ menedžer (engl. *Manager*). (odjeljak 5.6.7)

Uloge menadžera, tragača i trenera može fizički obavljati jedna osoba. Te tri uloge su menedžerske uloge.

XP projektne uloge

Programer:

Programer je srce XP-a. Kada bi programer uvek mogao donositi odluke koje pažljivo balansiraju izmedju kratkoročnih i dugoročnih prioriteta, ne bi bilo potrebe za nekom drugom tehničkom osobom osim programera. Programeri su, dakle, uz naručioce, druga polovina osnovnog tima u XP-u. XP programer je zapravo isto kao i programer bilo koje druge discipline razvoja softvera. Glavni zadatak programera je rad sa programima, odnosno kôdom; čineći ih većim, jednostavnijim i bržim. Zadatke kodiranja XP programera se mogu rezimirati ovako:

• testiranje - kodiranje – refaktorisanje što je u suprotnosti s tradicionalnim pristupom:
• dizajn - kodiranje - testiranje.

Zadatak programera, osim razvoja kôda, je i komunikacija sa ostalim osobama koje sudeluju u projektu, dakle projektnog tima. Programer je autor testova za svoj kôd kojima pokazuje da njegov kôd radi i kako radi. Postoje veštine koje XP programer mora posedovati, a najvažnija medju njima je programiranje u pâru. Sledćca veština je osećaj za jednostavnost. Važno je održavati kôd čim više jednostavnim kako bi promene na kôdu bile lakše (npr. dodavanje nove funkcionalnosti, ispravljanje evidentiranih neispravnosti, refaktorisanje i slično) te kako bi bila veća razumljivost tog kôda.
Ostale veštine koje treba pojedovati XP programer su tehnički orijentisane i tiču se znanja dobrog programiranja. Uključuju znanje solidnog programiranja, refaktorisanja i pisanja jediničnih testova.

XP projektne uloge

Naručilac:
Naručilac je, uz programera, druga polovina osnovnog tima u XP-u. Dok programer zna *kako* programirati, naručilac zna *šta* programirati. Osnovna veština koju treba posedovati naručilac je da piše dobre, programerima razumljive korisničke priče. Potreban je stav koji podstiče opštu uspešnost projekta.
Naručilac odredjuje prioritet priča, odnosno zahteva unutar iteracije, znajući koja je važnost pojedine priče za kupca.
Pisanje testova prihvć´enosti je takodjr jedna od zadataka naručioca.
Na osnovu rezultata testova, naručioc odredjuje koja funkcionalnost sistemaa je zadovoljena.

Tester:
Budući da je mnogo testnih odgovornosti na ledjima programera, uloga testera u XP timu je fokusiranost na naručioca. Prvenstveno, pomaže naručiocu u pisanju funkcijskih testova. Ako funkcijski testovi nisu deo integracijske garniture, odgovornost testera je da redovno pokreće funkcijske testove i rezultate objavljuje na vidljivom mestu.
XP tester nije odeljena osoba posvećena ’probijanju’ sistema. Zadaci testera su redovno pokretanje testova, obradjivanje rezultata testiranja i provjera da programska pomagala za testiranje rade u redu.

XP projektne uloge

Tragač:
Tragač je osoba koja vodi računa o dogovorenim i završenim zadacima u pojedinoj iteraciji. On meri količinu obavljenog posla XP tima. XP propisuje praćenje nekoliko metrika. Najvažnija metrika je brzina projekta. Važan podatak je promena brzine projekta, količina prekovremenog rada i odnos rezultata testova (postotak evidentiranih neispravnosti u odnosu na ispravnu funkcionalnost). Svi ti podaci mere napredak i pomažu odrediti da li se projekt odvija prema dogovorenom rasporedu u trenutnoj iteraciji. Tragač može upozoriti ako je ugrožen dogovoreni raspored. Kako bi odredio brzinu projekta unutar iteracije, tragač obično svaki dan ili svaka dva dana prati završene zadatke unutar projektnog tima. Kada XP tim bude odredjivao trajanje iteracije na osnovu procenjenog trajanja korisničkih priča, moći će se koristit podatak o brzini projekta iz prethodnih iteracija za donošenje realnijih procena. Redovno praćenje napretka može pomoći XP timu da bolje uoči i popravi svoje nedostatke, znajući u svakom trenutku da li je na dobrom putu da izvrši dogovoreno.

Trener:
Trener je odgovorna osoba za celokupan proces. Njegov zadatak je pratiti da li se članovi tima pridržavaju primenjene razvojne metodologije. Svako u XP timu je odgovoran razumeti vlastitu ulogu u XP-u. Trener je treba ’dublje’ razumeti, znati koje alternativne prakse mogu pomoći rešiti odred¯ene probleme, kako drugi timovi primenjuju XP, koje su ideje iza XP-a i kako su povezane sa trenutnom situacijom. Trener u XP-u vodi tim i njegov je mentor. Njegova pozicija je da vodi XP tim svojim primerom. Glavna vrlina mu je posedovanje golemog iskustva i tehničkog znanja. Trener vodi tim kako bi tim razumeo XP i razvoj softvera. Veštine koje razvija XP su jednostavan dizajn, refaktorisanje i testiranje. Ako tim ima i tehnička znanja i znanja procesa, trener mora konstantno podsećati tim na procedure kojih se trebaju pridržavati u odredjenim situacijama. Posao se trenera smanjuje sa starošću tima.

XP projektne uloge

Savetnik:

XP tim sa vremena na vreme treba tehničkog savetnika u vidu konzultanta, uprkos fleksibilnosti i znanju koje poseduje. Uloga konsultanta je pomoći timu rešiti odredjeni tehnički problem ili mu razjasniti nejasnoće iz domena kojom se XP tim bavi.
Tim treba kreirati posebne testove kako bi ukazao na problem i kako bi bilo vidljivo da je problem rešen. Ideja je poučiti tim kako da ubuduće rešava tehničke probleme.

Menadžer:

Menadžer je osoba izvan tima i predstavlja tim prema spoljašnjem svetu. On oformljuje, tj. slaže tim i nabavlja potrebne resurse. Takodje, upravlja timom (organizuje sastanke, beleži napredak i slično). Vlasnik je tima, ali i njihovih problema.

Sled dnevnih aktivnosti u XP-u

Jedan tipičan dan XP programiranja može se ovako sumirati:

1. Dnevni ’stojeći’ sastanak na početku dana. Identifikuju se problemi pa se odredjuje ko će ih rešiti (problemi se ne rešavaju na sastanku). Donosi se izveštaj (pregled) aktivnosti od prethodnog dana.

2. Formiranje parova programera. Sâv kôd koji će biti uključen u isporuku je kreiran od dva programera koji rade u pâru za jednim računarem. Oba programera iz pâra su uključena u problematiku, odlučivanje i jedinčcno testiranje.

3. Testiranje: Implementacija testova prethodi kodiranju. Važno je pokriti testovima veći deo kôda (sve što može krenuti loše), po mogućnosti kompletan kôd. Za sve nepoznanice i probleme, potrebno se je obratiti naručiocu.

4. Kodiranje: Nakon testiranja sledi kodiranje. Važno je implementirati najjednostavnije stvari koje bi mogle funkcionisati.

Sled dnevnih aktivnosti u XP-u

Jedan tipičan dan XP programiranja može se ovako sumirati:

5. Refaktorisanje: Refaktoriranje je tehnika poboljšavanja kôda bez promene funkcionalnosti.

6. Pitanja i odgovori: Za sva moguća pitanja i nejasnoće, na raspolaganju je naručilac.

7. Integriši ili odbaci (engl. Integrate or Toss): Programeri trebaju integrisati i isporučivati kôd u zajednički kontrolni repozitorijum izvornog kôda i prilikom toga ponoviti testove. Ako neki testovi ne prolaze, potrebno je otkriti i popraviti moguće probleme tako da svi testovi na kraju prodju. Ako nešto nije dobro, treba to odbaciti i pokušati napraviti iz početka (isti dan ako još ima vremena ili sledeći dan).

8. Kraj posla u 17h: Potrebno je držati se 40-satne radne nedelje, bez prekovremenog rada. Sve što se radilo tog dana je integrisano ili odbačeno.

HVALA NA PAŽNJI