UNIVERZITET U NOVOM SADU
PRIRODNO-MATEMATIČKI FAKULTET
DEPARTMAN ZA MATEMATIKU
I INFORMATIKU

Robert Pap

Ekstremno programiranje kao metod agilnog razvoja softvera

– diplomski rad –

Novi Sad, 2008.

Sadržaj

Predgovor

Softver je pojam koji se može opisati kao kolekcija programa, podataka i dokumentacije koji za cilj ima da izvrši neke zadatke na računaru. Jedna od važnih osobina softvera je i to da je relativno složen. Već je i *Booch* istakao da su ozbiljni softverski proizvodi toliko komplikovani da je skoro nemoguće da jedna osoba shvati sve karakteristike nekog softvera. Ovaj zaključak još više dobija na snazi ako se uzme u obzir činjenica da prosečna veličina softvera svake godine samo raste. Po *Denert*-u softverska rešenja traže i visoku preciznost, prema tome softver je sklon greškama. Iz svega ovoga se može zaključiti da je razvoj softvera jedan veoma važan i ozbiljan proces i da taj razvoj u većini slučajeva zahteva više ljudi.

Ljudi su već odavno uvideli da proces razvoja softvera mora biti nadgledan. 60-ih godina prošlog veka pojavili su se prvi projekti u kojima je učestvovalo preko 1000 ljudi. Ove 60-te godine su ostale upamćene i po još nečemu: 1965. godina se smatra početkom „**softverske krize**“. Ova kriza je „uspešno“ identifikovala aktuelne nedostatke i probleme u razvoju softvera. Veliki broj projekata je premašio svoj budžet ili su bili prekasno završeni i isporučeni. Ovo je kompanije sveukupno koštalo milijarde dolara. Postojali su i drugi problemi kao što je na primer povreda vlasništva. Zbog niske sigurnosti softverskih proizvoda (i zbog raznih softverskih defekata) dešavale su se prve krađe identiteta. Na kraju, ne sme se izostaviti ni to da su zbog loše napisanih softvera neki ljudi izgubili čak i svoje živote (na primer *Therac-25*, računarski kontrolisan uređaj za terapijsku radijaciju, fatalno je ozračio neke svoje pacijente). Iako se 1985. smatra za kraj softverske krize, neki smatraju da ona traje još i danas.

Naučna Komisija NATO-a (eng. *NATO Science Committee*) je 1968. i 1969. godine sponzorisala dve naučne konferencije na kojima je po prvi put spominjan pojam „softverskog inženjerstva“, kao neka vrsta rešenja aktuelnih problema. Mnogi smatraju ove dve konferencije kao zvanični početak softverskog inženjerstva i profesije softverskog inženjerstva.

Pojam **softverskog inženjerstva** se može definisati na više načina. Po *Pagel*-u i *Six*-u, „Softversko inženjerstvo ... bavi se ekonomskim razvojem softvera visokog kvaliteta“. Po *Sommerville*-u, „Softversko inženjerstvo je inženjerska disciplina koja se bavi praktičnim problemima razvoja velikih softverskih sistema“. Po *IEEE* standardu, „Softversko inženjerstvo je primena sistematskih, disciplinovanih, merljivih pristupa razvoju, rukovanju i održavanju softvera; drugim rečima, primeni inženjerstva na softver“. Zajednička karakteristika ovih definicija je „razvoj softvera“, drugim rečima, jedan od glavnih ciljeva softverskog inženjerstva je upravo proučavanje modela za sistematski razvoj softvera. Ovi modeli se zovu **modeli procesa**, i važni su za organizaciju projekta (projektom se ne sme haotično upravljati), za planiranje projekta (planiranje vremena i troškova), kao i za utvrđivanje teškoća i kritičnih delova tokom razvoja. Uopšteno rečeno, model procesa je razvojni plan koji definiše opšti proces razvoja softverskog proizvoda.

Jedan od najranijih modela procesa je bio klasični vodopadni model razvijen 1970. godine koji je – kako se ispostavilo – imao velike nedostatke, ali je ipak bio bolje rešenje od haotičnog koordinisanja softverskim projektima. Vremenom su nastale razne varijante ovog osnovnog modela koje su već uvažile njegov najveći nedostatak: nedostatak iteracije. Ovako je nastao iterativni i evolutivni razvoj, a iz jedne radikalne ideologije koja se dosta razlikovala od dosadašnje prakse, nastali su agilni metodi. Jedan od najpoznatijih agilnih metoda je zapravo ekstremno programiranje.

Cilj ovog rada je da prikaže osnove ekstremnog programiranja kao agilnog metoda razvoja softvera. Rad se sastoji iz dva glavna dela. U prvom delu će biti predstavljen iterativni, evolutivni i agilni razvoj. Prvo će biti predstavljen klasični vodopadni model, a zatim će se uvesti pojam iteracije. Ovo znanje će biti potrebno za objašnjenje iterativnog i evolutivnog razvoja, a kasnije i agilnog razvoja kao unapređenja iterativnog razvoja. Biće reči i o tome, zašto se smatra iterativni i agilni razvoj boljim rešenjem od klasičnog vodopadnog modela. Na kraju ovog dela će biti opisana četiri poznata iterativna i agilna metoda. Jedan od ovih metoda – ekstremno programiranje – će biti glavna tema drugog velikog dela ovog rada. Prikazaće se osnovna ideja iza ovog metoda, a zatim će se ući u malo dublju analizu principa koji predstavljaju sastavni deo ekstremnog programiranja i koji su neophodni za uspešnu primenu ovog metoda u praksi.

Ovom prilikom se zahvaljujem svom mentoru *dr Zoranu Budimcu* na korisnim sugestijama i primedbama. Takođe bih želeo da se zahvalim rodbini na podršci tokom mog studiranja.

Novi Sad, 2008.

Robert Pap

1. Iterativni, evolutivni i agilni razvoj

Kao što je već rečeno, modeli procesa služe za proučavanje sistematskog razvoja softvera. Uopšteno rečeno, to je razvojni plan koji definiše opšti proces razvoja softverskog proizvoda. Model procesa se može i preciznije definisati, prema čemu on određuje koje aktivnosti se izvršavaju, od strane kojih osoba u kojim ulogama, kojim redosledom će aktivnosti biti izvršavane, koji proizvodi će biti razvijani i kako će se procenjivati njihov kvalitet. **Aktivnost** je u suštini podproces modela procesa. Pod pojmom **uloga** podrazumevamo saradnika koji izvršava određenu aktivnost. U razvoju softverskih proizvoda uobičajene uloge su: programer, vođa projekta, projektant, itd. Međutim, mora se napomenuti da različiti metodi definišu i različite uloge, prema tome jedan od važnih kriterijuma za odabir korišćenog metoda je procenjivanje da li će vođa projekta naći osobe koje bi odigrale uloge definisane u posmatranom metodu.

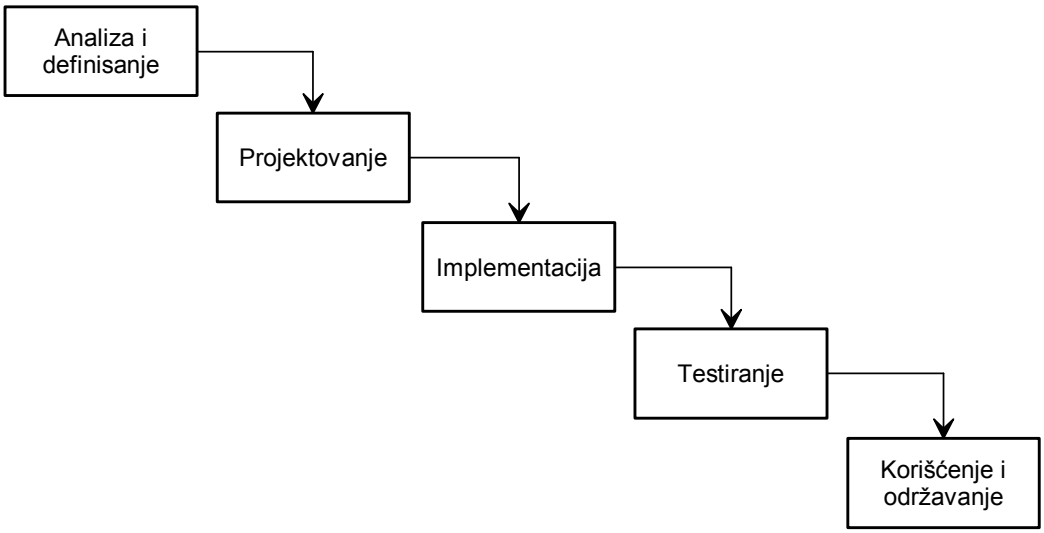
Ovaj deo rada se bavi definisanjem iterativnog i evolutivnog razvoja kao i definisanjem agilnog razvoja. Prvo je potrebno uvesti pojam iteracije i evolucije. Posle toga će se objasniti suština iterativnog, evolutivnog i agilnog razvoja zajedno sa odgovorom na pitanje: „Zašto su ovakvi načini razvoja bolji od starog vodopadnog modela?“. Na kraju ovog dela će biti dat opis četiri poznata predstavnika iterativnog i agilnog razvoja. Pre svega biće malo reči i o klasičnom vodopadnom modelu koji predstavlja osnovu za uvođenje drugih modela procesa.

1.1. Klasični fazni model (klasični vodopadni model)

U početku, razvoj softvera je bio jedan nekoordinisan i haotičan proces. Klijenti softvera su iznosili svoje zahteve šta sve žele da vide u programu. Možda je bolje ove zahteve zvati samo idejama, pošto na početku razvoja niko ne može precizno reći kako je zamislio finalni proizvod. S druge strane, ovi zahtevi nisu bili precizno dokumentovani, i zato se zovu *neformalni zahtevi*. Posle toga, tim za razvoj je razvio softver i isporučio ga. Šta se sve događalo unutar procesa razvoja je teško definisati, jer ni uloge nisu bile tako razdvojene (znači svi su radili ponešto), ni plan razvoja nije bio precizno definisan. Prema tome nije ni čudo da su neki ovaj rani način razvoja softvera nazvali *crnom kutijom*. Finalni proizvodi (ako je tim uopšte stigao do finalnog proizvoda) su bili blago rečeno – problematični: razvoj je bio skup i spor, finalni proizvod nije odslikavao stvarne želje tj. zahteve klijenata, a verovatno je bio i prepun grešaka.

Kao odgovor na ove probleme 70-ih godina prošlog veka stigao je klasični fazni model ili kako se popularnije naziva **klasični vodopadni model** (eng. *Classic Waterfall Model*). Glavni cilj ovog modela procesa je bio da uvede red u razvoj softvera: da bude dobro isplaniran i dokumentovan sa jasno definisanim ciljevima i ulogama. Vodopadni model se sastoji od četiri različite faze i od jedne dodatne pete faze (slika 1). Te faze su redom:

1. **Analiza i definisanje** (eng. *Analysis and Definition*)
2. **Projektovanje (ili dizajn)** (eng. *Design*)
3. **Implementacija** (eng. *Implementation*)
4. **Testiranje** (eng. *Test*)
5. **Korišćenje i održavanje** (eng. *Usage and Maintenance*)



Slika 1: Klasični vodopadni model

Glavni ciljevi **faze analize i definisanja** su analiza problema koji se rešava i definisanje zahteva softverskog proizvoda, drugim rečima opis spoljnog ponašanja softverskog proizvoda. Ova faza se deli na dve podfaze:

- Faza planiranja
- Faza definisanja

U fazi planiranja se pravi studija izvodljivosti, dok se u fazi definisanja pravi definicija proizvoda. **Studija izvodljivosti** (eng. *Feasibility Study*) je jako važna jer predstavlja osnovu ugovora sa klijentima. Ovde se procenjuju i troškovi posmatranog projekta, pa vođa projekta može odrediti veličinu tima potrebnu za razvoj softvera, potrebno vreme za razvoj, kao i budžet. Ako se proceni da je budžet nedovoljan za razvoj posmatranog softverskog proizvoda, projekat se može ukinuti. Ovo je najzgodniji trenutak za ukidanje projekta jer bilo koji kasniji trenutak već predstavlja trošak koji se ne može nadoknaditi. Za razliku od studije izvodljivosti, **definicija proizvoda** (eng. *Product Definition*) predstavlja osnovu projektovanja koja sadrži sve važne dokumente koji su važni u sledećoj velikoj fazi, a to je faza projektovanja. Dva najvažnija dokumenta su:

- **Specifikacija zahteva** (eng. *Software Requirements Specification*, *SRS*) – kompletan opis ponašanja sistema koji se razvija
- **Model proizvoda** (eng. *Product Model*) – gradi formalizovan model proizvoda (kao deo definicije proizvoda), a uz to pretvara neprecizan, verbalan opis u specifikaciji zahteva u precizniji oblik. Postoji u dve varijante: **strukturna analiza** (eng. *Structured Analysis*) i **objektno-orijentisana analiza** (eng. *Object-Oriented Analysis*).

Druga faza u vodopadnom modelu je **faza projektovanja (dizajna)**. Cilj ove faze je specifikacija strukture softvera i specifikacija komponenti i njihovih veza, prema tome ova faza ima dva dokumenta: **arhitekturu softvera** (eng. *Software Architecture*) koja opisuje komponente i njihove međusobne veze, i **specifikaciju komponenti** (eng. *Specification of Components*) koja pojedinačno opisuje komponente kao crne kutije. U praksi, cilj ove faze je da proširuje, modifikuje i optimizira model proizvoda (tj.

strukturnu analizu ili objektno-orijentisanu analizu u zavisnosti koja je korišćena). Ovako nastaje **strukturni dizajn** i **objektno-orijentisani dizajn**.

Treća faza je **faza implementacije**. Ova faza dopunjuje arhitekturu softvera i započinje programiranje komponenti u odabranom jeziku implementacije. Rezultati ove faze su: izvorni program, objektni program, dokumentacija, a često i test-dokumentacija. Posebna pažnja se posvećuje na stil i metodologiju programiranja.

Četvrta faza je **faza testiranja**. U ovoj fazi se vrši testiranje komponenti i njihove integracije. Iako može da zvuči čudno, testiranje oduzima najviše vremena u celom razvoju softverskog proizvoda i uglavnom iznosi 50% ili više. Kratko rečeno, cilj ove faze je da se isporuči softverski proizvod bez grešaka. Testiranje je jako važan aspekt u razvoju softvera jer prekasno primećene greške (greške koje se nalaze u isporučenom proizvodu) mogu kompanije da koštaju i milione dolara. Postoje više vrsta testova:

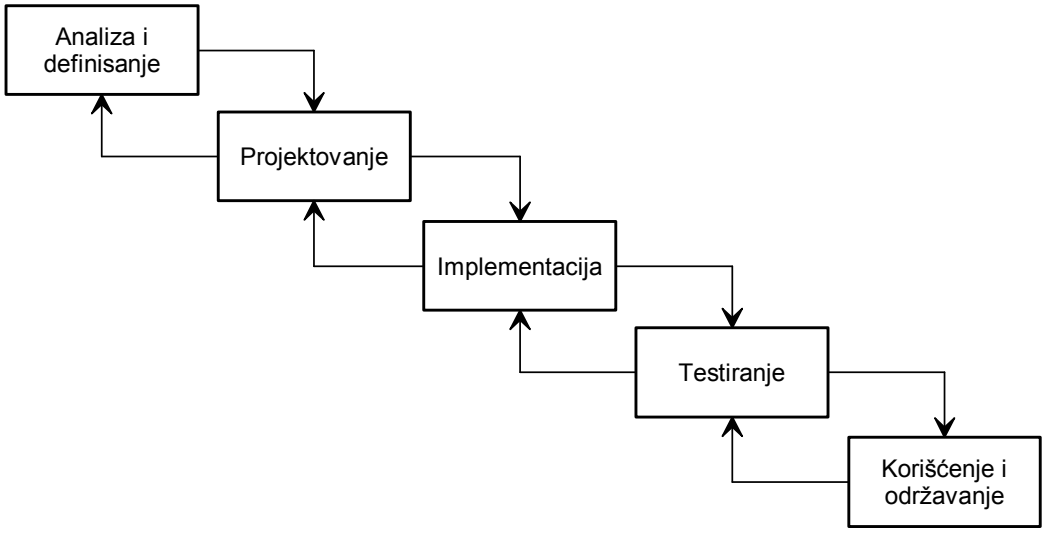
- **Test modula** (eng. *Unit Test*)
- **Integracioni test** (eng. *Integration Test*)
- **Sistemski test** (eng. *System Test*)
- **Test prihvatanja** ili **prijemni test** (eng. *Acceptance Test*) – ovo je sistemski test ali sa prisustvom klijenta
- **Terenski test** (eng. *Field Test*, *Installation Test*) – testiranje u ciljnim okruženjima, podrazumeva i instaliranje proizvoda
- **Regresioni test** (eng. *Regression Test*) – ponovljeno testiranje svaki put kad se izmeni programski kod

Posle četvrte faze softverski proizvod je spreman za isporuku. U nekim slučajevima ovde se i završava proces razvoja softvera i neke varijacije vodopadnog modela nemaju petu fazu (**faza korišćenja i održavanja**), neke imaju, dok su neke varijacije prosto spojile ovu fazu sa fazom testiranja. Razlog za ovu dodatnu fazu leži u činjenici da posle isporuke softvera taj proizvod treba i održavati. Ovo podrazumeva tehničku podršku sa ciljnom publikom (tj. korisnicima koji koriste softver), uvažavanje primedbi, dodavanje novih zahteva ili modifikacija postojećih, i ispravljanje do sada skrivenih grešaka. Zanimljivo je da fazu održavanja ne mora da radi isti tim koji je bio zadužen za razvoj samog softvera.

Klasični vodopadni model je bio veoma popularan, i postao je svetski standard. Međutim, ovaj model je imao i neke ozbiljne probleme. Najveći problem je bio to da nije omogućio vraćanje na prethodne faze. Ako je razvojni tim napravio grešku u fazi analize i prešao na fazu dizajna, nije mogao da se lako vrati na prethodnu fazu. S druge strane, nije omogućavao da se kroz neku fazu prolazi više puta. Zbog ovih problema nastale su razne varijacije vodopadnog modela, a kasnije su se pojavili i neki potpuno drugačiji modeli koji su išli svojim putem. Prva varijacija polaznog vodopadnog modela je bio *iterativni fazni model*.

1.2. Iterativni i evolutivni razvoj

Iterativni fazni model je bio prva modifikacija klasičnog vodopadnog modela. Ovaj model je održao faze definisane u polaznom modelu, ali je dodao jednu novinu. Naime, ovaj model je omogućavao vraćanje na prethodnu fazu, time rešavajući prvi problem vodopadnog modela[1] (slika 2). Međutim, drugi problem je bio i dalje nerešen. Bez obzira na ovo, iterativni fazni model je uneo svežinu u proces razvoja i odredio smer razvijanja kasnijih modela. Suština cele priče je bio pojam *iteracije*.



Slika 2: Iterativni fazni model

Iteracija (eng. *Iteration*) se može posmatrati kao mini-projekat sa posebnim fazama analize, projektovanja, implementacije i testiranja. Način razvijanja koji koristi ovakve iteracije se naziva **iterativni razvoj** (eng. *Iterative Development*). Znači, to je način razvoja softvera čiji se ukupni životni vek sastoji od nekoliko sukcesivnih iteracija. Cilj svake iteracije je da do kraja posmatrane iteracije razvije jedno **iterativno izdanje** (eng. *Iterative Release*). Iterativno izdanje je jedan stabilan, integrisan, iztestiran, *delimično* završen sistem. Većina ovih iterativnih izdanja je samo *unutrašnje izdanje* i nije primenjeno za *spoljašnju* isporuku (na tržište), već ima značaj samo za razvojni tim. Iterativno izdanje poslednje iteracije je finalni proizvod tj. proizvod koji je spreman za isporuku.

Iako u teoriji jedno iterativno izdanje može biti i pročišćavanje programskog koda od grešaka ili optimizacija koda radi poboljšanja performansi, u praksi je to uglavnom povećanje *funkcionalnosti*. Znači, svaka sledeća iteracija sadrži neku novu funkcionalnost i tako ovaj delimično završen sistem vremenom postaje sve veći i kompletniji u odnosu na prethodna iterativna izdanja. Ovaj način razvoja naziva se **inkrementalnim razvojem** (eng. *Incremental Development*). Koncept razvoja sistema kroz ovakve iteracije se naziva **iterativno-inkrementalnim razvojem** (eng. *Iterative*

[1] Mada nije omogućio skokove, npr. vraćanje dve faze unazad.

and Incremental Development, *IID*), ali u većini slučajeva izjednačava se sa iterativnim razvojem. *IID* je postao osnova agilnih metoda.

Postavlja se pitanje, šta raditi tokom jedne iteracije? Postoje dve vrste iterativnog planiranja i razvoja:

- **Razvoj usredsređen ka riziku** (eng. *Risk-Driven Iterative Planning*)
- **Razvoj usredsređen ka klijentima** (eng. *Client-Driven Iterative Planning*)

Razvoj usredsređen ka riziku u prvim iteracijama uglavnom uzima najteže elemente za razvoj, a tek kasnije dodaje ostale elemente. **Razvoj usredsređen ka klijentima** uzima za sledeću iteraciju one elemente koje je klijent definisao za tu iteraciju. *IID* preporučuje mešavinu ove dve vrste. Naime, razvoj usredsređen ka riziku može eliminisati velike troškove ukoliko se projekat mora ukinuti, a razvoj usredsređen ka klijentima prvo implementira one elemente koji imaju najveću (tržišnu i poslovnu) vrednost za klijenta. Mešavina ove dve vrste je zgodna i zbog toga što klijenti uglavnom ne mogu da prepoznaju elemente koji su komplikovani i teški za implementaciju, dok programeri ne mogu da prepoznaju elemente koji imaju najveću tržišnu i poslovnu vrednost.

Kod iterativnog razvoja takođe se postavlja pitanje, koliko bi trebalo da traje jedna iteracija. Ovo zavisi od korišćenog metoda razvoja, ali uglavnom traje od nedelju dana do šest nedelja (može da traje i duže, ali se ne preporučuje, sem kod velikih razvojnih timova). Kod dužine iteracije ne sme se izostaviti ni pojam ***timebox***-a. *Timebox* označava da je krajnji datum tj. završetak posmatrane iteracije u potpunosti *fiksiran*, i ne može se naknadno produžiti. Kad je jasno da na početku postavljeni zahtevi neće moći da se implementiraju do kraja *timebox*-ovane iteracije, umesto da se produži rok završetka iteracije, iz iteracije se izbacuju zahtevi sa manjim prioritetom (tj. smanjuje se opseg te iteracije). *IID* ne zahteva da svaka iteracija unutar projekta ima istu dužinu, ali zahteva (ili bar strogo preporučuje) *timebox*-ovanje svake iteracije.

Pored iterativnog razvoja u praksi se koriste i drugi termini kao što su evolutivni razvoj i adaptivni razvoj. Iako su ovi termini dosta slični, između njih postoje i razlike. **Evolutivni razvoj** (eng. *Evolutionary Development*) implicira da zahtevi, plan, procene i rešenja s vremenom evoluiraju ili se profinjavaju kroz iteracije. Znači, kod ove vrste razvoja osnovna ideja je da se funkcije proizvoda razvijaju postepeno kroz iteracije, a ne odjednom. **Adaptivni razvoj** (eng. *Adaptive Development*) je nešto slično. On implicira na to da se elementi prilagođavaju tako da zadovolje rezultate **povratne sprege** (eng. *Feedback*) od strane korisnika, programera, test-osoba, itd.

Planiranje kod evolutivnog i adaptivnog razvoja je malo specifično. Naime, na početku projekta se ne navode svi detalji vezani za softver koji se razvija. Razlog tome leži u činjenici da na početku procesa razvoja nije ni moguće sve odrediti i precizirati. Takođe, ovaj način planiranja se bolje rezonuje sa promenama i novim problemima. Prema tome, umesto da se troši nepotrebna energija za nešto što se ne može precizno predvideti, koriste se delimične definicije i procene. Ali to ne znači da će kroz celi projekat vladati nepoznanica i magla. Kako vreme prolazi, procene i vremenski rokovi će postati sve jasniji i precizniji. *McConnell* je ovo nazvao **konusom neizvesnosti** (eng. *Cone of Uncertainty*). Ovaj način planiranja se naziva **adaptivno planiranje** (eng. *Adaptive Planning*). Iz svega ovoga se može zaključiti da adaptivno planiranje gleda samo u skoru budućnost, dok uobičajeno planiranje (korišćeno u vodopadnom i u drugim modelima) gleda u daleku budućnost.

Kao što je već rečeno, većina iterativnih izdanja je samo *unutrašnja*, tj. nije namenjena isporuci na tržište, dok je poslednje iterativno izdanje već finalni proizvod.

Tokom razvoja ne mora samo poslednje iterativno izdanje da bude *spoljašnje*. Iteracije koje čine projekat mogu biti grupisane, i na kraju svakog skupa iteracija može se izdati jedno *spoljašnje* iterativno izdanje. Takvi skupovi iteracija su poznati pod nazivom **inkrementalna isporuka** (eng. *Incremental Delivery*). Jedna inkrementalna isporuka ima dužinu od tri do dvanaest meseci. Postoji i jedna druga vrsta isporuke. **Evolutivna isporuka** (eng. *Evolutionary Delivery*) je profinjavanje inkrementalne isporuke u kojoj postoji snažan pokušaj da se dobije povratna sprega od strane korisnika. Rezultati povratne sprege će se koristiti kao putokaz za planiranje sledeće isporuke. Evolutivne isporuke traju mnogo kraće: nedelju dana ili dve nedelje. Razlog tome je što brža isporuka proizvoda krajnjim korisnicima. Znači, osnovna razlika između ove dve vrste isporuke je u povratnoj sprezi: kod evolutivne isporuke ona određuje plan sledeće isporuke, a kod inkrementalne isporuke ne. U praksi se preporučuje mešavina ova dva načina isporuke.

Dva najpoznatija metoda *IID*-a su *Evo* i *UP* (i *RUP*, njegova najpoznatija varijanta). O njima će biti više reči nešto kasnije u poglavlju 1.5.

Sve u svemu, *IID* je postao jako popularan, pa nije ni čudo da su stručnjaci nastavili sa eksperimentisanjem, i tako je nastao *agilni razvoj*.

1.3. Agilni razvoj

Agilni razvoj (eng. *Agile Development*) predstavlja unapređenje i profinjavanje iterativno-inkrementalnog razvoja (*IID*). Agilni razvoj koristi *timebox*-ovan iterativni i evolutivni razvoj, adaptivno planiranje, i preporučuje evolutivnu isporuku. **Agilnost** (eng. *Agility*) predstavlja rapidan i fleksibilan odgovor na promene, njena suština je *manevaribilnost*, dok je prema *Kent Beck*-u njen moto „*zagrli promene*“ (eng. „*Embrace Change*“).

2001. godine jedna grupa zainteresovana za iterativne i agilne metode sastala se sa ciljem da nađe zajedničku reč. Iz ovoga je nastala **Agilna Alijansa** (eng. *Agile Alliance*), jedna od vodećih grupa iz ove oblasti. Zajednička vizija ove grupe je zapisana u **agilnom manifestu** (eng. *The Agile Manifesto*), i ona glasi:

Pojedinci i interakcija	...	preko procesa i alata
Softver koji radi	...	preko sveobuhvatne dokumentacije
Saradnja sa klijentima	...	preko pregovaranja kroz ugovore
Odgovor na promene	...	preko praćenja plana
Znači, postoji vrednost u stavkama zdesna, ali se više vrednuju stavke sleva.		

Pored agilnog manifesta Agilna Alijansa je razvila još jedan dokument pod nazivom **Agilni principi** (eng. *The Agile Principles*) koji se sastoji od dvanaest principa:

1. Najveći prioritet je zadovoljiti klijenta kroz stalne isporuke softvera.
2. Prihvatiti nove zahteve ili zahteve koji menjaju funkcionalnost softvera čak iako je proces razvoja softvera pri kraju.
3. Često isporučiti softver koji radi u periodima od nekoliko nedelja do nekoliko meseci.
4. Poslovni ljudi i programeri moraju raditi zajedno sve dok traje projekat.
5. Graditi projekat oko motivisanih pojedinaca, dati im potrebnu podršku, i verovati im.
6. Najefikasniji način protoka informacija između razvojnog tima i spoljašnjeg „sveta“ je konverzacija licem u lice.
7. Softver koji radi predstavlja primarnu meru napretka.
8. Agilni procesi promovišu konstantan ritam razvoja.
9. Stalna pažnja oko tehničkog savršenstva i dobrog dizajna pojačava agilnost.
10. Jednostavnost – umetnost da se maksimizira količina nezavršenog rada – je neophodna.
11. Najbolje arhitekture, zahtevi i dizajn pojavljuju se u samo-organizirajućim timovima.
12. U uobičajenim intervalima razvojni tim odslikava kako biti više efektivan, a zatim prilagođava svoje ponašanje u skladu sa tim.

Definisati bilo šta o agilnom razvoju je dosta težak posao zato što svi agilni metodi imaju svoje principe. Tako je i sa upravljanjem agilnim projektima. Jedan menadžer agilnog projekta se dosta razlikuje od uobičajenog menadžera projekta. Na primer, on ne pravi vremenske raspodele, i ne vrši procene – ovo radi celi razvojni tim. Dalje, on (u većini slučajeva) ne naređuje ljudima šta treba da rade. Umesto toga on naglašava „treniranje“ neiskusnih članova u timu, dostavljanje resursa, održanje vizije celog projekta, otklanjanje barijera, promovisanje agilnih principa, itd.

Kod agilnih metoda, komunikacija i povratna sprega su od presudnog značaja. Kod komunikacije se najviše ceni konverzacija licem u lice. Takođe je važno održavanje dobrog i zdravog morala unutar razvojnog tima, pa su kod mnogih agilnih metoda pojedinci važniji od samog procesa.

Prema *Ogunnaike* i *Ray*, svi procesi (ne samo procesi razvoja softvera) se mogu kategorizovati na *definisane* i na *empirijske*. Pre toga se mora napomenuti da bez obzira šta se proizvodi (softver, televizor, obuća, prehrambeni proizvod ili građevinski objekat), odgovarajući proces se može podeliti na *predvidljivu* ili *masovnu proizvodnju* i na *razvoj novog proizvoda*. Kod **predvidljive** ili **masovne proizvodnje** (eng. *Predictable Manufacturing* ili *Mass Manufacturing*) moguće je prvo završiti specifikacije i tek onda početi proizvodnju, unapred proceniti troškove, identifikovati i vremenski raspodeliti aktivnosti, itd. Kod **razvoja novog proizvoda** (eng. *New Product Development*) teško je na početku napraviti detaljnu specifikaciju (koja se vremenom neće menjati), teško je unapred proceniti troškove, a teško je identifikovati i aktivnosti. Većina stručnjaka smatra da razvoj softvera u većini slučajeva spada u drugu kategoriju (razvoj novog proizvoda). Prema tome, razvoj softvera više spada u empirijske procese. Naime, **definisan proces** (eng. *Defined Process*) se sastoji od skupa predefinisanih aktivnosti koje se moraju ispoštovati tokom razvoja, pa ovo više odgovara predvidljivoj ili masovnoj proizvodnji. Za razliku od definisanog procesa, **empirijski proces** (eng. *Empirical Process*) je baziran na čestom merenju i brzom odgovoru na promene, i zato se više koristi u „nestabilnim“ domenima. S druge strane, jedan proces može biti **zasnovan na principima** (eng. *Principle-Based*) i **na pravilima** (eng. *Rule-Based*). Umesto da se prati unapred definisani skup pravila o ulogama, odgovornostima i aktivnostima, celi tim i menadžer su vođeni agilnim principima i agilnim manifestom. Znači, agilni metodi su po klasifikaciji *empirijski* i *zasnovani su na principima*.

Najpoznatiji agilni metodi su *ekstremno programiranje* i *Scrum* metod. O njima će biti više reči nešto kasnije u poglavlju 1.5.

1.4. Zašto iterativni i agilni razvoj?

Danas se smatra da je klasični vodopadni model jedan rizičan model koji smanjuje produktivnost. Kao što je već u prethodnom poglavlju razjašnjeno, razvoj softvera u većini slučajeva nije *predvidljiv* proces. On je pun raznih promena i novosti, a s druge strane je još i veoma kompleksan. Softver se razvija mesecima ili čak godinama. Razviti jedan takav softver u jednoj iteraciji (tj. po klasičnom vodopadnom modelu) bi bio prevelik zalogaj za većinu razvojnih timova. Međutim, pokazalo se da vodopadni model dobro funkcioniše kod malih, kratkih projekata. Takođe, pokušaji da se jedan veći projekat razbije na manje celine (iteracije) i da se svaka iteracije radi po vodopadnom modelu, su isto bili uspešni. Iz svega ovoga se može zaključiti da je najveći nedostatak ovog klasičnog modela nedostatak kratkih iteracija. Razlozi zbog kojih dugačke iteracije ne uspevaju su mnogobrojni, a neki od njih su:

- **Nemogućnost da se naprave precizne procene i vremenski rasporedi za duže periode**
- **Promenljivi zahtevi** – klijenti retko mogu da odrede sve zahteve na početku razvoja softvera. Veliki procenat zahteva stiže mnogo kasnije, a po *Johnson*-u čak 45% zahteva koji se određuju na početku projekta se nikad neće koristiti.
- **Kasni početak testiranja** – po vodopadnom modelu, faza testiranja je četvrta po redu, a u slučaju velikih projekata to znači kasni početak testiranja (kod nekih projekata čak dve godine od početka projekta). Testiranje može dovesti do iznenadnih problema koji mogu dovesti do propadanja celog projekta. Zato testiranje mora da počne što pre.

Ključni faktori zbog kojih se iterativni razvoj smatra boljim od klasičnog vodopadnog modela su zaista mnogobrojni, ali najvažniji su sledeći:

- **Iterativni razvoj je manje rizičan od vodopadnog modela** – smatra se da iterativni razvoj povećava produktivnost i uspeh, a smanjuje rizik.
- **Rano otkrivanje rizičnih elemenata u razvoju** – iterativni razvoj promoviše rešavanje najozbiljnijih problema na početku razvoja (razvoj usredsređen ka riziku).
- **Lako savladavanje promena** – iterativni razvoj se ne bori protiv promena nego ih smatra svakodnevnim pojavama.
- **Upravljiva kompleksnost** – smatra se da su veliki i kompleksni softverski proizvodi rizičniji od manjih softvera. Iterativni razvoj ovaj problem rešava tako što ih dekomponuje na manje celine – iteracije.
- **Poboljšano poverenje i zadovoljstvo već od početka razvoja** – kratke i česte iteracije daju osećaj kompletnosti. Napredak u razvoju je vidljiv već od početka. Ovo je važno i zbog psiholoških razloga – lično zadovoljstvo i poverenje u timu povećaju produktivnost.
- **Rani delimični proizvodi** – klijenti mogu precizno pratiti napredak razvoja kroz delimične proizvode. Ovo povećava poverenje klijenata u razvojni tim.
- **Bolje praćenje napretka razvoja** – u vodopadnom modelu teško je proceniti napredak u razvoju, naročito u ranijim fazama razvoja. Rezultat jedne iteracije – iztestiran softver – je mnogo bolji indikator.

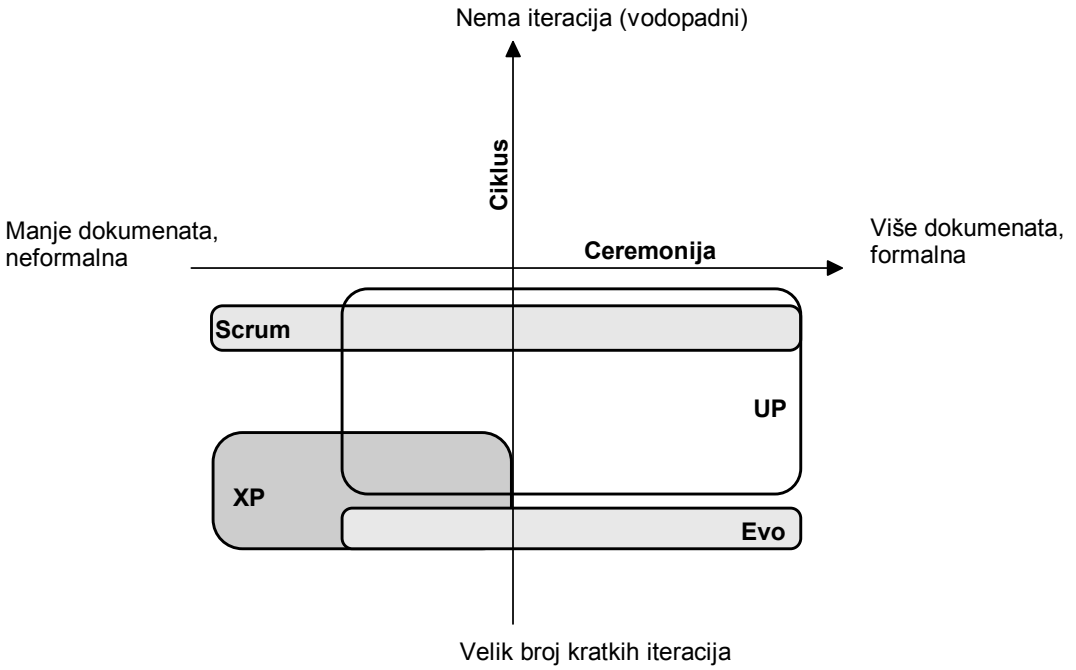
- **Visok kvalitet, manji broj grešaka** – iterativni metodi zahtevaju da se testiranje počne veoma rano.
- **Finalni proizvod više odslikava želje klijenta** – klijenti su aktivno uključeni u razvoj softvera i to za celo vreme trajanja razvoja.
- **Poboljšana komunikacija** – smatra se da veliki broj projekata propada zbog neadekvatne komunikacije između razvojnog tima i klijenta ili između paralelnih razvojnih timova (ukoliko je reč o velikom projektu).

Najvažnije pitanje je u svakom slučaju sledeće: da li vredi preći sa vodopadnog modela na iterativne i agilne metode? Pre svega se ne sme zaboraviti da i vodopadni model može biti dobar kod nekih timova. Prema tome, ako neki razvojni tim koristi vodopadni model, a istovremeno može da isporuči kvalitetan softver na vreme, uprava firme i klijenti su zadovoljni napretkom, a tim je produktivan, onda možda i nema potrebe preći na nešto novo. Međutim, ukoliko postoje problemi koji bi bili rešivi prelaskom na iterativni i agilni razvoj, onda treba isprobati i ove nove metode. Mora se napomenuti da prelazak sa starog modela na novi zahteva vreme i razumevanje. Za vreme prelaznog perioda moguće je da će produktivnost razvojnog tima pasti (umesto da raste), ali ako uprava firme ima strpljenje, a članovi tima su odlučni da žele savladati novi metod, onda će to u većini slučajeva i uspeti.

U sledećem poglavlju će biti više reči o iterativnim i agilnim metodima.

1.5. Iterativni i agilni metodi

U ovom poglavlju će biti malo detaljnije objašnjeni najpoznatiji iterativni i agilni metodi: *Evo*, *UP*, *ekstremno programiranje* i *Scrum*. Zajednička osobina ovih metoda je to da imaju relativno kratke *timebox*-ovane iteracije sa adaptivnim, evolutivnim profinjivanjem plana i zahteva. Mora se napomenuti da još ni dan danas ne postoji dogovor, koji je od ovih metoda agilan i koji je samo iterativan. Naime, iterativni razvoj je sam po sebi stariji od agilnog razvoja, pa se postavlja pitanje da li se *Evo* i *UP* – originalno kao predstavnici iterativnog razvoja – mogu smatrati i agilnim ili ne. *Evo* i *UP* se u striktnom smislu ne mogu smatrati agilnim, naročito u njihovim ranijim definicijama. Međutim, u današnje vreme su već oni dovoljno evoluirali da bi ljudi mogli da i njih smatraju agilnim.



Slika 3: Klasifikacija metoda prema stepenu ceremonije i po ciklusu

Iterativni i agilni metodi se mogu klasifikovati prema stepenu ceremonije i po ciklusu. **Klasifikacija prema stepenu ceremonije** (eng. *Degree of Ceremony*) određuje stepen poštovanja dokumenata, formalnih koraka i slično. Drugim rečima, ova klasifikacija određuje količinu dokumentacije i – generalno rečeno – nivo formalnosti i definisanosti celokupnog metoda razvoja. S druge strane, **klasifikacija po ciklusu** (eng. *Cycles*) predstavlja broj i dužinu iteracija. Raspoređivanje ovih metoda po klasifikacijama se može predstaviti na grafički način (slika 3). Po slici, klasični vodopadni model je na samom vrhu po ciklusu jer nema iteracije. S druge strane, *Evo* metod je na dnu po istoj klasifikaciji jer ima najkraće iteracije – svega nedelju dana. Ekstremno programiranje preporučuje da iteracija traje između nedelju dana i četiri nedelje, dok *UP* između dve i šest nedelja. Kod *Scrum*-a, jedna iteracija traje tačno trideset kalendarskih dana. Što se tiče druge klasifikacije (prema stepenu ceremonije), ekstremno programiranje je „najslobodnije“, dok su *Evo* i *UP* više formalizovani i zahtevaju više dokumenata. Zanimljivo je pogledati *Scrum* koji se prostire kroz celu

osu. Razlog tome leži u činjenici da ovaj metod „ćuti“ u vezi dokumenata, tj. nivo dokumentacije prepušta razvojnom timu. U svakom slučaju, kod svakog iterativnog i agilnog metoda važi jedan zaključak: „što manje dokumenata, to bolje“.

U nastavku će biti kratko prikazani prethodno spomenuti metodi.

1.5.1. Objedinjeni proces (UP) i racionalan objedinjeni proces (RUP)

Objedinjeni proces (eng. *Unified Process*, *UP*) je poznat i popularan iterativni metod, naročito njegova varijanta zvana **racionalni objedinjeni proces** (eng. *Rational Unified Process*, *RUP*). Koreni *UP*-a i *RUP*-a dopiru do *Barry Boehm*-a i njegovog tzv. *spiralnog modela*. Ovaj model je ostavio snažan utisak na neke saradnike firme *Rational Corporation* kao što su *Philippe Kruchten*, *Grady Booch*, *Mike Devlin*, *Rich Reitman* i *Walker Royce*, naročito zbog iterativne prirode ovog modela i zato što je usredsređen ka riziku. *UP* i *RUP* metod je nastao unutar firme *Rational* kao mešavina *Boehm*-ovog spiralnog modela i višedecenijskog iskustva saradnika unutar *Rational*-a. Najveći deo posla je urađen između 1995. i 1998. godine vođen od strane *Philippe Kruchten*. Firma *Rational* je uvidela zaradu u ovom modelu i tako je nastao *Rational Unified Process* (*RUP*), ali su kreatori izdali i uopšteniju verziju *RUP*-a koja je bila otvorenog tipa i nazvali su je *Unified Process* (*UP*). Ovi metodi su bili kompatibilni sa njihovim **objedinjenim jezikom modeliranja** (eng. *Unified Modeling Language*, *UML*). Prva knjiga o ovoj tematici je nastala 1999. godine („*The Unified Software Development Process*“ od *Ivar Jacobson*-a).

U smislu stepena ceremonije i ciklusa (slika 3), *UP* preporučuje iteracije između dve i šest nedelja. S druge strane, *UP* je formalniji u smislu dokumenata jer i u najmanjem obimu zahteva više dokumenata u odnosu na druge metode. Međutim, i ovde važi pravilo da „što je manje dokumenata, to bolje“. Čovek bi na prvi pogled mislio da po stepenu ceremonije nema velikih razlika između *UP*-a i *Scrum*-a, međutim, postoji jedna velika razlika između njih. Naime, iako i *Scrum* omogućava veoma visok nivo dokumenata, on ništa ne definiše u vezi njih. *UP*, s druge strane, ima preko pedeset predefinisanih dokumenata: određuje njihovu namenu, način korišćenja, itd. Naravno, njihovo korišćenje nije obavezno. Od tih dokumenata neki su: **model slučajeva korišćenja** (eng. *Use-Case Model*), **model dizajna** (eng. *Design Model*), **model podataka** (eng. *Data Model*), **plan iteracije** (eng. *Iteration Plan*), itd.

UP je u suštini samo generalni *kostur* (eng. *framework*) iterativnog procesa i zahteva konkretizaciju da bi funkcionisao u praksi. Jedna konkretizacija *UP*-a je npr. *RUP*. Kod svakog projekta *UP* mora da bude *prilagođen* za posmatrani projekat, što podrazumeva izbor dokumenata od skupa predefinisanih. Ovaj proces prilagođavanja se naziva **slučaj razvoja** (eng. *Development Case*).

Neki od ključnih principa *UP*-a su:

- Relativno kratke *timebox*-ovane iteracije
- Preporučuje se implementacija najtežih delova softvera odmah na početku
- Visok nivo ponovne upotrebe
- Promene što pre uvrstiti u sam projekat
- Razvojni tim kao celina

Na početku ovog odeljka je rečeno da u striktnom smislu *UP* nije agilan metod. Možda je najočiglednija razlika između *UP*-a i nekih agilnih metoda u njihovim

vrednostima. Kod *UP*-a čak i ne postoje unapred definisane vrednosti, ali se svakako mogu izvući iz konteksta. Neke vrednosti su:

- Važno je primeniti vodilje *UP*-a i njegove principe
- Bolje je najteže (najrizičnije) elemente implementirati odmah na početku kao i one elemente koji za klijenta imaju visoku tržišnu vrednost
- Važno je proces prilagoditi jedinstvenim potrebama posmatranog projekta
- Korisno je imati jedan dobro-definisan proces koji sadrži vodilje o aktivnostima, o listi dokumenata, itd.

Iz ovih vrednosti jasno se može zaključiti da one više promovišu sam projekat od pojedinaca.

UP se može primeniti kod različitih razvojnih timova. Kao minimum zahteva bar tri člana u timu, ali primenljiv je i u mnogo većim timovima – i do nekoliko stotina članova.

Najvažnije uloge unutar *UP*-a su:

- **Zainteresovane strane** (eng. *Stakeholders*) – klijenti, menadžer proizvoda, itd.
- **Programer** (eng. *Implementer*) – piše programski kod, testove, itd.
- **Test-osoba** (eng. *Tester*) – piše sistemske testove
- **Arhitekta softvera** (eng. *Software Architect*) – održava viziju arhitekture
- **Sistem-analitičar** (eng. *System Analyst*) – bavi se analizom zahteva
- **Dizajner baze podataka** (eng. *Database Designer*), **dizajner korisničkog interfejsa** (eng. *User Interface Designer*)
- **Menadžer projekta** (eng. *Project Manager*)
- **Inženjer procesa** (eng. *Process Engineer*) – definiše i profinjava slučaj razvoja

Kao i svaki metod, i *UP* ima svoje prednosti i probleme. Neke od prednosti su sledeće:

- Veliki naglasak na teške i rizične elemente
- Dobro-definisani i precizirani pomoćni dokumenti, zajednički rečnik
- Detaljne vodilje oko korišćenja metoda
- Laka *custom*-izacija metoda u skladu sa potrebama posmatranog projekta
- Preporučuje se i neiskusnim razvojnim timovima zbog preciznih dokumenata i vodilja unutar metoda. Kod iskusnih timova preporučuje se veća *custom*-izacija metoda čime se dobija veliki skok u smislu agilnosti.
- Odgovara i malim i velikim razvojnim timovima
- Naglašava korišćenje slučajeva korišćenja
- Široko primenjen metod, veliki izbor literature

Od problema najvažniji su sledeći:

- Jako puno detalja
- Kao što se *UP* može modifikovati da bude agilan, moguće je uraditi i obrnuto. Naime, neki razvojni timovi koriste *UP* u vodopadnom maniru što je pogrešno.
- Ne obraća dovoljnu pažnju na značaj komunikacije

1.5.2. Evolutivno upravljanje projektom (Evo)

Evolutivno upravljanje projektom (eng. *Evolutionary Project Management*, *Evo*) je verovatno najstariji *IID* metod sa značajnim agilnim i adaptivnim osobinama. Razvio ga je *Tom Gilb*, pionir u iterativnom i evolutivnom razvoju. *Gilb* je već 60-ih godina prošlog veka počeo da primenjuje neke iterativne i evolutivne principe (koji su bili dosta različiti u odnosu na uobičajene prakse). Svoje ideje je prvi put izneo 1976. godine, dok njegova knjiga „*Principles of Software Engineering Management*" iz 1988. predstavlja prvi ozbiljni pokušaj da se svet upozna sa iterativnom i evolutivnom filozofijom. Mnogi smatraju da se svi kasniji iterativni i agilni metodi (uključujući *UP*, *Scrum* i ekstremno programiranje) zasnivaju na njegovim idejama.

U smislu stepena ceremonije i ciklusa *Evo* je dosta specifičan (slika 3). Naime, ovaj metod preporučuje jako kratke iteracije u smislu ciklusa – svega nedelju dana (eventualno dve nedelje ako je potrebno). Što se tiče ceremonije, najviše liči na *UP*, mada postoje i očigledne razlike. Naime, *Evo* ne zahteva visoku preciznost pri pisanju dokumenata, a preporučuje se da se sve piše kratko i sažeto. Međutim, on može biti i veoma precizan jer za opis zahteva može da koristi jedan strukturni jezik koji je razvijen baš za ove svrhe: **Planguage**[2]. Od korišćenih dokumenata neki su: **specifikacija zahteva** (eng. *Requirement Specification*), **specifikacija dizajna** (eng. *Design Specification*), ***Evo*-plan** (eng. *Evo Plan*), **specifikacija zahteva performansi** (eng. *Performance Requirement Specification*), itd.

Evo promoviše postepen razvoj prvo onih zahteva kod kojih klijent smatra da su najvažniji (*razvoj usredsređen ka klijentima*). Jedna od prepoznatljivih ideja ovog metoda je merenje **zahteva performansi** (eng. *Performance Requirements*). *Performans* (ili doprinos) sadrži **zahteve kvaliteta** (eng. *Quality Requirements*) kao što je pouzdanost, **zahteve kapaciteta posla** (eng. *Workload Capacity Requirements*) kao što je propustljivost i **zahteve štednje resursa** (eng. *Resource Savings Requirements*).

Evo je zasnovan na pet vrednosti koje su:

- Savladavanje posmatranog projekta aktivnim i realnim merenjem
- Razvoj usredsređen ka klijentima
- Biti skroman oko kompleksnih sistema – razlagati probleme na manje celine i rešavati ih postepeno kroz iteracije
- Stavljanje naglaska na rezultate, ne na formalnost
- Nagraditi razvojni tim u skladu sa merljivim rezultatima

Važno je naglasiti da *Evo* nije zamišljen samo za softverske projekte. Iz ovoga se može zaključiti da ovaj metod može biti primenljiv za različite projekte različitih veličina. Veličina razvojnog tima isto može da varira.

Pošto se *Evo* može primeniti kod različitih projekata (ne samo za razvoj softvera), on definiše samo generičke uloge. Najvažnije uloge su:

- **Vlasnik** (eng. *Owner*) – odgovoran za specifikacije za sledeću iteraciju
- **Zainteresovana strana** (eng. *Stakeholder*) – prima rezultate na kraju svake iteracije

[2] Termin *Planguage* predstavlja igru reči i sastoji se od kombinacije dve reči: plan (eng. *Plan*) i jezik (eng. *Language*).

- **Programer** (eng. *Implementer*) – piše programski kod
- **Arhitekta sistema** (eng. *Systems Architect*) – odgovoran za odluke oko arhitekture sistema
- **Menadžer projekta** (eng. *Project Manager*) – odgovoran za merenje rezultata

Kao svaki metod, i *Evo* ima svoje prednosti, ali i svoje probleme. Neke od prednosti su:

- Vidljivi rezultati već od početka, česta izdanja klijentima
- Naglasak na merenju
- Aktivno učestvovanje klijenata kroz trajanje projekta
- Korišćenje jezika *Planguage*
- Lako se meša sa drugim iterativnim i agilnim metodima

Najveći problemi *Evo*-a su:

- Previše naglašava merenje
- Previše je uopšten (pošto nije namenjen samo za razvoj softverskih proizvoda)

1.5.3. Scrum

Scrum je jednostavan ali funkcionalan agilan metod. Koreni *Scrum*-a dopiru do 1986. godine. Te godine se pojavio jedan članak od *Hirotaka Takeuchi*-a i *Ikujiro Nonaka* pod nazivom „*The New New Product Development Game*" (objavljen unutar *Harvard Business Review*). Ovaj članak je izložio najbolje prakse i principe korišćene unutar deset inovativnih kompanija u Japanu. Autori su koristili engleski termin „*scrum*"[3] za adaptivne i samo-upravljive timske prakse i principe. *Jeff Sutherland* je bio fasciniran ovim člankom i jednim izveštajem iz 1994. (objavljen unutar „*Borland Software Craftsmanship: A New Look at Process, Quality, and Productivity*" od *J. Coplien*) koji je predstavio jedan jako produktivan projekat unutar firme *Borland Corporation* koji je efektivno koristio dnevne sastanke. *Sutherland* je 1994. godine radeći u firmi *Easel Corporation* počeo da razvija *Scrum*. Godinu dana kasnije mu se pridružio i *Ken Schwaber*. Njihovi rezultati su bili objavljeni 1995. godine u članku „*The Scrum Development Process*", a kasnije su finalizirani u knjigama „*SCRUM: A Pattern Language for Hyperproductive Software Development*" 1998. godine i „*Agile Software Development with Scrum*" 2002. godine.

U smislu stepena ceremonije i ciklusa (slika 3), *Scrum* je jako specifičan u vezi ciklusa: svaka iteracija traje tačno trideset kalendarskih dana i svaka iteracija se zove **sprint** (eng. *Sprint*). Međutim, za razliku od ciklusa u kojem je *Scrum* dosta strog, prema klasifikaciji po stepenu ceremonije funkcioniše skroz obrnuto jer ništa ne specificira: sve to prepušta razvojnom timu. Jedino što su kreatori *Scrum*-a ostavili *Scrum*-timovima je: „što manje dokumenata, to bolje". Ovo se nikako ne može smatrati nemarom jer omogućava timovima da rade jako različite projekte: od razvoja softvera u medicini ili u prehrambenoj industriji (stroga dokumentacija i formalnost) do razvoja korisničkih Web-prezentacija (malo dokumenata). Svaki *Scrum*-projekat bi (kao

[3] Reč „*scrum*" u ragbiju označava adaptivnu, prilagodljivu sposobnost sportskog tima da prenese loptu kroz stazu. Inače ova reč je jedna varijanta termina „*scrimmage*" koji se može prevesti kao: sudar, borba (prsa u prsa), koškanje, čarka, komešanje, itd.

minimum) trebao da ima tzv. **zalihe proizvoda** (eng. *Product Backlog*) koje predstavljaju listu svih stavki projekta: funkcije, slučajeve korišćenja, probleme, defekte, poboljšanja, tehnologije, itd. Za razliku od ovog dokumenta koji sadrži listu svih stavki za *celi* projekat, postoji još jedan dokument koji sadrži istu listu ali samo za *trenutnu* iteraciju tj. sprint: **zalihe sprinta** (eng. *Sprint Backlog*).

Neki od ključnih principa *Scrum*-a su:

- samo-upravljiv i samo-organizirajući tim
- bez dodatnih zahteva kad se jednom počne iteracija
- dnevni sastanci
- jedna iteracija traje trideset kalendarskih dana
- demonstracija proizvoda na kraju svake iteracije

Scrum promoviše samo-upravljive razvojne timove i dnevne sastanke, i naglašava važnost principa u upravljanju projektom. Zasnovan je na pet vrednosti:

- **Obećanje** – kad *Scrum*-tim jednom obeća upravi da će implementirati sve zahteve do kraja iteracije, tim dobija autoritet i autonomiju za postizanje postavljenih ciljeva na onaj način na koji taj tim smatra da je najbolje. Niko više sa spolja ne može da doda nove zahteve. Menadžer ne upravlja timom, nego pomaže da rad prođe sa što manje problema (dostavlja potrebne resurse, otklanja barijere, itd.).
- **Fokus** – razvojni tim se fokusira na postavljene ciljeve unutar trenutne iteracije.
- **Otvorenost** – otvorenost *zaliha proizvoda* omogućava svima da pregledaju rad i prioritete. Dnevni sastanci omogućavaju članovima uvid u napredak projekta.
- **Poštovanje** – svi članovi razvojnog tima su poštovani bez obzira na njihove vrline i mane, i nisu žigosani zbog neuspelih iteracija. Problemi pojedinaca su problemi celog tima, a menadžer pomaže da se otklone ti problemi.
- **Hrabrost** – menadžer ima hrabrost da planira tako da ne upravlja timom. Drugim rečima, menadžer ima poverenje u svom timu. S druge strane, razvojni tim ima hrabrost da prihvata odgovornost za samo-upravljanje timom.

Jedan *Scrum*-tim se uglavnom sastoji od sedam članova ili manje. U međuvremenu su počeli da ga koriste i u velikim timovima sa nekoliko stotina članova. *Scrum* se može koristiti i u velikim projektima tako što će se razvojni tim razbiti na *podtimove*. Svaki podtim je jedan poseban *Scrum*-tim koji radi uobičajeno, s tim da menadžer komunicira i sa menadžerima drugih timova.

Najvažnije uloge unutar *Scrum*-a su:

- **Vlasnik proizvoda** (eng. *Product Owner*) – kreira zalihe proizvoda i određuje prioritet svake stavke unutar nje, određuje ciljeve za sledeći sprint tj. iteraciju, itd. Drugim rečima, on je klijent projekta.
- **Scrum-tim** (eng. *Scrum Team*) – pored razvoja softvera ovaj tim kreira i osvežava zalihe sprinta. Termin „član tima“ se retko koristi.
- **Scrum-gospodar** ili **menadžer projekta** (eng. *Scrum Master*) – predstavlja mešavinu člana tima i uobičajenog menadžera. Znači, on učestvuje i u razvoju proizvoda i u upravljaju. Komunikacija između tima i uprave firme se vrši pomoću njega. Takođe, on upravlja dnevnim sastancima.

- **Pilići** (eng. *Chickens*) – svi koji nemaju pravo govora samo posmatranja. U *Scrum*-slengu oni koji imaju pravo govora se zovu **svinje** (eng. *Pigs*). Članovi tima se smatraju svinjama.

Scrum je metod koji ima svoje prednosti, ali i probleme. Neke od prednosti su sledeće:

- Jednostavni principi. Potrebni dokumenti (*zalihe proizvoda*, *zalihe sprinta*) se lako kreiraju.
- Samo-upravljanje. Problem pojedinca je problem celog tima.
- Otvorenost celog projekta kroz dokumente
- Lako se kombinuje sa drugim metodima

Od problema najvažniji su:

- Previše se naglašava samostalnost tima i štura komunikacija sa „pilićima“. Rezultat ove filozofije je (po nekima) nedovoljna komunikacija sa vlasnikom proizvoda. Često se dešava da je vlasnik proizvoda ujedno i ekspert u posmatranom domenu, dok u samom timu nema eksperata u posmatranom domenu. U tom slučaju razvoj proizvoda se nepotrebno otežava.
- *Scrum* je „ćutljiv“ u vezi dokumenata i prepušta razvojnom timu da određuje stepen dokumentovanja, što možda daje preveliku slobodu početnim *Scrum*-timovima. S druge strane, nedefinisanost i nekonzistentnost dokumenata otežava njihovu ponovnu upotrebu u kasnijim projektima.

1.5.4. Ekstremno programiranje (XP)

Ekstremno programiranje (eng. *Extreme Programming*, *XP*) je jedan od najpoznatijih agilnih metoda. Razvio ga je *Kent Beck* zajedno sa *Ward Cunningham* sredinom 80-ih godina prošlog veka kad su radili u istraživačkoj grupi firme *Tektronix*. Osnovi *XP*-a su se rodili iz ove saradnje. *Beck* je kasnije nastavio sa istraživanjem *XP*-a dodajući nove principe i tako ga je kompletirao. U tome je umnogome pomogao jedan projekat sredinom 90-ih unutar *Chrysler*-a zvan *C3* u kojem je i sam *Beck* bio angažovan. *C3* je bio jedan sistem platnog spiska baziran na objektno-orijentisanom programskom jeziku *Smalltalk*. *Beck* je u projekat ubacio svoje principe, a u tome su mu pomagali *Ron Jeffries* i *Martin Fowler*. *Beck* je 1999. godine izdao prvu ediciju *XP*-a u svojoj knjizi „*Extreme Programming Explained, First edition*“, dok je drugu ediciju izdao već 2004. godine u knjizi „*Extreme Programming Explained, Second edition*“.

Kako *Beck* priznaje, reč „*ekstremno*“ je dolazila iz njegovog uverenja da je teško imati previše dobrih stvari unutar razvoja softvera. Zato ako već postoje prakse i principi koji su očigledno dobri i funkcionalni, zašto ih ne forsirati do maksimuma, do ekstrema? U skladu sa ovim sledi:

- Dobro je testirati, zato treba da se uradi test modula na celom programskom kodu i test prihvatanja za svaku funkciju.
- Dobro je pregledati kod, a još bolje ako se to radi odmah posle kreiranja tog koda. Iz ovoga je nastalo programiranje u parovima.
- Dobro je vršiti čestu integraciju koda, još bolje ako to radi celi razvojni tim. Zašto ne bi onda postojao neki kompjuterski alat koji bi ovo radio automatski?
- Kratke iteracije i prevremena povratna sprega su dobre, zato treba te iteracije skratiti da traju svega nedelju dana ili dve nedelje.

- Dobro je uključiti klijenta u razvoj softvera. Zašto ih onda ne pozvati da celo radno vreme ne provode sa razvojnim timom?
- Komunikacija je dobra stvar, oralna komunikacija je još bolja. Onda zašto ne rušiti zidove i napraviti jednu veliku prostoriju u kojoj bi svi sedili zajedno bez obzira na njihove uloge?

U smislu stepena ceremonije i ciklusa (slika 3), *XP* preporučuje iteracije između nedelju dana i četiri nedelje, i jako mali broj dokumenata.

XP ima veliki broj principa, a neki od njih su: igra planiranja, programiranje u parovima, standardi kodiranja, stalna integracija, refaktorisanje, itd.

XP naglašava saradnju između članova tima, brz razvoj softvera koji radi, i mnoštvo agilnih principa radi što bržeg reagovanja na promene. Zasnovan je na četiri vrednosti, a to su:

- **Komunikacija** – *XP* je uvideo da je jedan od vodećih razloga za neuspeh projekata nedostatak komunikacije ili spora komunikacija.
- **Jednostavnost** – programirati tako da zadovolji samo trenutne potrebe.
- **Povratna sprega** – zahteva se povratna sprega tj. mišljenje svih članova tima bez obzira na njihove uloge.
- **Hrabrost** – razvojni tim mora imati hrabrost da razvije softver brzo i efikasno i da brzo reaguje na promene.

XP se uglavnom primenjuje kod *malih* razvojnih timova: ne više od dvadeset članova u timu. Doduše, danas se već koristi i u većim timovima ali samo sa iskusnim *XP*-ekspertima.

Najvažnije uloge unutar *XP*-a su:

- ***On-site* klijenti** (eng. *On-site Customers*) – klijenti koji provode vreme zajedno sa razvojnim timom tokom celog radnog vremena. Softver treba da izgleda tako da zadovolji njihove zahteve. Pišu zahteve, određuju njihov prioritet, ali pomažu i u testiranju.
- **Programeri** (eng. *Programmers*) – pišu programski kod, testove. Identifikuju zadatke i procenjuju ih.
- **Test-osobe** (eng. *Testers*) – pomažu *on-site* klijentima da napišu svoje testove, a takođe pišu i svoje specijalizovane testove.
- **Treneri** (eng. *Coaches*) – pomažu početnicima da što brže usvoje principe *XP*-a.

Mnoštvo agilnih principa unutar ovog metoda pomaže programerima, pa i celom razvojnom timu, da u rekordnom vremenu reaguju na promene, čak iako se te promene dešavaju na kraju projekta. *XP* je orijentisan ka komunikaciji u celom razvojnom timu: *on-site* klijentima, programerima, test-osobama, itd. Znači, svi rade zajedno u istoj prostoriji. Zbog ove činjenice broj pisanih, formalnih dokumenata je smanjen na apsolutni minimum jer su sve potrebne osobe na dohvat ruke: ako neko ima pitanje o bilo čemu, treba samo da pita. Praktično jedini pomoćni, ne-kompjuterizovani alati korišćeni u ovom metodu su mali listići ili ceduljice od kartona zvane **stori-listići** (eng. *Story Card*s).

Kao i svi metodi, *XP* ima svoje prednosti, ali i svoje probleme. Neke prednosti su:

- Praktične razvojne tehnike koje brzo daju rezultate
- Naglašava aktivno učestvovanje klijenta
- Programeri procenjuju zadatke, pa tek se onda ovi zadaci raspoređuju vremenski, a ne obrnuto
- Dostavlja *on-site* klijentima da napišu testove prihvatanja

Problemi unutar *XP*-a mogu biti:

- Zahteva prisustvo klijenata (*on-site* klijenti). Ovo nije uvek moguće, što će otežati oralnu komunikaciju između njih i ostatka tima. *XP* ne definiše načine kojima se ovo može rešiti.
- Većina agilnih principa unutar *XP*-a radi u simbiozi, i nažalost to je i jedan od glavnih razloga što još neiskusni *XP*-timovi ne uspevaju. Naime, zbog ove simbioze potrebno je zajedno primeniti sve principe, šta više, izostaviti neke principe zbog bilo kakvih razloga je rizično jer smanjuje uspešnost *XP*-tima. Početnici nisu svesni ovog rizika i zbog toga su često razočarani sa rezultatima.
- Novi članovi se teško uklapaju u tim jer nema pisanih dokumenata koji se mogu uzeti za savladanje zahteva ili arhitekture softvera.
- Neki programeri ne žele da programiraju u parovima.
- Neke firme zahtevaju više dokumenata. *XP* nema definisano rešenje za ovaj problem.

Ekstremno programiranje će biti detaljnije opisano u drugom delu ovog rada.

2. Ekstremno programiranje

U prvom delu rada su bili opisani iterativni i agilni razvoj zajedno sa četiri njihova predstavnika. Jedan od njih je i *ekstremno programiranje* koje će detaljnije biti opisano u ovom delu rada. Za uvod o ekstremnom programiranju pogledati odeljak 1.5.4.

Kao što je već rečeno, ekstremno programiranje (*XP*) je razvio *Kent Beck*. Prvu ediciju svoje knjige „*Extreme Programming Explained*" izdao je 1999. godine, a drugu ediciju 2004. godine dok. Druga edicija je donekle izmenjena, neki principi su dodati, a neki su se stopili sa drugima. Međutim, u ovom radu će se koristiti jedna unapređena varijanta *Beck*-ove druge edicije ekstremnog programiranja razvijena od strane *James Shore*-a i *Shane Warden*-a. Velikih razlika između ove varijante i *Beck*-ove druge edicije nema. Razlike su:

- Dodati su neki novi principi
- Neki principi su izbačeni jer se pokazalo da nemaju veliku primenu u praksi
- Neki principi su izdvojeni da stoje posebno
- Nazivi nekih principa su promenjeni

Naravno mora se napomenuti da bez obzira na sve ove vodilje, ekstremno programiranje (kao i agilni razvoj generalno) ostaje jedan *kreativan* proces koji uvažava činjenicu da je svaki softver jedinstven proizvod, pa se proces prilagođava u skladu sa tim. Međutim, da bi čovek mogao da stigne do nivoa da „krši pravila", mora prvo dobro da razume oblast, a za tako nešto su neophodni višegodišnja praksa i iskustvo.

Ovaj deo će početi udubljivanjem u ekstremno programiranje. Kao prvo, mora se naći motivacija za njegovu primenu . Ovde će se definisati pojam uspeha, i kako može *XP* da ga poboljša. Zatim će biti detaljno opisan životni ciklus *XP*-a. Posle toga sledi navođenje uloga ovog metoda sa detaljnim objašnjenjem svake od njih. Zatim će biti navedena kratka, ali važna terminologija koja se često koristi kod ekstremnog programiranja. Kratko upoznavanje sa *XP*-om će se završiti korisnim savetima, koji bi mogli da pomognu novim timovima pri prelasku na ekstremno programiranje. Suština *XP*-a se nalazi u velikom broju principa. Preostala poglavlja će se baviti opisom ovih principa koji su grupisani u sledeće grupe: *razmišljanje*, *saradnja*, *izdavanje*, *planiranje* i *razvoj*.

2.1. Uvodne napomene

2.1.1. Motivacija

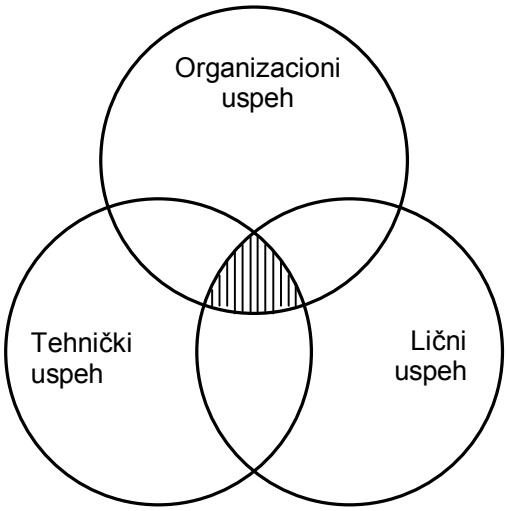
Glavni razlog zbog čega razvojni timovi prelaze na neki agilni metod nije samo poboljšanje produktivnosti, nego nešto sasvim drugo. Naime, jedino pitanje koje vredi postaviti je: „Da li će razvojni tim prelaskom na neki agilni metod biti *uspešniji*?". Pojam *uspeha* se definiše kao sposobnost tima da isporuči proizvod na vreme, u okviru budžeta, i da bude u skladu sa specifikacijom. Po grupaciji *Standish*, jedan projekat može biti:

- **Uspešan** (eng. *Successful*) – završen na vreme, u okviru budžeta, sa svim zamišljenim funkcijama i osobinama
- **Osporen** (eng. *Challenged*) – završen, ali sa prekoračenjem budžeta, vremenskog roka ili sa manjim brojem zamišljenih funkcija i osobina

- **Oštećen** (eng. *Impaired*) – projekat je obustavljen

Međutim, ove definicije nisu baš realistične. Naime, jedan projekat može biti uspešan čak iako softver ne zaradi ni dinar (pošto nije zadovoljio potrebe klijenata) ili može biti osporen (prekorači budžet i kasni sa isporukom) čak iako će kasnije donositi milione dolara prihoda (zato što ga korisnici obožavaju). Razlog ovih neslaganja leži u činjenici da postoje različiti oblici uspeha:

- **Lični uspeh** (eng. *Personal Success*) – svaki razvojni tim je sastavljen od pojedinaca. Jedan program koji radi ili iskustvo stečeno pri pisanju programa (čak iako program ne radi) predstavlja neki oblik „personalne nagrade“. Lični uspeh čini pojedinca srećnim. Srećni pojedinci čine srećan tim.
- **Tehnički uspeh** (eng. *Technical Success*) – kad je tim sposoban da razvije kompleksan softver, to predstavlja tehnički uspeh jer članovi tog tima znaju da je taj softver u suštini rezultat profesionalnog timskog rada, pisanja elegantnog programskog koda koji se lako održava, itd.
- **Organizacioni uspeh** (eng. *Organizational Success*) – bez obzira na prethodne oblike uspeha, ne sme se zaboraviti da jedan razvojni tim funkcioniše unutar neke firme. Softverski proizvod mora da zadovolji upravu te firme uključujući i sponzore (tj. klijente) posmatranog projekta. Njihovo zadovoljstvo je zapravo organizacioni uspeh.



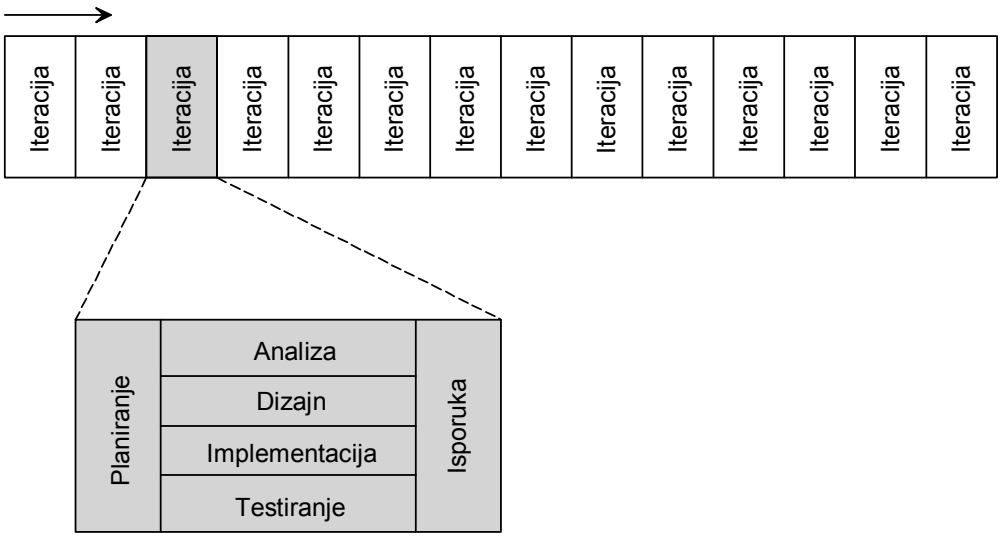
Slika 4: Lični, tehnički i organizacioni uspeh

Lako se može primetiti da su ovi oblici uspeha međusobno povezani (slika 4). Bez ličnog uspeha nema motivacije pojedinaca što može negativno uticati na tehnički uspeh, a samim tim i na organizacioni uspeh. Bez tehničkog uspeha softver će biti osporen što može dovesti do raznih problema sa upravom firme ili sa klijentima. Problematični klijenti ili uprava mogu dovesti do padanja morala članova tima, što će ponovo smanjiti sve oblike uspeha. Agilni metodi fokusiraju se na postizanje visokog nivoa kod sva tri oblika uspeha.

2.1.2. Životni ciklus kod ekstremnog programiranja

Životni ciklus ekstremnog programiranja se dosta razlikuje u odnosu na uobičajene modele. Jedna karakteristika *XP*-a (a i drugih agilnih metoda) je izbegavanje formalnih i pisanih dokumenata. Međutim, zašto se forsira ovako nešto kad se zna da su specifikacija zahteva, model proizvoda, arhitektura softvera, i drugi dokumenti presudni za uspeh projekta? Odgovor leži u činjenici da *XP* zapravo ne izbegava ove dokumente već primenjuje jedan drugačiji pristup: umesto pisanih dokumenata koristi „*verbalne*“ dokumente. Takođe, umesto da postoje posebne faze u životnom ciklusu kao kod uobičajenih metoda, *XP* ih sklapa sve zajedno i primenjuje ih svaki dan. Sve ovo je moguće zahvaljujući visokom stepenu komunikacije. Jedna iteracija kod *XP*-a može da traje od nedelju dana do tri nedelje (a u retkim slučajevima i duže, mada se to ne preporučuje). Kratke iteracije omogućavaju razvojnom timu lako i brzo prilagođavanje korisničkim zahtevima.

Slika 5 prikazuje kako izgleda tipičan životni ciklus kod ekstremnog programiranja. Kao što se može videti, svaka iteracija se sastoji iz nekoliko faza: planiranje, analiza, projektovanje (dizajn), implementacija (kodiranje), testiranje i isporuka. Međutim, samo faza planiranja i isporuke stoji posebno na početku i na kraju iteracije; sve ostale faze se dešavaju simultano. Pošto su iteracije relativno kratke, može se čak reći da su praktično sve faze simultane. U nastavku će biti dat malo detaljniji opis spomenutih faza.



Slika 5: Faze unutar jedne iteracije

Faza planiranja (eng. *Planning*) je prva faza u svakoj iteraciji. U ovoj fazi *on-site klijenti* (klijenti koji sede celo vreme sa razvojnim timom dajući im podršku kada god je to potrebno) odrede zahteve koje oni žele videti u isporučenom proizvodu na kraju trenutne iteracije. Ovo je zgodno zato što će ovako razvojni tim brže implementirati one elemente proizvoda koji imaju visoku tržišnu vrednost. Kad *on-site* klijenti odrede zahteve za implementaciju u trenutnoj iteraciji, programeri procenjuju te zahteve i određuju šta se može realno implementirati do kraja iteracije. Ovaj proces se naziva **igra planiranja** (eng. *The Planning Game*). U ovoj fazi se određuje i procena rizika.

Faza analize (eng. *Analysis*) je u uobičajenim metodima pisanje specifikacije zahteva. Kod *XP*-a ova faza funkcioniše drugačije. *On-site* klijenti sede zajedno sa timom celo vreme i uvek su na raspolaganju kad neko ima neko specifično pitanje (a to je veoma često).

U **fazi projektovanja (dizajna)** (eng. *Design*) i u **fazi implementacije (kodiranja)** (eng. *Coding*) programeri koriste *inkrementalni razvoj* za kreiranje arhitekture softvera, tj. dizajniranje se vrši u malim koracima. Da bi ovako nešto uspelo, programeri koriste jedan specijalan način razvoja koji se naziva **razvoj usredsređen ka testiranju** (eng. *Test-Driven Development*, *TDD*) koji vešto kombinuje dizajn sa programiranjem i testiranjem. *TDD* bolje funkcioniše kad dva programera rade zajedno. Ovaj princip se naziva **programiranje u parovima** (eng. *Pair Programming*).

Faza testiranja (eng. *Testing*) se u *XP*-u shvata ozbiljno, zato u testiranju učestvuju ne samo test-osobe, nego i programeri i *on-site* klijenti. Prva linija odbrane je uključena već u *TDD* u vidu testa modula i testa integracije. *On-site* klijenti su odgovorni za tzv. **klijentske testove** (eng. *Customer Tests*) – apstraktne testove koji dokazuju da neka funkcija u programu radi tako kako je klijent to zamislio. Oni su takođe odgovorni za testiranje korisničkog interfejsa. Test-osobe rade tzv. **istraživačko testiranje** (eng. *Exploratory Testing*) radi otkrivanja rupa u programu (ali i za testiranje stabilnosti i performansi).

Faza isporuke (eng. *Deployment*) je poslednja faza svake iteracije. U ovoj fazi razvojni tim mora da isporuči softver koji radi. Ovo izdanje softvera bi trebalo da sadrži sve one zahteve koji su definisani na početku iteracije, i isporučuje se uglavnom unutrašnjim zainteresovanim stranama, a ponekad i spoljašnjim korisnicima.

Kad razvojni tim stigne do finalnog proizvoda, isti taj tim može preuzeti na sebe i odgovornost *održavanja* tog proizvoda. Ne postoji posebna faza za održavanje, nego se uglavnom koriste iste faze kao i ranije (mada su moguće blage izmene). Međutim, održavanjem proizvoda može se baviti i neki drugi tim. U tom slučaju originalni razvojni tim kreira dokumentaciju i održava kurseve za novi tim.

Kao što je već poznato, agilni metodi definišu skup **principa** (eng. *Practices*) pomoću kojih se dostiže potrebni nivo agilnosti. *XP* ima veliki broj takvih principa. Više reči o ovim principima će biti u narednim poglavljima.

2.1.3. Uloge ekstremnog programiranja

Svi članovi jednog razvojnog tima imaju svoje uloge. Uloge unutar *XP*-a nisu skroz disjunktne jer postoje i zajedničke aktivnosti kao što su razni sastanci i slično. Ipak, kad ljudi nisu na raznim sastancima, svi rade svoj posao. Uloge unutar *XP*-tima se mogu grupisati u četiri velike grupe: *on-site* klijenti, programeri, test-osobe i treneri. U nastavku će biti pojašnjena svaka grupa posebno.

Prva grupa uloga je grupa ***on-site*** **klijenata**[4] (eng. *On-Site Clients*). Oni su odgovorni za definisanje softverskog proizvoda. Oni ne moraju obavezno biti i pravi klijenti ili druge zainteresovane strane, ali najbolje razumeju njihove želje i interese. Najvažnija funkcija *on-site* klijenata je planiranje: određivanje vizije projekta, identifikacija zahteva kao i određivanje njihovog prioriteta, upravljanje rizikom, itd. Druga njihova važna uloga je pružanje pomoći programerima oko detalja zahteva. Pošto *XP* nema posebnu specifikaciju zahteva, *on-site* klijenti služe kao „*živa*“ specifikacija

[4] Često se koristi i skraćeni termin „klijenti“.

zahteva, i oni su na raspolaganju celom razvojnom timu u svakom trenutku. Oni su važni i kod testiranja jer su odgovorni za kreiranje *klijentskih testova* pomoću kojih se može pokazati da neka određena funkcionalnost radi tako kako su je klijenti zamislili. Naravno, sve ovo ne bi funkcionisalo bez ozbiljne komunikacije, pa je već jasno zašto je važno da i klijenti sede zajedno sa ostatkom tima. *On-site* klijenti se mogu podeliti na menadžere proizvoda, domenske stručnjake, dizajnere interakcije, i poslovne analitičare.

Menadžer proizvoda (eng. *Product Manager*) ima samo jedan, ali veoma težak zadatak: održavanje i promovisanje *vizije* proizvoda. Ovo uključuje dokumentovanje te vizije, predstavljanje vizije upravi firme, definisanje i profinjivanje zahteva, određivanje njihovih prioriteta, pomoć ostalim *on-site* klijentima, pregledanje napretka proizvoda, vođenje demonstracija proizvoda na krajevima iteracija, reklamiranje i promovisanje finalnog proizvoda, i bavljenje organizacionom politikom unutar firme. Dobar menadžer proizvoda poznaje tržište, zna šta je potrebno tržištu, poznaje konkurenciju, i ume da donosi i veoma teške odluke. Bez menadžera proizvoda – nema proizvoda. Svakom projektu je potreban tačno jedan menadžer proizvoda.

Da bi softver mogao da ima vrednost, mora da dokaže da je dovoljno profesionalan u tom domenu. Međutim, programeri u većini slučajeva nemaju dovoljno znanje o tom području. Zato su **domenski stručnjaci (eksperti)** (eng. *Domain Experts*) važni. Oni određuju detalje kod zahteva i pišu klijentske testove.

Dizajneri interakcije (eng. *Interaction Designers*) pomažu pri definisanju korisničkog interfejsa. Korisnički interfejs (eng. *User Interface*, *UI*) se kod većine korisnika izjednačava sa samim proizvodom. Njihova uloga je da procene kako bi korisnici proizvoda želeli da koriste taj proizvod: intervjuišu korisnike, prave ankete sa njima o njihovom mišljenju o proizvodu, a često ih i nadgledaju pri korišćenju samog softvera. Ne treba mešati dizajnera interakcije sa grafičkim dizajnerom.

Poslovni analitičar (eng. *Business Analyst*) služi kao pomoćni *on-site* klijent koji pomaže drugim *on-site* klijentima. On profinjuje korisničke zahteve tako što će kroz razgovor skupiti zaboravljene podatke.

Po iskustvu, optimalan razmer između programera i klijenata je 3:2 (tj. na tri programera, dva *on-site* klijenta). Mnogi bi smatrali da je ovo previše, ali je ovakva srazmera ipak potrebna jer *on-site* klijenti imaju puno posla i uvek moraju da budu na raspolaganju ostatku tima, a s druge strane, uvek moraju biti i jedan korak ispred programera. Jako je važno napomenuti da od ukupnog broja *on-site* klijenata u timu, jedan od njih *mora* biti menadžer proizvoda.

Druga grupa uloga je grupa **programera** (eng. *Programmers*). Ako je zadatak klijenata da povećavaju tržišnu vrednost softverskog proizvoda, onda je zadatak programera da smanjuju njegove troškove. Oni su odgovorni za pronalaženje najefikasnijeg načina za implementaciju zahteva. Ipak, najviše se bave programiranjem. Programeri uglavnom programiraju u parovima koristeći razvoj usredsređen ka testiranju (*TDD*), pišu testove, ali se bave i dizajnom proizvoda. Ako imaju neka pitanja, umesto da nagađaju moguća rešenja, dovoljno je da pitaju *on-site* klijente.

Jedan razvojni tim be trebao da ima četiri ili šest programera, i po mogućstvu njihov broj bi trebao da bude paran da bi svi mogli da programiraju u parovima. Takođe se preporučuje da bar jedan od njih bude iskusan programer (*trener-programer*).

Treću grupu uloga predstavljaju **test-osobe** (eng. *Testers*). Test-osobe pomažu razvojnom timu da isporuči softver bez grešaka. Oni pišu *istraživačke testove* radi otkrivanja nedostataka u proizvodu. Oni takođe testiraju i stabilnost i performanse softvera. Treba napomenuti da su test-osobe samo pomoćnici, tj. oni nisu odgovorni za

pronalaženje grešaka: svi u timu su odgovorni. Takođe je važno reći da test-osobe samo pronalaze greške, ne ispravljaju ih. Kad test-osoba pronađe neku grešku, ona će pomoći ostatku tima u pronalasku odgovora na to šta je pošlo naopako i šta se može preduzeti da se ovo sledeći put ne desi.

Najbolji razmer između programera i test-osoblja je 4:1 (tj. na svaka četiri programera dolazi jedna test-osoba). Neki bi smatrali da je njihov broj premali, ali ne treba zaboraviti da se svi članovi tima bave testiranjem.

Poslednja grupa uloga je uloga **trenera** (eng. *Coaches*). Iako se smatra da je *XP*-tim samo-organizirajući (što znači da svaki član sam određuje kako može najbolje da pomaže timu da uspe), to ne znači da razvojnom timu nije potreban jedan *lider*. Međutim, uloga ovog lidera nije ista kao kod uobičajenih metoda. Naime, umesto da definiše i podeli zadatke, on će pomoći timu da postigne svoje ciljeve. On će poduzeti potrebne mere da bi tim imao odgovarajuću prostoriju, odrediće sastav tima, igraće ulogu posrednika između uprave firme i tima, ali će pomoći i u svim poslovima oko planiranja i oko programiranja. Pošto se ovo dosta razlikuje od uobičajenog lidera, *XP*-lideri se zovu *trenerima*.

Treneri se mogu podeliti na trenere-programere i na menadžere projekta.

Treneri-programeri (eng. *Programmer-Coaches*) su iskusni programeri koji pomažu drugim programerima u timu. Njihov glavni cilj je promovisanje agilnih principa u programiranju kao što su: programiranje u parovima, *TDD*, itd.

Menadžer projekta (eng. *Project Manager*) pomaže da tim i uprava firme nađu zajedničku reč. Ovo podrazumeva aktivnu komunikaciju sa upravom firme, međutim, ne sme se pomešati menadžer projekta sa menadžerom proizvoda. Naime, dok se uloga menadžera proizvoda svodi na komunikaciju sa upravom firme *o proizvodu*, menadžer projekta komunicira sa upravom firme *o XP-projektu*. Ovo je potrebno jer uprava firme u većini slučajeva još ne razume dovoljno kako *XP* funkcioniše. Na primer, treba upravi objasniti zašto je timu potrebna zajednička prostorija, ili zašto se ne piše specifikacija zahteva, itd. Pored ove komunikacije sa upravom firme, menadžer projekta može da pomaže u bilo kom poslu oko planiranja.

Preporučuje se da svaki razvojni tim ima jednog trenera-programera i – ako je moguće i ako je potrebno – i jednog menadžera projekta.

2.1.4. Korišćena terminologija

Ekstremno programiranje, kao i svi drugi metodi, ima svoj rečnik pojmova. Najčešće korišćeni pojmovi (pored pojma iteracije o kome je već bilo reči nekoliko puta) su sledeći:

- **Refaktorisanje** (eng. *Refactoring*) – predstavlja proces izmene strukture programskog koda bez menjanja njegovog značenja ili ponašanja. *XP* stavlja veliki naglasak na ovo jer je idealan način za popravljanje kvaliteta izvornog koda.
- **Tehnički dug** (eng. *Technical Debt*) – predstavlja ukupnu količinu ishitrenih dizajnerskih i implementacionih odluka u projektu. Kad se primeti neka greška u izvornom kodu, ta greška se može ispraviti *brzo* ili *pravilno*. Brzo ispravljanje podrazumeva ubacivanje prljavog koda da bi program mogao *odmah* da radi. Pravilno ispravljanje podrazumeva malo dublju analizu koda da bi ispravljen izvorni kod mogao da radi i *sutra*. Prljavi kod se lako identifikuje u vidu dugačkih metoda, puno programskih linija u komentaru ili u vidu tekstualnih komentara tipa „`// ovo radi, ali ne znam zašto`“. Prljavi kod je veliki izvor

budućih grešaka. Koristi se pojam „tehnički dug“ jer se zna da se greške skupo ispravljaju. Nagomilavanje prljavog koda stvori tehnički dug jer će kasnije skupo koštati firmu. *XP* se aktivno bori protiv tehničkog duga u vidu raznih principa: jedan od njih je i refaktorisanje.

- ***Timebox*-ovanje** (eng. *Timeboxing*) – već je bio reči o *timebox*-u u poglavlju 1.2. *Timebox* označava da je krajnji datum tj. završetak posmatrane iteracije u potpunosti *fiksiran*, i ne može se naknadno produžiti. *Timebox*-ovanje pomaže timu i članovima tima da budu realistični.
- **Poslednji odgovorni trenutak** (eng. *The Last Responsible Moment*) – predstavlja poslednji trenutak da se donese neka odluka *bez* gubitka nekih alternativa. Ovo može biti zgodno jer se dodatno vreme može iskoristiti za sakupljanje novih informacija i podataka, ili čak novih alternativa pri donošenju odluke. Sve ovo povećava verovatnoću da se donosi najbolja moguća odluka. *XP* voli da koristi ovu praksu, pa se može reći da je XP *lenj* u donošenju odluka, ali ne treba ovaj termin mešati sa poslednjim *mogućim* trenutkom.
- **Storije** (eng. *Stories*) – predstavljaju pojedinačne elemente projekta. U većini slučajeva se izjednačavaju sa korisničkim zahtevima, ali su dovoljno pojednostavljene da bi mogle da se reše za dan ili dva. Važno je napomenuti da su storije usredsređene na korisnike i klijente, a ne na programere. Storije se pišu na male listiće od kartona koji se zovu **stori-listići** (eng. *Story-Cards*).
- **Brzina** (eng. *Velocity*) – predstavlja broj završenih storija u iteraciji. Naime, procena truda programera je ravnomerna, ali ne i tačna. Takođe, mogući su razni prekidi u njihovom radu koji onemogućavaju da procene truda budu u skladu sa kalendarskim vremenom. Brzina je elegantan način da se procene preslikavaju na kalendar. Kod dobrih *XP*-timova brzina je konstantna kroz iteracija.

2.1.5. Prelazak na ekstremno programiranje

Razvojni timovi koji žele da pređu na ekstremno programiranje moraju se dodatno pripremiti. Pre prelaska treba proveriti da li su zadovoljeni sledeći preduslovi:

1. **Podrška uprave** – jako je teško primeniti *XP* ukoliko uprava firme nije s ovim saglasna. Pošto *XP* zahteva ozbiljne promene u odnosu na uobičajen razvoj softvera, potrebna je aktivna podrška firme. Na primer, *XP* preporučuje korišćenje jedne velike zajedničke prostorije. Takođe je potrebno objasniti upravi da ne očekuje pregršt pisanih dokumenata. A pre svega uprava mora da ima strpljenje i razumevanje da prelazak sa jednog metoda razvoja softvera na novi zahteva vreme i da će za vreme tranzicije produktivnost tima verovatno opasti.
2. **Podrška tima** – kao što je važno dobiti podršku uprave, isto tako je važno dobiti saglasnost celog razvojnog tima. Tim koji ne želi preći na *XP* verovatno ni neće preći, pa u tom slučaju prelazak ne treba forsirati. Članovi koji su protiv će ili napustiti tim ili će namerno sabotirati trud preostalih članova.
3. **Tim koji sedi zajedno** – mnogi smatraju da je ovo jedan od ključnih faktora zbog kojih je *XP* uspešan. Bliskost članova tima poboljšava komunikaciju.
4. ***On-site* klijenti** – mogućnost da klijenti sede zajedno sa timom i da budu „pri ruci“ uvek kad neko ima pitanje za njih je presudna za uspeh projekta.
5. **Adekvatna veličina tima** – *XP* ne preporučuje timove veće od dvadeset članova (ali ni ne isključuje). Optimalna veličina tima je dvanaest članova: šest programera (uključujući i trenera-programera), četiri *on-site* klijenata, jedna test-osoba i jedan menadžer projekta. Inače, preporučuje se da broj programera bude

paran da bi programiranje u parovima bilo uspešno. Najmanja veličina tima je pet članova: četiri programera (uključujući i trenera-programera) i jedan menadžer proizvoda. S druge strane, najveća veličina tima je dvadeset članova: deset programera, šest *on-site* klijenata, tri test-osobe i jedan menadžer projekta. Naravno, moguće je angažovati i veće timove, ali u tom slučaju skup osnovnih principa *XP*-a se mora dodatno proširiti nekim novim principima. Takvo ekstremno programiranje se tada zove **napredno ekstremno programiranje** (eng. *Advanced XP*).

6. **Koristiti sve principe** – jedan od najčešćih razloga zbog kojeg novi *XP*-timovi ne uspevaju je mišljenje da je logičnije postepeno uvesti principe *XP*-a. Međutim, istina leži u suprotnom pravcu. Naime, svi principi unutar *XP*-a su na neki način povezani, i direktno utiču na celokupni razvoj softvera.

U narednim odeljcima će biti objašnjeni principi ekstremnog programiranja.

2.2. Grupa principa broj 1: Razmišljanje

Dobri programeri ne samo da programiraju kod koji radi, oni postavljaju sebi pitanje, zašto radi, probaju da ga dublje analiziraju, a zatim da ga unaprede da bi mogli da ga koriste po potrebi i u budućnosti. U ovom poglavlju će biti predstavljena pet principa koja pomažu programerima da unaprede te sposobnosti: programiranje u parovima, stimulisan rad, informativno radno okruženje, iskonska analiza i retrospektive.

2.2.1. Programiranje u parovima

Programiranje u parovima (eng. *Pair Programming*) duplira obim mozga jer programer radi u paru a ne sam. Programiranje u parovima – dve osobe i jedan kompjuter sa jednom tastaturom i mišem – je jedna od najprepoznatljivih praksi unutar *XP*-a.

Ovaj način programiranja zahteva dve osobe: jednog vozača i jednog navigatora. **Vozač** (eng. *Driver*) piše kod (kodira), a **navigator** (eng. *Navigator*) razmišlja. Ponekad razmišlja o tome šta vozač trenutno piše (nadgledanje), i obaveštava vozača u slučaju da je napravio grešku. Pored toga, navigator razmišlja i malo unapred o stvarima koje slede, a ponekad i o tome kako bi sve ovo moglo da se integriše u celinu. Zahvaljujući tome, vozaču je omogućeno da se više skoncentriše na detalje realizacije programskog koda jer ne mora da razmišlja o „globalnoj slici". S druge strane, sve ovo omogućava navigatoru strategijsko razmišljanje na višem nivou apstrakcije bez upletanja u sitne tehničke detalje. Rezultat zajedničkog rada je kvalitetniji programski kod. Po studiji *Cockburn*-a i *Williams*-a, oni koji programiraju u parovima, moraju da ulože 15% više truda od onih koji sami programiraju, ali će brže postići rezultate i imaće 15% manje grešaka.

Uparivanje poboljšava programerske navike zato što navigator sedi pored vozača (rezultat ovoga je smanjenje tehničkog duga). S druge strane, oni koji programiraju u parovima su više skoncentrisani iz prostog razloga zato što nisu tako izloženi raznim smetnjama: ako se javi neka smetnja (potrebna je pomoć nekoj osobi, zvoni telefon, itd.) jedan od programera će rešiti smetnju dok drugi može dalje da se skoncentriše i da nastavi sa radom.

Ne treba praviti rasporede uparivanja. Parovi se formiraju sasvim prirodno: jedan programer inicira uparivanje sa kolegom i drugi ga prihvata[5]. Takođe, parovi ne traju dugoročno: treba menjati partnere bilo kad, kad je to poželjno. Na primer, kad se završi rad na trenutnom zadatku ili kad se javi neka barijera koju ne može rešiti ni vozač ni navigator. Jedno uparivanje traje u proseku manje od četiri sata. Preporučuje se uparivanje sa svim programerima u timu, što poboljšava unutrašnje odnose tima i povećava osećaj timskog rada. Uloge vozača i navigatora treba isto menjati. U jednom uparivanju uloge se menjaju posle svakih nekoliko minuta (ne više od pola sata).

Kod ovog tipa programiranja važna je komunikacija između vozača i navigatora. Oba programera razmišljaju glasno: vozač govori o tehničkim detaljima, dok navigator

[5] Vredi spomenuti da uparivanje dva programera nije jedini način uparivanja. Programer može biti u paru i sa test-osobom (radi pisanja testova) ili sa *on-site* klijentom (radi objašnjenja rada neke funkcije).

govori o tome šta sledi dalje. Kad programeri primećuju da je komunikacija pogoršana, to je možda znak da treba da nađu drugog partnera.

Oni koji još nisu programirali u parovima verovatno će se osećati neprijatno na početku. Vozaču će možda biti neprijatno zbog osećaja da ga navigator posmatra. Možda će se osećati i inferiornim (a možda čak i nesposobnim) u odnosu na navigatora jer će imati osećaj da navigator sve zna pre njega. Treba istaći da je to sasvim normalno, i da će u obrnutim ulogama i situacija biti obrnuta. S druge strane, navigator će često imati želju da „isčupa“ tastaturu iz ruke vozača zbog raznih razloga: sporog kucanja, sporog razmišljanja, pravljenja grešaka, itd. U svakom slučaju ne sme se zaboraviti da u jednom paru nema nadređenog i podređenog; svi su ravnopravni.

Postavlja se pitanje, kako se pripremiti za ovaj način programiranja. Odgovor je jednostavan: **parne stanice** (eng. *Pairing Stations*). Jedna takva stanica se sastoji od jednog kompjutera sa jednom tastaturom, mišem i dve stolice. Važno je da oba programera imaju dovoljno mesta da sede udobno, pa se preporučuje sto širine oko dva metra. Stolice treba da budu jedna pored druge (osećaj ravnopravnosti) a ne jedna iza druge (osećaj nadgledanja i podređenosti). Naravno, moguće su razne varijacije i sve zavisi od ličnih afiniteta. Većina programera smatra da su bar dva miša potrebna, a najbolje je imati i dve odvojene tastature. Neki čak smatraju da je najbolje imati i dva monitora (ili da se na oba monitora prikazuje ista slika ili da se ekran proširuje na drugi monitor). Ako zbog nekih razloga nije moguće priključiti dva monitora na isti računar, preporučuje se korišćenje velikog monitora širokog (*wide-screen*) ekrana.

2.2.2. Stimulisan rad

Stimulisan rad (eng. *Energized Work*) uzima u obzir činjenicu da su ljudi najproduktivniji kad su stimulisani i motivisani. Ako čovek svakog dana ustane iz kreveta sa rečenicom „*Ponovo moram na posao*“ ili „*Ne ide mi se*“ onda to je znak da mu fali stimulacija i motivacija.

Stimulisan rad se sastoji iz dva suprotna vremenska perioda: vreme *na poslu* i vreme *van posla*. Ključna reč kod vremena na poslu je *fokusiranje*. Treba posvetiti potpunu pažnju na poslu, i smatra se da je smanjenje raznih smetnji najbolji način da se to realizuje: isključiti *Chat*, ne proveravati privatni e-mail, isključiti mobilne telefone, i slično. Takođe je moguće zamoliti menadžera projekta da zaštiti tim od nepotrebnih sastanaka. Stimulisan rad na poslu nije moguć bez kvalitetnog vremena van posla. Kao prvi korak preporučuje se da svaki dan posle isteka radnog vremena ljudi odu kući na vreme. Sve ostalo je već stvar privatne sfere. Najbolje je raditi takve stvari koje pozitivno odvraćaju pažnju od posla: kvalitetan provod sa porodicom ili sa prijateljima, baviti se hobi-aktivnostima, itd. Kvalitetno vreme van posla se završava adekvatnim odmorom.

Sve ovo zvuči jednostavnije nego što jeste. Postići stimulisan rad u razvojnom timu je težak posao jer zahteva podršku i u poslovnoj i u privatnoj sferi. Neke ideje mogu biti sledeće:

- **Podsetiti ljude da idu kući na vreme** – umorni ljudi nisu potrebni jer prave više grešaka i tehničkog duga, pa mogu da prave veću štetu nego koristi. Ako se neko razboli, treba da ostane kod kuće.
- **Koristiti programiranje u parovima** – poboljšava fokusiranje pojedinaca.
- **Zdrava ishrana** – brza hrana se ne preporučuje jer izaziva umor u popodnevnim satima.

- **Objasniti razvojnom timu viziju projekta** – kad članovi tima znaju, zašto je trenutni projekat toliko važan, ultimativni cilj projekta može da im da snagu i motivaciju za fokusiran rad.
- **Realističan plan** – realističan, ali pouzdan plan poboljšava raspoloženje razvojnog tima jer je zadovoljan napretkom.
- **Zaštititi razvojni tim od spoljašnjih „napada“** – menadžer projekta bi trebao da zaštiti tim od uprave firme. Međutim, konstruktivni sastanci su poželjni.
- **Sprijateljiti se sa ostatkom razvojnog tima** – imati prijatelje na radnom mestu, šta više, sprijateljiti se sa celim timom (ići zajedno na ručak, druženje van posla) je dosta teško, ali ako uspe, može biti jedan od najmoćnijih faktora za stimulisan rad.

Primećuje se da neke ideje i nisu baš u skladu sa današnjom praksom. Ovo se naročito odnosi na preporuku da svi odu kući na kraju radnog vremena. Naime, u većini današnjih društava prekovremeni rad je normalna stvar, čak i poželjna. U suštini, ekstremno programiranje ne isključuje mogućnost prekovremenog rada, ako to ne ugrožava stimulisan rad. U nekim momentima (na primer, pri kraju iteracije) uz prethodno odobravanje od strane razvojnog tima može se organizovati tzv. noćna žurka kodiranja na kojoj učestvuje celi tim. Ovo čak može da ima i pozitivne efekte: s jedne strane svi članovi su odlučni da završe posao do kraja iteracije, a s druge strane može da učvrsti prijateljstvo između članova. Međutim, konstantni prekovremeni rad može da napravi veću štetu nego korist. Po *DeMarco*-u, konstantan (ili čest) prekovremeni rad smanjuje produktivnost, kvalitet, izaziva fizičku i psihičku iscrpljenost kod pojedinaca, i smanjuje efektivno korišćenje normalnog radnog vremena.

2.2.3. Informativno radno okruženje

Informativno radno okruženje (eng. *Informative Workspace*) daje celom timu više mogućnosti da uvidi šta funkcioniše, a šta ne. Radno okruženje je centralno mesto kreativnog razmišljanja radi razvijanja kvalitetnog softvera, pa treba i da izgleda u skladu sa tim. Takvo radno okruženje se zove informativno radno okruženje.

Lako se može primetiti razlika između zdravog i nezdravog radnog okruženja. Zdravo radno okruženje po ekstremnom programiranju uvek vibrira ali ne od napetosti već od aktivnosti: ljudi pričaju o projektu, rade zajedno, ponekad se čuje neka šala, kad neko traži pomoć ubrzo se javlja neko da pomogne, itd. Ritam rada nije užurban, ali je svakako odlučan. S druge strane, nezdravo radno okruženje je relativno tiho: oseća se napetost, nema puno komunikacije između članova, ljudi često gledaju na sat.

Jedno informativno radno okruženje potpomaže ljudima da komuniciraju. Zidovi prostorije su natrpani tablama i listićima od kartona. Na tablama su nacrtani razni grafikoni i dijagrami koji pružaju informacije o samom projektu.

Na kraju treba napomenuti da se ne isplati kompjuterizovati ove grafikone. Neki grafikoni moraju biti stalno vidljivi, pa bi to zahtevalo nabavku skupih projektora ili velikih tankih ekrana. Takođe, oni se brže ažuriraju ručno.

2.2.4. Iskonska analiza

Iskonska analiza (eng. *Root-Cause Analysis*) je koristan alat za identifikaciju stvarnih uzroka problema. Kad nešto pođe naopako u projektu, prirodna reakcija čoveka je frustracija i potreba da se nađe krivac – neka *osoba*. Međutim, ovo neće rešiti problem. Pravi krivac je u suštini – sam *proces*.

Ultimativni cilj iskonske analize je da eliminiše pravi uzrok problema – da se ovako nešto ne desi u budućnosti. Da bi ovo proradio, koristi se tehnika postavljanja pitanja „*Zašto?*“ pet puta. Jedan primer može biti i sledeći:

„Naš tim često ima problem sa sklapanjem koda u program koji radi.

Zašto? Zato što naš program često ne može da se kompajlira.

Zašto? Zato što ljudi integrišu programski kod bez testiranja.

A zašto integrišu kod u celinu bez testiranja? Zato što testovi traju predugo i nema vremena da sačeka kraj testova.

Zašto testovi traju tako dugo? Zato što svaki test provede previše vremena u bazi podataka.

Zašto testovi provedu toliko vremena u bazi podataka? Zato što program ima previše provera u bazi podataka, iako bi u normalnim slučajevima polovina od njih bila dovoljna.“

Iz ovog primera se može lako zaključiti da je jedna dizajnerska greška stvorila lavinu skroz drugih grešaka. U većini slučajeva ljudi bi se zaustavili posle drugog pitanja „*Zašto?*“ i okrivili bi programere zato što ne testiraju kod pre integrisanja. Savladavanje ove tehnike zahteva praksu, ali vredi jer ima jako široku primenu.

2.2.5. Retrospektive

Retrospektive (eng. *Retrospectives*) predstavljaju način analiziranja i poboljšanja celog procesa razvoja. Ne postoji savršen proces. Razlog tome leži u činjenici da je svaki razvojni tim jedinstven, ali ovo važi i za situacije sa kojima se tim suočava. Što je još gore, one se stalno menjaju. Za uspešnu borbu protiv ovih problema potrebno je menjati i proces razvoja, i u tome retrospektive u vidu sastanaka mogu pomoći.

Postoji više tipa retrospektiva. Najčešći tip je **retrospektiva iteracije** (eng. *Iteration Retrospective*) koja se dešava na kraju svake iteracije[6]. Kod iskusnih timova retrospektiva traje tačno jedan sat (*timebox*-ovano). Važno je još napomenuti da se očekuje da na sastanku bude prisutan celi tim.

Postoji više načina vođenja retrospektive iteracije. Način koji će biti naveden u nastavku se sastoji iz četiri tačke. Retrospektive se u ovom slučaju izvode na sledeći način:

1. **Norm Kerth-ova primarna direktiva** (eng. *Norm Kerth's Prime Directive*) – retrospektiva se nikad ne sme koristiti za omalovažavanje i napad na pojedince – svi članovi tima su dali sve od sebe u skladu sa njihovim sposobnostima. Zato je potreban neki tip garancije da će se članovi tima na sastanku ponašati u skladu sa tim. Jedna ideja je i to da se napiše *Norm Kerth-ova primarna direktiva* na tabli koja glasi:

   „Bez obzira šta ćemo danas otkriti, mi razumemo i zaista verujemo da su svi dali sve od sebe u skladu sa njihovim znanjem u to vreme, njihovim veštinama i sposobnostima, dostupnim resursima i datom situacijom.“

   Treba pitati sve prisutne posebno da li se slažu sa direktivom. Očekuje se verbalno i glasno „*Da*“ od svih prisutnih. Ukoliko se neko ne slaže, preporučuje se odustajanje od retrospektive.

[6] Postoje i drugi tipovi kao što su retrospektive izdanja ili retrospektive projekta.

2. **Brainstorming** (30 minuta) – ako se svi slažu sa primarnom direktivom, treba napisati sledećih šest kategorija na tablu: „*ugodno*“, „*frustrirajuće*“, „*zagonetno*“, „*manje*“, „*više*“, „*isto*“. Treba prisutnima podeliti karton-listiće i olovke da napišu sva dešavanja u toku iteracije koja su bila ugodna, frustrirajuća i zagonetna. Takođe trebaju da napišu i pojmove i događaje koje treba manje ili više praktikovati ili ih ne treba menjati. Svaki pojam, misao ili događaj se piše na poseban listić. Prisutni pročitaju svoje listiće i predaju vođi sastanka da ih zalepi na tablu (svaki listić se stavlja kod odgovarajuće kategorije).
3. **Nemo mapiranje** (eng. *Mute Mapping*) (10 minuta) – služi za kategorizaciju pojmova. Predstavlja neku vrstu traženja zavisnosti, s tim da nema pričanja. Znači, glavni cilj je pomerati nalepljene kartone i listiće na tabli, i tako grupisati pojmove. Treba pozvati prisutne da ustanu i da probaju da grupišu listiće. Kad su prisutni zadovoljni sa rezultatom, treba ih zamoliti da sednu, a zatim vođa sastanka uzima marker i crta krugove oko grupa formirajući skupove. Posle toga, treba zamoliti tim da posebno imenuje svaku grupu (kategoriju). Kad je svaka grupa imenovana, treba ih ocenjivati tako da svaki član tima glasa za neku grupu. Kategorija sa najviše broja glasova pobeđuje. Ta kategorija predstavlja trenutno najveći problem tima koji se mora rešiti u sledećoj iteraciji. Ukoliko nema pravog pobednika (više grupa je dobilo isti broj glasova), proizvoljno se bira jedna grupa (na primer bacanjem novčića).
4. **Cilj retrospektive** (eng. *Retrospective Objective*) (20 minuta) – kad je izglasan pobednik, svi ostali listići se uklone i skoncentriše se na pobedničku grupu. Sad treba odrediti kako se može rešiti ovaj problem pomoću poboljšanja procesa. Ovo je idealni trenutak za iskonsku analizu, ali vredi uključiti i ostale učesnike. Rezultat iskonske analize može biti jedna ili više ideja. Pobednička ideja će biti *cilj retrospektive*. Cilj retrospektive će biti jedan od zadataka u sledećoj iteraciji.

Retrospektive predstavljaju odličan alat za poboljšanje procesa u skladu sa trenutnim problemima, ali ne treba zaboraviti ni to da one mogu biti i veoma opasne u slučaju da se ne izvedu na pravilan način. Najveći problem je to da se mogu iskoristiti za međusobno optuživanje. U tom slučaju pojedinačni privatni razgovor sa tim ljudima može pomoći. Drugo rešenje može biti angažovanje profesionalnog rukovodioca za vođenje sastanka. Ako ni to ne može pomoći, onda možda ne vredi održati retrospektive.

2.3. Grupa principa broj 2: Saradnja

Kod razvoja softvera sve se vrti oko *informacija*. Što efikasnije programeri pristupaju i shvate potrebne informacije, to efikasnije mogu da razviju softver. Efikasan pristup i adekvatna količina informacija ima veliku korist i kod klijenata i kod menadžera, jer im s jedne strane omogućava bolje upravljanje vremenskom raspodelom, a s druge strane su više pripremljeni da odgovore na pitanja programera. U ovom odeljku će biti predstavljeno osam principa koji pomažu timu i svim zainteresovanim stranama da efikasno sarađuju: poverenje, sedenje zajedno, uključivanje stvarnih klijenata, zajednički jezik, stojeći sastanci, standardi kodiranja, demonstracija iteracije i izveštavanje.

2.3.1. Poverenje

Poverenje (eng. *Trust*) je ključni pojam bez koga nema napredovanja. Svaka grupa prolazi kroz razne faze razvoja (*Tuckman*)[7]. Potrebno je dosta vremena da tim prođe kroz ove faze.

Postavlja se pitanje šta je potrebno da bi tim prošao kroz prethodno navedene faze. Pre svega, tim mora da ima odgovornost za proizvod koji se razvija. Članovi tima moraju da misle o drugim članovima kao „*mi*“ a ne „*oni*“. Pojedinac će zato ispraviti sve nađene greške bez obzira ko ih je napravio. Ako je problem ozbiljniji, tražiće pomoć. Sve ovo je nemoguće bez *poverenja*. Članovi tima moraju imati poverenje jedan u drugog. S druge strane, potrebno je da i uprava kompanije ima poverenje u tim.

Za poboljšanje poverenja između članova tima mogu se koristiti sledeći ideje:

- **Međusobno razumevanje programera i klijenata** – izgradnja dobrog odnosa između programera i klijenata je od suštinskog značaja. Ovo je inače tipičan problem u razvojnim timovima. Klijenti misle da programeri ne posvećuju dovoljno pažnje njihovim zahtevima koji su prema njihovom mišljenju od suštinskog značaja. S druge strane, programeri misle da klijenti uopšte ne razumeju koliko se teško nešto implementira, i da se previše slobodno igraju sa rokovima. Ovaj problem se može rešiti stavljanjem programera i klijenata u istu prostoriju, tj. treba zamoliti klijente da budu *on-site* klijenti. Ovo je najbolji način da programeri uvide da je i život klijenata težak jer svaki klijent ima svog nadređenog koga ništa ne zanima sem rezultata. S druge strane, klijenti će ovako uvideti koliko je teško nešto implementirati, ali i to da će stroži vremenski rokovi samo još više usporiti produktivnost tima.
- **Međusobno razumevanje programera i test-osoblja** – treba izgraditi dobre odnose i između programera i test-osoblja. Programeri uglavnom potcenjuju ulogu test-osoblja, dok test-osobe ili misle da su programeri nepažljivi, ili konstantno imaju osećaj da su nepoželjni u prostoriji jer se njihov rad svodi na pronalaženje grešaka u tuđim radovima. Programeri moraju da razumeju da test-osobe nisu tu da ih ponize i da je uvek bolje primetiti greške pre isporuke softvera, a ne posle isporuke. Takođe je važno uvideti da test-osobe imaju dobru metodologiju otkrivanja grešaka koja se razvija višegodišnjom praksom. S druge strane, test-

[7] Te faze uključuju formiranje (stvaranje grupe, eng. *Forming*), stormiranje (konflikti, eng. *Storming*), normiranje (konsenzus, eng. *Norming*) i izvođenje (produktivni rad, eng. *Performing*).

osobe moraju da uvide da svako može da pogreši. Kad se pronađe neka greška, to nije razlog za slavlje, nego prilika da se ljudima objasni šta se može učiniti da se ta greška ne desi u budućnosti.

- **Zajednički obroci** – jesti zajedno i uživati u međusobnom društvu može lako da poboljša međusobne odnose u timu. Jedino je važno da tim jede za istim stolom.
- **Kontinuitet tima** – kad jedan razvojni tim završi softver i isporuči ga, u većini slučajeva se raspušta. To je zato što uprava firme posmatra ljude kao resurse. Kad se pojavi neki novi projekat, uprava formira novu grupu. Ova nova grupa će morati ponovo da pređe kroz sve faze, što je – na neki način – trošenje vremena. Osnovna ideja iza kontinuiteta tima je da se timovi ne raspuštaju. Znači, umesto da se pojedinci tretiraju kao resursi, celi tim se tretira kao resurs. Kad se kreira novi projekat, tim će se dodeliti projektu a ne pojedinci.

Poboljšanje poverenja između članova tima je jedna stvar, ali treba imati na umu da sve zavisi od uprave firme. Razviti poverenje sa upravom firme je važan, ali težak posao, naročito kod agilni metoda. Sledeće ideje mogu biti od pomoći za poboljšanje poverenja prema timu od strane uprave firme:

- **Dokazati da tim vredno radi** – jedan energičan tim pozitivno deluje na upravu firme. Stimulisan rad, informativno radno okruženje i demonstracije iteracije su samo neki od *XP*-principa koji mogu biti od pomoći.
- **Ispunjavanje obećanja** – uprave u većini slučajeva ne znaju puno o razvoju softverskih proizvoda, ali znaju da procene rezultate. Ono što oni cene je softver koji radi i ispunjavanje obećanja. *XP* je dobar u ovim poljima. Naime, na početku iteracije razvojni tim da obećanje da će implementirati definisane zahteve. Na kraju iteracije prikaže softver u vidu demonstracije iteracije, pokazujući da tim ume da ispuni obećanja. Ovo je možda najefikasniji način poboljšanja poverenja između tima i uprave.
- **Upravljanje problemima** – ne postoji projekat bez problema. Zato je najbolje najteže zahteve i zadatke uraditi već na početku projekta. Kad neko naiđe na neki veći problem, trebalo bi odmah obavestiti ostatak tima. Neki problemi će rezultovati promenom samog plana pojednostavljenjem ili izbacivanjem nekih zahteva. U tom slučaju treba obavezno informisati i upravu firme. Uprava firme neće biti oduševljena, ali će svakako ceniti dodatni napor da je tim obavestio upravu o problemu.
- **Poštovanje klijentskih ciljeva** – zamoliti klijente da budu *on-site* klijenti i sede zajedno sa programerima i test-osobama nije lako. Da bi „tranzicija“ bila bezbolna, treba zamoliti programere da više poštuju klijentske ciljeve – čak iako bi to značilo odustajanje od ciničnih programerskih viceva o vremenskim rokovima. S druge strane, treba zamoliti klijente da se ne žale oko vremenskih rokova i da se ne svađaju sa programerima oko procene zahteva.
- **Promovisanje tima** – dobar način da se dokaže kompetentnost tima je otvorenost prema celoj kompaniji. Tim može da zakači neke svoje grafikone o napretku projekta na zidove firme. Takođe je dobra ideja pozvati celu upravu i sve zainteresovane da učestvuju na demonstracijama iteracije.
- **Biti iskren** – bez obzira koliko je promovisanje tima važno, treba biti uvek iskren o napretku projekta. Zataškavanje poznatih problema i grešaka kod demonstracije iteracije i hvalisanje mogućnostima softvera koje nisu stoprocentno završene radi

trenutne slave može imati negativne efekte u budućnosti. Razlog je očigledan: ostali će misliti da je tim jako uspešan, pa će očekivati sličan rezultat i na sledećoj demonstraciji iteracije. Relativno loši rezultati na sledećim demonstracijama će razočarati zainteresovane strane.

2.3.2. Sedenje zajedno

Sedenje zajedno (eng. *Sit Together*) čini komunikaciju brzom i preciznijom. Razmena poruka putem elektronske pošte je možda najsporiji način komunikacije. Razmena poruka putem *Chat*-a je mnogo brža, ali je daleko od idealne. Telefoniranje i tele-konferencije predstavljaju još bolje rešenje, ali ni ovi vidovi komunikacije nisu tako efikasni kao komunikacija licem-u-lice. Pošto je adekvatna komunikacija jedan od ključnih faktora za uspeh projekta, svi modeli procesa moraju da imaju neku ideju za poboljšanje komunikacije.

Uobičajeni metodi nedostatak ili sporost komunikacije rešavaju formalnim i detaljnim dokumentima: specifikacijom zahteva, modelom proizvoda, itd. Kad neki programer ima neko pitanje, samo prelista dokumente. Međutim, ovo ima i svoje nedostatke. Pre svega, nemoguće je unapred proceniti šta sve treba da sadrži finalni proizvod. Uvek će neko naći nešto što fali u dokumentima. S druge strane, ovi dokumenti su uglavnom puni dvosmislenih, nepreciznih ili čak kontradiktornih detalja. Programer u ovom slučaju ima dve opcije: ili će poslati poruku klijentu da razjasni problem (i čeka na odgovor) ili će ići svojim putem i implementirati taj detalj na onaj način za koji smatra da je najbolji.

XP smatra da je najbolje, kad svi sede u istoj prostoriji bez obzira na njihove uloge. U tom slučaju, kad programer ima neko pitanje, samo pita jednog *on-site* klijenta. Sedenje zajedno efikasno eliminiše vreme čekanja na odgovor i poboljšava produktivnost (po studiji *Teasley*-a, sedenje zajedno duplira produktivnost). Zbog poboljšane komunikacije formalni dokumenti gube svoj smisao.

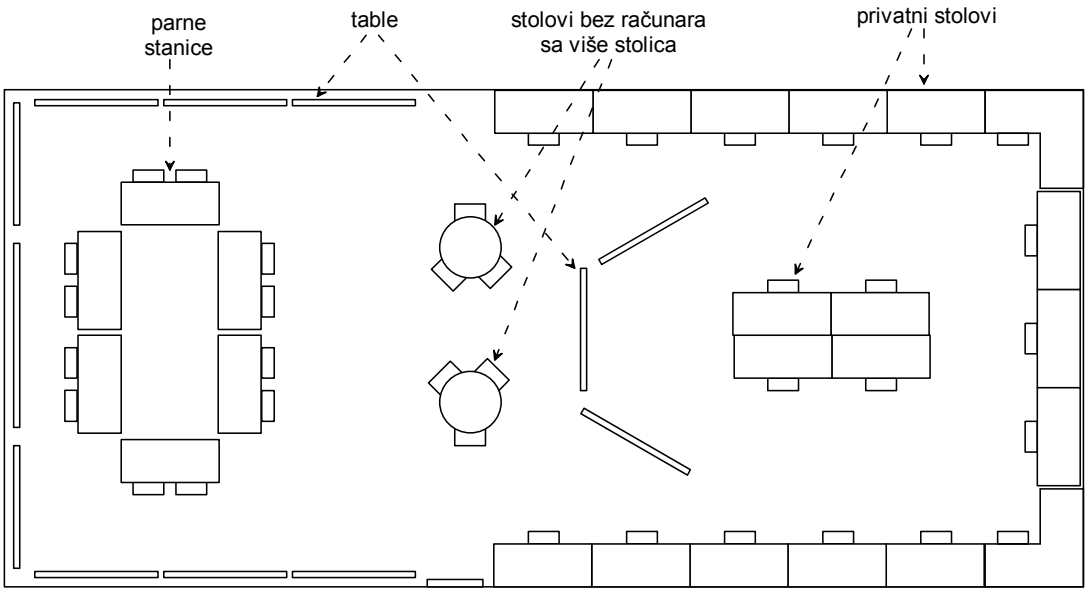
Članovi tima moraju da sede dovoljno blizu da bi kratke konverzacije između njih bile adekvatne bez ustajanja ili vikanja. Razgovor između članova će se čuti kroz celu prostoriju stvarajući stalnu vibraciju i šum. Postavlja se pitanje da li ovo predstavlja smetnju za ostale članove ili ne. Odgovor je potvrdan. Razne smetnje mogu uzrokovati prekide rada koji su dosta štetni. *XP*, međutim, dozvoljava ovaj nivo smetnje jer ima jedan princip koji ih efikasno neutrališe: programiranje u parovima.

Nagovoriti članove tima da sede zajedno može biti težak posao ali nije jedini problem. Sedenje zajedno nema puno smisla ukoliko radno okruženje ne odgovara. Uobičajene prostorije za razvoj uglavnom imaju male radne „kutije“ koje su odvojene tankim zidovima. Često su i prostorije relativno male. *XP*-tim zahteva veliku prostoriju bez „kutija“. Izgradnja takve prostorije će u većini slučajeva zahtevati rušenje zidova. Uprava firme često ne razume zašto je ovo potrebno (a s druge strane, dosta je skupo).

Imati odgovarajuću prostoriju je jedno – opremiti takvu prostoriju je nešto sasvim drugo. Jedna solidna prostorija bi izgledala na sledeći način (slika 6):

- Nekoliko parnih stanica da bi programeri mogli da praktikuju programiranje u parovima. Bilo bi dobro imati malo više parnih stanica od broja parova-programera, da bi programeri mogli da budu u paru i sa test-osobama ili sa klijentima. Ove dodatne parne stanice su odlične i u slučaju da neko od programera mora da radi sam.
- Test-osobe bi sedele malo dalje od programera, ali i dalje dosta blizu da bi programeri mogli da čuju njihovu konverzaciju

- Domenski eksperti i dizajneri interakcije mogu sesti još dalje od programera, ali dovoljno blizu da bi mogli da odgovore na kratka pitanja bez vikanja.
- Menadžer proizvoda i menadžer projekta bi trebali da sede zajedno dovoljno blizu drugim članovima tima da i oni budu deo vibracije, ali i dovoljno daleko da njihov razgovor ne odvraća pažnju drugih.
- Mnogi kažu da zajedničke prostorije eliminišu privatnost pojedinaca. Da bi se rešio ovaj problem, treba obezbediti mali privatni prostor za svakog člana tima. Svaki član tima bi trebao da ima jedan kompjuter ili laptop za privatne svrhe.
- Pored velike zajedničke prostorije treba imati i jednu drugu odvojenu prostoriju u kojoj bi članovi tima mogli da obavljaju privatne razgovore telefonom ili licem-u-lice.
- Zidovi bi trebali da budu prekriveni tablama. Ovo je potrebno da bi mogao da funkcioniše princip informativnog radnog okruženja.
- Ako je moguće, treba nabaviti i jedan projektor. Ovo omogućava predstavljanje kompjuterizovanog materijala celom timu bez prelaska u konferencijske sale.



Slika 6: Jedna solidna prostorija za tim od 13 članova (6 programera).

2.3.3. Uključivanje stvarnih klijenata

Uključivanje stvarnih klijenata (eng. *Real Customer Involvement*) pomaže razvojnom timu da razume šta se razvija. Naime, softverski proizvod se razvija za klijenta. Zato je *XP* otišao korak dalje i rešio da klijente i programere dovede u istu prostoriju: ti klijenti su *on-site* klijenti. Oni su odgovorni za pravljenje zahteva zajedno sa određivanjem njihovih prioriteta. Prioritet zahteva se određuje u zavisnosti od tržišne vrednosti tog zahteva.

Poznato je da nisu svi projekti namenjeni istim korisnicima. U zavisnosti od toga, projekti se dele u sledećih pet grupa:

1. **Lični razvoj** (eng. *Personal Development*) – razvojni tim razvija softver za sopstvene svrhe. U ovom slučaju klijent je sam razvojni tim i zbog toga nema potrebe da se angažuju neki eksterni klijenti.
2. **Unutrašnji proizvoljni razvoj** (eng. *In-House Custom Development*) – dešava se kad kompanija zamoli neki svoj razvojni tim da razvije softver za potrebe te kompanije. U ovom slučaju inicijator projekta je sama kompanija (preciznije rečeno jedna osoba). Iako se čini da je ovo težak posao, barem je lak u tom smislu da se lako dobavljaju *on-site* klijenti. Inicijator projekta bi bio menadžer proizvoda, dok neki zaposleni u firmi bi mogli da budu domenski eksperti.
3. **Eksterni proizvoljni razvoj** (eng. *Outsourced Custom Development*) – jako liči na unutrašnji proizvoljni razvoj, s tim da je inicijator projekta neka druga firma. Sve što važi kod unutrašnjeg proizvoljnog razvoja, važi i ovde. Jedini problem može biti mnogo teže dobavljanje *on-site* klijenata jer su ti klijenti radnici te druge firme, pa možda ne mogu da sede zajedno sa timom. U tom slučaju treba ih pitati da li bi im odgovaralo rešenje da se razvojni tim preseli kod njih.
4. **Vertikalno-tržišni softver** (eng. *Vertical-Market Software*) – ovaj tip projekta se razlikuje od prethodnih tipova jer uključuje više kompanija. Novi problem kod ovog tipa razvoja je to da isporučen softverski proizvod treba da zadovolji potrebe svih klijenata. U ovom slučaju *on-site* klijenti bi trebali da budu pravi korisnici a ne zainteresovane strane. Ovaj pristup omogućava ravnomerno zadovoljenje svih zainteresovanih strana.
5. **Horizontalno-tržišni softver** (eng. *Horizontal-Market Software*) – ovaj tip projekta ima još širi krug klijenata u odnosu na vertikalno-tržišni softver jer pored pravna lica uključuje i privatna lica. Zbog toga se finalni proizvod može kupiti u prodavnicama, preko interneta, itd.

2.3.4. Zajednički jezik

Zajednički jezik (eng. *Ubiquitous Language*) pomaže članovima tima da se međusobno razumeju. Treba imati u vidu da su članovi jednog *XP*-tima dosta različiti. Programeri u većini slučajeva nemaju nikakvo znanje o domenu softvera koji se razvija. S druge strane, domenski eksperti verovatno ne znaju da programiraju. Zato je važno da se pronađe neki zajednički jezik.

Smatra se da je lakše da programeri govore jezikom domena nego obrnuto. Kad jedan programer ima pitanje za klijenta, to pitanje mora biti formulisano tako da ne sadrži programerske fraze i izraze: klase, metode, objekte, rekurzije, stabla, atribute, parametre, itd.

Ponekad je dosta teško „prevesti“ jezik domena nazad na jezik implementacije. U tom slučaju preporučuje se korišćenje jezika domena i u programu. Ovo podrazumeva uvođenje domenskih koncepata i elemenata u dizajn samog softvera. Proces kreiranja softvera „u duhu“ domena se naziva **dizajn usredsređen ka domenu** (eng. *Domain-Driven Design*).

2.3.5. Stojeći sastanci

Stojeći sastanci (eng. *Stand-Up Meetings*) omogućavaju da članovi tima budu dobro informisani. Uobičajeni metodi koriste nedeljne *statusne sastanke* na kojima vođa sastanka pročita zadatke, a ostali prisutni razgovaraju o tome: kako napreduju sa zadacima, da li su naišli na neki problem, itd. Međutim, mana ovih sastanaka je to da

traju dosta dugo i prekidu normalan rad. *XP* je statusne sastanke zamenio informativnim radnim okruženjem i dnevnim stojećim sastancima.

Stojeći sastanci su neformalni: članovi tima stoje u krugu i svi redom kažu nekoliko rečenica o napretku, problemima, uspesima, šta će raditi sledeće, itd. Kod iskusnih *XP*-timova jedna osoba govori otprilike pola minuta, dok celi sastanak ne traje više od deset minuta zavisno od veličine tima. Ključni pojam kod ovih sastanaka je *sažetost*: prisutnima treba dati samo grubu ideju o statusu projekta. Zato se i ovi sastanci zovu „*stojeći*“ sastanci: ljudi ne vole dugo da stoje na jednom mestu i to će ih indirektno navesti da pričaju manje.

2.3.6. Standardi kodiranja

Standardi kodiranja (eng. *Coding Standards*) daju jedan šablon koji omogućava „bešavno“ spajanje pojedinačnih radova programera u celinu. Naime, svi programeri imaju svoj stil pisanja programskog koda koji su razvijali godinama. Kod timskog rada, međutim, ovo može da bude problem. Svađati se o tome čiji je stil najbolji – nema smisla. Zato je važno razviti standarde kodiranja.

Standardi kodiranja predstavljaju skup vodilja za koji su svi programeri dali svoju saglasnost. Ovo ne podrazumeva samo *stil kodiranja*, nego i *načine kodiranja* kao što su:

- Korišćeni alati i razvojna okruženja
- Način čuvanja datoteka
- Način rukovanja sa greškama i izuzecima
- Konvencije u dizajnu (na primer: način rukovanja sa *null*-vrednostima, veličina metoda, imenovanje metoda, klasa i polja, itd.)

Treba napomenuti da stil kodiranja nije tako važan kao način programiranja. Možda će programski kod izgledati ružno, ali različiti pristupi o načinu kodiranja mogu biti izvor budućih grešaka.

O standardima kodiranja najbolje je govoriti već tokom prve iteracije u vidu nekoliko jednočasovnih sastanaka sa svim programerima. Ako za izabran programski jezik već postoji zvanični standard, i ako su svi prisutni saglasni s tim, treba ga prihvatiti. Ako korišćen jezik kodiranja nema zvanični standard, nije dovoljno precizan, ili nije u skladu sa potrebama trenutnog softvera koji se razvija, programeri tima mogu ili dopuniti postojeći standard ili napraviti jedan skroz novi.

2.3.7. Demonstracija iteracije

Demonstracija iteracije (eng. *Iteration Demo*) prezentuje trud tima da se razvije softver u skladu sa ciljevima zainteresovanih strana. Ona je jedna prezentacija na kojoj će biti prikazan delimično završen softver. Ona se organizuje na kraju svake iteracije (znači, ukoliko iteracija traje svega nedelju dana, onda se demonstracija iteracije održava svake nedelje). Pravljenje demonstracije iteracije može biti zgodno zbog sledećih razloga:

- Predstavlja konkretnu demonstraciju o napretku projekta.
- Omogućava timu da bude iskren o napretku. Naime, demonstracije iteracije su otvorene za sve zainteresovane što otežava odlaganje održanja demonstracije radi ispravljanja grešaka.
- Predstavlja odličan način sakupljanja povratne sprege od prisutnih.

Demonstraciju iteracije bi trebao da vodi menadžer proizvoda, pošto najbolje poznaje zainteresovane strane. Na prezentaciju su svi dobrodošli: sponzori, druge zainteresovane strane, uprava firme, pravi korisnici, drugi razvojni timovi, itd. Sama prezentacija ne bi trebala da traje dugo jer se održava relativno često (na kraju svake iteracije). Optimalna dužina demonstracije je deset minuta (ne duže od pola sata).

Demonstracija iteracije počinje kratkim opisom zahteva koji su bili planirani za tekuću iteraciju. Ako je tokom iteracije došlo do promene plana, treba objasniti šta se desilo. Ne treba ulepšavati situaciju: najbolje je biti direktan. Ovo će prisutnima biti jasan signal da je tim preuzeo odgovornost za projekat i da je dovoljno profesionalan da reši probleme. Posle ovog kratkog uvoda, sledi demonstracija svakog zahteva posebno. Ukoliko je došlo do promena u nekim zahtevima ili storijama, treba to objasniti. Kao što je već rečeno, ne treba ništa ulepšavati ili prećutati: prisutni bi to odmah primetili. Ovo važi i u slučaju kad tim nije u stanju da pokaže bilo šta novo u odnosu na prethodnu demonstraciju.

Joshua Kerievsky preporučuje da na kraju demonstracije treba naći glavnog sponzora (naručioca, inicijatora) softvera i postaviti mu dva važna pitanja:

1. „*Da li ste zadovoljni sa dosadašnjim radom?*“
2. „*Da li možemo da nastavimo?*“

Ukoliko postoji adekvatna komunikacija između naručioca i razvojnog tima, odgovori ne bi trebali da predstavljaju nikakvo iznenađenje. Ukoliko je odgovor na prvo pitanje negativan, to je siguran znak da nešto nije u redu. Treba odmah razjasniti uzroke nezadovoljstva naručioca (u većini slučajeva to je prespor napredak) i uvrstiti njegove predloge u sledeću iteraciju. Međutim, negativan odgovor na drugo pitanje je ujedno i kraj projekta.

2.3.8. Izveštavanje

Izveštavanje (eng. *Reporting*) pomaže razvojnom timu da dokaže upravi kompanije da tim radi na odgovarajući način. Postoje više izveštaja koji se mogu koristiti. Oni se dele na izveštaje o napretku i na izveštaje o upravljanju. **Izveštaji o napretku** (eng. *Progress Reports*) ne služe za nadgledanje tima i za njegovo usmeravanje, već da zainteresovane strane imaju poverenje da će tim biti sposoban da donosi najbolje moguće odluke. **Izveštaji o upravljanju** (eng. *Management Reports*) su namenjeni upravi firme i sadrže precizne informacije koje omogućavaju da se analiziraju trendovi i da se odrede ciljevi (tj. da se dokaže da tim radi na odgovarajući način). Koliko dokumenata napraviti za zainteresovane strane i za upravu zavisi isključivo od njih, ali ne treba praviti više dokumenata od onoga što zahtevaju jer bi to značilo trošenje vremena.

Izveštaji o napretku se kod *XP*-a prave lako, šta više, veliki broj od tih dokumenata su u suštini nusprodukti *XP*-a (pa ne treba trošiti dodatno vreme za njihovo pravljenje). Preporučuje se korišćenje sledećih dokumenata:

- **Deklaracija vizije** (eng. *Vision Statement*) – opisuje šta se radi unutar projekta, zašto se radi, itd. Za ovaj dokument su odgovorni *on-site* klijenti. Više o deklaraciji vizije u odeljku 2.5.1.
- **Demonstracija iteracije** – iako nije pisan dokument, javna prezentacija o napretku razvoja je možda najbolji način da se dokaže zainteresovanim stranama profesionalizam razvojnog tima.

- **Planovi izdanja i iteracije** – ovi dijagrami odlično prikazuju napredak razvoja softvera.
- **Burn-up dijagram** (eng. *Burn-Up Chart*) – predstavlja grafički prikaz napretka zajedno sa budućim očekivanjima. Više o *burn-up* dijagramu u odeljku 2.5.4.

Od izveštaja o upravljanju sledeći dokumenti mogu biti od važnosti:

- **Produktivnost** – produktivnost se veoma teško meri (*Fowler*), ali se zato može meriti uticaj tima kroz iteracije na tržišnu vrednost (na primer: povratak uloženih sredstava).
- **Propustljivost** (eng. *Throughput*) – predstavlja ukupan broj implementiranih funkcija od strane razvojnog tima.
- **Defekti** (eng. *Defects*) – služi za brojanje defekata i grešaka u izvornom kodu.
- **Upotreba vremena** (eng. *Time Usage*) – uprava firme želi da zna da li njeni radnici pošteno koriste svoje radno vreme. Zato se preporučuje da svi članovi vode računa o sebi. Da bi ovo bilo lakše, upotreba vremena se može napisati i na drugu stranu stori-listića.

Zbog manjka pružanja adekvatnih informacija preporučuje se izbegavanje sledećih dokumenata:

- **Linije izvornog koda** (eng. *Source lines of code*, *SLOC*) i **funkcijski poeni** (eng. *Function Points*) – ove tehnike se koriste za utvrđivanje veličine softvera. Nažalost, neki ih koriste i za utvrđivanje produktivnosti. Ne treba zaboraviti da veličina koda nije povezana sa vrednostima ili funkcionalnošću. Forsiranje ovih tehnika za utvrđivanje produktivnosti će kao rezultat verovatno imati smanjenu produktivnost. Naime, ovo će navesti programere da se više bave pisanjem dugačkih kodova, a da se manje bave kvalitetnim dizajnom.
- **Broj storija** – nažalost, neki koriste ukupan broj storija kao meru produktivnost, što je pogrešno.

2.4. Grupa principa broj 3: Izdavanje

Već je nekoliko puta bilo spomenuto da razvoj softvera nije predvidljiv proces. Ljudi često ne znaju kako da se odnose prema softveru. Zbog toga se često dešava da kad razvojni tim završi program, potrebno je još nekoliko nedelja ili čak meseci da bi softver bio spreman za isporuku. Kod *XP*-a je važno da softver bude spreman za isporuku u bilo kom momentu. Da bi ovo postigao, *XP* koristi sedam principa o kojima će biti izlagano u nastavku: biti „gotov-gotov“, bez grešaka, kontrola verzija, deseto-minutna izgradnja, stalna integracija, kolektivno vlasništvo koda i dokumentacija.

2.4.1. Biti „gotov-gotov“

Biti „gotov-gotov“ (eng. *„Done Done“*) garantuje da je ono što je završeno, spremno i za isporuku. Znači, kad je neki zahtev, tj. storija „gotova-gotova“ to ne znači samo da je završena, nego i da je iztestirana i integrisana u sistem. Tačna definicija, šta znači biti „gotov-gotov“ zavisi od kompanije, ali u većini slučajeva to znači da je storija:

- **Iztestirana** – ovo podrazumeva test modula, test integracije i klijentske testove
- **Izkodirana** – programski kod je kompletno napisan
- **Dizajnirana** – kod je refaktorisan i u skladu je sa arhitekturom sistema
- **Integrisana** – storija radi u celosti i dobro se uklapa u celinu
- **Može da se kompajlira i instalira** – storija je uključena u instalaciju
- **Pregledana** – klijenti su pregledali storiju i zaključili da je u skladu sa njihovim očekivanjima
- **Ispeglana** – sve eventualne greške su ispravljene
- **Prihvaćena** – klijenti su složni da je storija završena. Storija ne može biti „gotova-gotova“ sve dok klijenti ne ocene da je to tako.
- **(Dokumentovana)** – storija je dokumentovana i uključena u *help*-fajl. Ovo je opciono jer se može raditi i kasnije od strane drugih članova tima.

Početni programeri u *XP*-timovima smatraju da nije lako biti „gotov-gotov“. Razvoj usredsređen ka testiranju (*TDD*), stalna integracija i deseto-minutna izgradnja mogu biti od pomoći. Važno je uključiti i *on-site* klijente i test-osobe u razvoj.

2.4.2. Bez grešaka

Princip pod nazivom **bez grešaka** (eng. *No Bugs*) omogućava isporuku softvera bez posebne *faze testiranja*. Većina ljudi bi smatrala da je tako nešto nemoguće. Međutim, uz korišćenje agilnih principa zajedno sa disciplinovanim radom je već – sasvim realno.

Poboljšanje kvaliteta softvera kod uobičajenih metoda označava korišćenje faze testiranja za efikasno *pronalaženje grešaka* u programskom kodu. Kod agilnih metoda glavni cilj je *generisati što manje grešaka*. Da bi se to postiglo, preporučuje se korišćenje sledećih pet preporuka:

1. **Pisati što manje grešaka** – za ovo je potreban stimulisan rad uz *TDD* i programiranje u parovima. Sve ovo će smanjiti verovatnoću pojavljivanja grešaka u samom kodu, ali ne i onih grešaka zbog kojih će program raditi, ali ne na

odgovarajući način. Za ispravljanje ovih tipova defekata potrebno je aktivno angažovanje *on-site* klijenata i test-osoba.

2. **Eliminisati izvore budućih grešaka** – dizajn koji izgleda elegantno na početku možda neće odgovarati kasnije, postavši potencijalan izvor budućih grešaka. Ovim se povećava i tehnički dug. Za smanjenje tehničkog duga preporučuje se često refaktorisanje i korišćenje jednostavnog dizajna.
3. **Odmah ispraviti greške** – po *McConnell*-u, što se više čeka da se ispravi neka greška, to će ispravka biti skuplja. Ispravka jedne greške počinje tako što se napiše jedan test koji je jasno demonstrira. Zatim treba ispraviti grešku i proveriti da li će i test proći. Međutim, ovde još nije kraj. Naime, treba naći i uzrok te greške. U većini slučajeva to je loš dizajn. Ako je to slučaj, treba ga ispraviti ili bar poboljšati. Ponekad će se desiti da je greška previše komplikovana. U tom slučaju preporučuje se kreiranje jedne potpuno nove storije koja će se rešiti kroz trenutne ili sledeće iteracije.
4. **Testirati proces** – prethodne preporuke su se odnosile na već postojeće greške. Ova preporuka služi da se predvide greške pre nego što su se one pojavile. Ovo se vrši *istraživačkim testiranjem* od strane test-osoba. Istraživačko testiranje omogućava test-osobi da identifikuje one probleme na koje programeri i klijenti ne bi ni pomislili (na primer, upad u server). Ukoliko se nađu takvi problemi, to je znak da postoji problem u samom procesu razvoja. U tom slučaju treba da se vrši diskusija sa celim timom radi poboljšanja procesa.
5. **Ispraviti proces** – kad je nivo grešaka u timu dovoljno nizak, može da se uvede i princip iskonske analize. Ova napredna tehnika je najbolja za pronalaženje dubokih uzroka problema. Ispravljanje ovih problema će pojavljivanje sličnih grešaka učiniti nemogućim.

Kao što se može videti, većina ovih preporuka je već uključena u ostale principe *XP*-a, pa ni njihovo korišćenje neće predstavljati teškoću. Međutim, važi i obrnuto: bez ozbiljne primene ostalih principa *XP*-a neće ni ovaj princip biti uspešan.

2.4.3. Kontrola verzija

Kontrola verzija (eng. *Version Control*) omogućava programerima tima da ne ometaju rad drugih programera. Naime, kod razvojnih timova više programera radi na istom izvornom kodu, pa čak i na istom parčetu koda. Ovo bi moglo brzo da izmakne kontroli: na svakom računaru bi bio deo koda koji ne bi postojao na drugim računarima, i zato bi sklapanje koda u celinu bio težak posao. Umesto toga, trebalo bi da postoji jedan centralni sistem koji bi bio odgovoran za upravljanje izvornim kodom: **sistem kontrole verzija** (eng. *Version Control System*).

Kod kontrole verzija postoji poseban rečnik pojmova. Najvažniji izrazi su sledeći:

- **Repozitorijum**, **skladište** (eng. *Repository*) –glavno skladište fajlova. Smešta se na server kontrole verzija. Sadrži i *istoriju* fajlova, tj. njihove starije verzije.
- **Radna kopija** (eng. *Working Copy*, *Sandbox*) – svaki programer bi trebao da ima svoju kopiju repozitorijuma na računaru na kojem radi (zato se i naziva radna kopija).
- **Ažuriranje** (eng. *Update*) – ažuriranje radne kopije iz repozitorijuma na najsvežiju verziju ili na neku stariju verziju.

- **Zaključavanje** (eng. *Lock*) – zaključavanje fajlova svima osim vlasniku radne kopije.
- **Odjava** (eng. *Check Out*) – predstavlja kreiranje sveže radne kopije iz repozitorijuma. Sastoji se iz dve akcije: ažuriranja i zaključavanja.
- **Prijava** (eng. *Check In*, *Commit*) – kopiranje fajlova iz radne kopije u repozitorijum.
- **Spajanje** (eng. *Merge*) – proces kombinovanja fajlova i razrešenja konflikata. Ukoliko dva programera rade na istom parčetu koda, i oba programera prijave svoje izmene u repozitorijum, drugi programer će morati prvo da ubaci izmene prvog programera u svoj kod.

Postoje dva modela kontrole verzije. Prvi model je **zaključni model** (eng. *Lock Model*) koji podrazumeva da čim neko počne da radi na nekom fajlu, sistem zaključava taj fajl za ostale programere. Ovo je loše rešenje jer onemogućava rad na tom fajlu drugim programerima. Umesto toga preporučuje se korišćenje **konkurentnog modela** (eng. *Concurrent Model*). Ovo omogućava paralelni rad programera bez obzira na kom fajlu trenutno rade. Čak iako dva programera rade na istoj liniji koda, sistem će ili zamoliti drugog programera da ručno ubaci izmene prvog programera pre prijave, ili će to uraditi automatski.

Jedna zgodna mogućnost kontrole verzije je to da se radna kopija može ažurirati i na neku *stariju* verziju. Ovo omogućava efikasnije ispravljanje pronađenih grešaka. Naime, u slučaju greške dovoljno je stalnim ažuriranjem na starije verzije koda pronaći onu prijavu u kojoj se prvi put pojavljuje ta greška. Poređenjem ove verzije koda sa prvom prethodnom verzijom (u kojoj se još nije pojavila greška) se lako može pronaći onaj deo koda koji je proizveo grešku.

Preporučuje se da se pored izvornog koda skladište i skoro svi ostali elementi projekta: korišćeni alati, biblioteke, dokumentacija, testovi, klijentski podaci (beleške o zahtevima, storijama), itd. Razlog tome leži u činjenici da su i ovi elementi promenljivi, tj. ažuriraju se ili se zamenjuju sa novijim verzijama. Međutim, izvršna verzija softvera se ne skladišti.

Nije potrebno napraviti sopstvenu kontrolu verzija jer postoje i gotova softverska rešenja. Od komercijalnih alata preporučuje se *Perforce*, dok je od slobodnih softvera najpoznatiji *Subversion* sa *frontend*-om *TortoiseSvn*.

2.4.4. Deseto-minutna izgradnja

Deseto-minutna izgradnja (eng. *Ten-Minute Build*) omogućava sklapanje projekta u iztestiranu izvršnu verziju za manje od deset minuta. Pošto se skoro celi projekat drži na serveru kontrole verzija (repozitorijum), projekat može da se sklapa u izvršnu verziju u bilo kom trenutku. Ovo ne podrazumeva samo kompajliranje i izvršavanje sekvenci testova, nego i nameštanje radnog okruženja, kao što su: konfigurisanje *Web*-servera, inicijalizacija lokalne baze podataka, nameštanje *Registry*-ja, pokretanje procesa, itd. Zbog toga je deseto-minutna izgradnja zgodna ne samo za demonstracije iteracije nego i za prebacivanje celog projekta na novi računar razvojnog tima.

Sve ovo se postiže automatizacijom celog postupka. Postoje razni alati specijalizovani za ovo. Kod *Jave* najpoznatiji alat je *Ant*, kod *.NET*-a *MSBuild* ili *NAnt*, itd. **Skripta izgradnje** (eng. *Build Script*) mora da bude dovoljno samostalna da bi mogla da radi i u *Offline* režimu.

Pošto je ovaj princip toliko važan, preporučuje se rana automatizacija (već u prvoj iteraciji). Na početku, skripta izgradnje će biti dosta jednostavna zbog svežine samog projekta. Najbolje je da skripta izgradnje bude u stanju da uradi samo to što je potrebno u datom momentu, ne više od toga. Napretkom razvoja softvera će i skripta rasti.

Najveći problem kod deseto-minutne izgradnje je to da se dosta često izvršava (zbog stalne integracije svakih nekoliko sata). Zato je važno da vreme izgradnje traje manje od deset minuta (najbolje je otprilike pet minuta). U većini slučajeva razlog sporosti izgradnje su veliki broj testova ili previše komplikovani testovi. Drugi najčešći razlog sporosti je sporo kompajliranje (tada optimizacija koda može pomoći).

2.4.5. Stalna integracija

Stalna integracija (eng. *Continuous Integration*) omogućava razvojnom timu da preskoči dugačku i rizičnu fazu integracije. Naime, kod većine projekata postoji ozbiljna vremenska razlika između trenutka završetka rada i trenutka isporuke softvera: treba još spajati kod svih programera u celinu, napraviti instalaciju, dokumentaciju, i slično. Sve ovo može da potraje nedeljama ili čak mesecima. Uz stalnu integraciju razvojni tim je u stanju da bilo kad isporuči softver.

Tajna iza stalne integracije je u tome da se integracija radi svakih nekoliko sati. Integracija se sastoji iz tri koraka:

1. Osvežiti radnu kopiju na najsvežiju verziju iz repozitorijuma – možda je u međuvremenu neki drugi programer osvežio repozitorijum
2. Kompajlirati ovu novu radnu kopiju zajedno sa novim programskim kodom i proveriti da li radi kako treba
3. Prijaviti ovu radnu kopiju nazad na repozitorijum

Stalnom integracijom se lakše ispravljaju greške. Pošto je vremenska razlika između dve uzastopne integracije relativno mala, dovoljno je samo pogledati koji deo koda je izmenjen. Ali da bi ovo proradilo, potrebno je imati poverenje u repozitorijum. Zato zlatno pravilo stalne integracije glasi: *repozitorijum se uvek mora kompajlirati*.

Da bi prethodno spomenuto zlatno pravilo bilo primenljivo, moraju se rešiti dva problema: da ono što radi na lokalnom računaru, radi na bilo kom računaru i da niko ne dobija programski kod za koji još nije dokazano da radi. Za uspešno rešavanje ovih problema potrebno je imati jedan dodatni centralni računar za integraciju (ovo je ustvari repozitorijum). Kad se prijavi novi kod u repozitorijum, treba otići do tog računara i tamo takođe kompajlirati kod. Međutim, pošto više programera radi na projektu istovremeno, mora se rešiti problem sinhronizacije. Plišane igračke predstavljaju odlično rešenje. Znači, kad neki programer želi da integriše programski kod prijavom tog koda u repozitorijum, uzima plišanu igračku i drži je kod sebe sve dok ne kompajlira prijavljen kod na centralnom računaru, i sve dok se ne uveri da je kompajliranje uspešno završeno. Ukoliko neki drugi programer hoće da integriše novi kod i vidi da plišana igračka nije na mestu, to znači da neko već integriše i moraće da čeka. Iz ovoga se već jasno vidi, zašto je toliko važno da vreme izgradnje projekta traje manje od deset minuta. Inače, ovaj način integracije se naziva **sinhronizovana integracija** (eng. *Synchronous Integration*).

2.4.6. Kolektivno vlasništvo koda

Kolektivno vlasništvo koda (eng. *Collective Code Ownership*) omogućava timu rešavanje problema bez obzira gde se ti problemi pojavljuju. Već je nekoliko puta

spominjano da u *XP*-timu ne postoje pojedinci nego celi tim. Celi tim je odgovoran za projekat. Ako neko napravi grešku, niko ne traži krivca. Ovo se može postići samo kolektivnim vlasništvom koda koje podrazumeva da je vlasnik programskog koda sam tim a ne pojedinci. Takođe, svi članovi tima mogu da pogledaju izvorni kod celog projekta i ako neko primeti neku grešku, može da je ispravi bez ikakvih obaveštenja i restrikcija. Ovo je potpuno obrnuta filozofija u odnosu na *lično* vlasništvo koda koje podrazumeva da je vlasnik nekog dela koda pojedinac koji je i odgovoran za to parče koda i verovatno niko drugo ne poznaje njegov rad.

Prihvatanje principa kolektivnog vlasništva koda može da bude problem kod onih ljudi koji su ponosni na svoj rad i teško podnose kad vide da neko menja njihovu kreaciju. Njima treba objasniti da i oni mogu slobodno da menjaju tuđe radove. Kolektivno vlasništvo koda ima još jednu veliku prednost: u slučaju da neki programer mora biti odsutan neko vreme, ili zauvek ode iz firme, ostatak tima će moći bez problema da nastavi sa radom. Iz svega ovoga se može zaključiti da je kolektivno vlasništvo koda manje rizično od ličnog vlasništva.

2.4.7. Dokumentacija

Kasnija **dokumentacija** (eng. *Documentation*) smanjuje troškove dokumentacije i povećava njenu preciznost. Dokumentacija je jedan vid komunikacije. Uobičajeni metodi najviše vrednuju pisani vid komunikacije, dok *XP* smatra da je verbalna komunikacija licem-u-lice bolje rešenje. Postoje tri tipa dokumentacije:

- **Radna dokumentacija** (eng. *Work-In-Progress Documentation*) – sadrži one informacije koje su od suštinskog značaja za sam projekat. Ovi dokumenti objašnjavaju šta treba da sadrži proizvod, kako treba da izgleda, kako treba da funkcioniše, i slično. Kod uobičajenih metoda specifikacija zahteva, model proizvoda i slični dokumenti su radni dokumenti. Kao što je već poznato, *XP* je zamenio ove dokumente sa verbalnom dokumentacijom.
- **Dokumentacija proizvoda** (eng. *Product Documentation*) pruža poslovnu vrednost za proizvod. U ovu kategoriju spadaju priručnici za korisnike i izveštaji. Pošto su ovi dokumenti deo proizvoda, treba ih tako i tretirati: posebnom storijom.
- **Predajna dokumentacija** (eng. *Handoff Documentation*) se koristi kad se razvojni tim sprema da preda projekat nekom drugom timu. Ovo se dešava najčešće nakon izdanja finalnog proizvoda, kad održavanje tog softvera preuzima neki drugi tim. Ova dokumentacija može da sadrži kratki opis projekta, teškoće, izazove, kompromise, i slično.

2.5. Grupa principa broj 4: Planiranje

Smatra se da što je veći projekat, to je teže upravljati njime. Jedan od razloga je to da razvoj softvera nije predvidljiv proces. Uobičajeni metodi se boje promena zato što su one teško predvidljive. Umesto da predvidi promene, *XP* se prilagođava da bude „u skladu“ sa njima. Naravno, ovo može lako da izmakne kontroli, ali zato postoji osam principa koji mogu biti od pomoći: vizija, planiranje izdanja, igra planiranja, upravljanje rizikom, planiranje iteracije, lenčarenje, storije i procenjivanje.

2.5.1. Vizija

Vizija (eng. *Vision*) otkriva u kom pravcu ide projekat i zašto se kreće u tom pravcu. Svaki projekat počinje sa jednom *idejom* (bez obzira da li je reč o softveru ili o nečem drugom). Ova ideja može biti bilo šta. Ukoliko se pronađe odgovarajuća tržišna vrednost iza te ideje, neko će početi da je finansira. Iz ideje nastaje vizija, a od vizije – projekat.

Međutim, vizija se često izgubi u tranziciji, što nije ni čudo, pošto je projekat jedan kompleksan poduhvat. Nažalost, gubitak vizije ili iskrivljena vizija je jedan od razloga zbog kojih finalni proizvod ne opstaje na tržištu. Zato je kontakt sa **vizionarima** (nosačima vizije, eng. *Visionaries*) od presudnog značaja. Kod *XP*-a ova uloga je dodeljena menadžeru proizvoda koji najbolje razume njihov način razmišljanja i ume da bude medijator između njih. Ukoliko postoji samo jedan vizionar, najbolje je njega proglasiti za menadžera proizvoda. Da se ne bi izgubila vizija projekta, kreira se, zajedno sa vizionarima, **deklaracija vizije** (eng. *Vision Statement*). Ovaj dokument je relativno kratak (jedna strana), i sastoji se iz tri dela:

- **Šta bi trebalo da se ostvaruje?** – opisuje problem koji će biti rešen rezultatom projekta (tj. proizvodom).
- **Zašto je projekat vredan?** – opisuje gde će se osetiti poboljšanje u odnosu na sadašnju situaciju.
- **Koji su kriterijumi uspeha projekta?** – objašnjava kako će se primetiti da je projekat uspeo.

Nakon kreiranja deklaracije vizije, ona treba da bude stalno promovisana. Ovo se postiže tako što će biti deo informativnog radnog okruženja. Takođe, jako je važno da vizionari budu blizu tima (a po mogućstvu u timu): treba ih pozivati na demonstracije iteracije, angažovati ih da učestvuju na kreiranju planiranja izdanja, itd.

2.5.2. Planiranje izdanja

Planiranje izdanja (eng. *Release Planning*) pruža razvojnom timu putanju do cilja. Međutim, kod *XP*-a i agilnih metoda, planiranje izgleda donekle drugačije u odnosu na uobičajene metode. U nastavku će biti izlagane neke preporuke sa jednostavnim primerima.

Pretpostavka je sledeća: jedan razvojni tim dobija zadatak da razvije dva softverska proizvoda u vidu dva odvojena projekta. Vreme trajanja svakog projekta je tri meseca. Tim ima nekoliko mogućnosti. Jedna od njih je paralelni razvoj oba proizvoda (50% radnog vremena na jednom projektu, 50% na drugom). Međutim, ovo rešenje se nikako ne preporučuje, jer će samo zbuniti celi tim (po *DeMarco*-u, gubitak produktivnosti je najmanje 15%). Drugo rešenje može biti 100% posvećivanje nekom

projektu, ali da se svakog meseca menja projekat. Treće rešenje je isto 100% posvećivanje nekom projektu, ali tako što se ne prelazi na drugi projekat sve dok prvi projekat nije završen. Ako se uzme u obzir činjenica da većina kompanija želi da se što pre vidi povratak uloženih finansijskih sredstava, treće rešenje je najlogičniji izbor. Naime, kod trećeg rešenja je jedan projekat gotov za tri meseca i počinje da vraća uložena sredstva, dok će kod drugog rešenja morati da prođe pet meseci da bi se završio jedan projekat. Kao što se vidi (tabela 1), oba scenarija zahtevaju šest meseci za završetak oba projekta, ali postoji velika razlika između njih u vidu povratka sredstava.

	Scenario 2		Scenario 3	
Mesec	**Projekat 1**	**Projekat 2**	**Projekat 1**	**Projekat 2**
1	✓		✓	
2		✓	✓	
3	✓		✓	
4		✓	10€	✓
5	✓		10€	✓
6	10€	✓	10€	✓
7	10€	10€	10€	10€
Ukupno vraćen novac	**30€**		**50€**	

Tabela 1: 100% posvećivanje projektu sve dok se ne završi (Scenario 3) se više isplati

	Scenario 1	Scenario 2	
Mesec	**Izdanje 1**	**Izdanje 1**	**Izdanje 2**
1	✓	✓	
2	✓	✓	
3	✓	✓	
4	✓	6€	✓
5	✓	6€	✓
6	✓	6€	✓
7	10€	6€	4€
Ukupno vraćen novac	**10€**	**28€**	

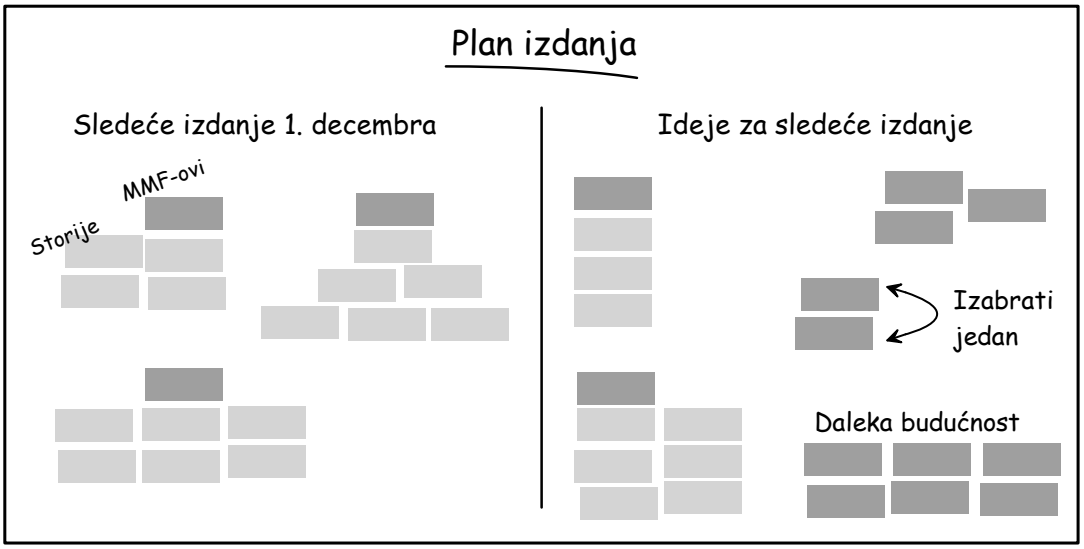
Tabela 2: Razvoj podeljen na dva izdanja (Scenario 2) se više isplati

U nekom drugom primeru pretpostavka neka bude sledeća: razvojni tim dobija zadatak da razvije jedan softverski proizvod za šest meseci. Naravno, kompanija i u ovom slučaju želi da što pre vidi povratak uloženih sredstava. Tim ima dve mogućnosti: ili da završi kompletni proizvod i izdaje ga posle šest meseci, ili da sakupi najvrednije mogućnosti proizvoda, izda taj (delimičan) softver posle trećeg meseca, a nakon toga

implementira preostale mogućnosti i izda kompletni proizvod do kraja šestog meseca. Kao što se može videti (tabela 2), drugo rešenje je mnogo bolje, jer iako softver još nije kompletiran (pa ni ne vredi toliko), on već počinje polako da vraća uložena sredstva posle trećeg meseca.

Iz ovih primera je već jasno koliko mogu male odluke u planiranju da utiču na visinu povratka uloženih sredstava. Iz drugog primera se može zaključiti da se isplati raditi česta izdanja. Preporučuje se korišćenje **minimalne tržišne mogućnosti** (eng. *Minimum Marketable Feature*, *MMF*). *MMF* je najmanji skup funkcionalnosti ili mogućnosti koji ima vrednost na tržištu. Pri razvoju softvera treba uz pomoć *on-site* klijenata naći one funkcije koje imaju najveću vrednost na tržištu (to su uglavnom jedinstvene mogućnosti sa kojima konkurentni proizvodi ne raspolažu). Moguće je te funkcije zamisliti kao storije koje se mogu grupisati u *MMF*-ove. U drugim slučajevima bolje je obrnuti postupak (zamisliti *MMF*, a zatim ga dekomponovati na storije). Zatim treba te *MMF*-ove raspodeliti po mogućim izdanjima softvera.

Sve ovo ne bi moglo da funkcioniše bez jednog adekvatnog plana. Postoje dva tipa plana: *scopebox*-ovan plan (unapred fiksira i definiše mogućnosti, a datumi izdanja se određuju dinamično) i *timebox*-ovan plan (unapred fiksira datume izdanja, a mogućnosti se raspodele u skladu sa tim datumima). Nažalost, veliki broj kompanija fiksira obe komponente (i vreme i mogućnosti) što se nije pokazalo dobrim rešenjem. *Timebox*-ovan plan je popularniji i prema mnogima – bolji. Ovaj tip plana izdanja počinje određivanjem datuma izdanja. Posle toga igrom planiranja treba odrediti *MMF*-ove, storije, proceniti ih i odrediti njihove prioritete. Preporučuje se da se prvo rade najjeftinije storije, ali sa najvećom tržišnom vrednošću. Sve ovo može da bude dosta težak posao pošto su mogućnosti skoro beskrajne. U tom slučaju planiranje u *poslednjem odgovornom trenutku* može pomoći. Konačan raspored storija je plan izdanja razvojnog tima (slika 7) na kojem se određuju storije za trenutno izdanje, ali se navedu ideje i za sledeća izdanja.



Slika 7: Plan izdanja

2.5.3. Igra planiranja

Igra planiranja (eng. *The Planning Game*) kombinuje stručnost celog tima radi kreiranja realističnog plana. Pretpostavlja se da klijenti imaju najviše znanja o *vrednostima*: šta firme najviše vrednuju. S druge strane, programeri imaju najviše znanja o *troškovima*: koliko košta implementacija i održavanje neke funkcije ili mogućnosti. Njihov zajednički napor da se *maksimizira* vrednost projekta uz *minimiziranje* troškova se zove igra planiranja.

U igri planiranja programeri igraju ulogu *procenjivača* – oni procenjuju koliko će vremena trebati da se implementira neka storija. Dok programeri procenjuju, *on-site* klijenti određuju *prioritet* te storije – neke storije vrede više od drugih, pa su i važnije. Ovako opisan scenario igranja se ponavlja za svaku storiju. Rezultat igre planiranja je sam plan.

Ova igra je relativno teška, zato se preporučuje da svi sede ispred jednog velikog stola. Klijenti napišu storije na stori-listiće, programeri ih procene, a zatim ih klijenti rasporede na tabli u skladu sa njihovim prioritetom. Tokom igre se vrši komunikacija između prisutnih koja može da izmakne kontroli. Razlog može biti nezadovoljstvo davanja prioriteta storijama ili nerazumevanje suprotne strane. Na primer, *on-site* klijenti će često postaviti pitanje, zašto je neka storija toliko skupa. U tom slučaju očekuje se od programera da to objasne klijentima na jasan način (*zajednički jezik* može pomoći).

2.5.4. Upravljanje rizikom

Upravljanje rizikom (eng. *Risk Management*) omogućava timu da napravi i da ispoštuje dugoročna obećanja. Poznato je da je svaki projekat izložen raznim rizicima, ali bez obzira na sve to, uprava kompanije očekuje da tim pokaže rezultate.

Rizici deluju na procene kao množioci. Koliko rizik može da utiče na procene zavisi od tumačenja, i firma bi trebala da ima svoj prihvaćen stav o tome. Ukoliko nema, može da se koristi *Shore*-ova i *Warden*-ova tabela (tabela 3). Ova tabela pokazuje verovatnoću ostvarivanje ciljeva na vreme. Tabela ima dva pogleda: *rigorozan* (baziran na simulatoru *Riscology* od *DeMarco*-a i *Lister*-a) i *rizičan* (preuzet od *Little*-a). Na primer, kod rizičnog pogleda verovatnoća da se sve završi na vreme je 10%, dok se dupliranjem vremenske procene dobija verovatnoća od 50%. Ako se rigorozno poštuju *XP*-principi, zbog smanjenog rizika može se koristiti rigorozni pogled, u suprotnom bolje je koristiti rizični pogled.

	Pogled		
Verovatnoća	**Rigorozan**	**Rizičan**	**Opis**
10%	x1	x1	Skoro nemoguće („*izostaviti*“)
50%	x1.4	x2	Pola-pola („*optimistički cilj*“)
90%	x1.8	x4	Skoro sigurno („*obećati*“)

Tabela 3: Rigorozan i rizičan pogleg po Shore-u i Warden-u

Ova tabela je tako napravljena da već sadrži „*opšte*“ rizike koji se mogu javljati kod bilo kog projekta (bolest, odmor, novi zahtevi, itd.). Mnogo važniji su „*jedinstveni*“ rizici. Zbog toga se preporučuje pravljenje tzv. **cenzusa rizika** (eng. *Risk Census*) (*DeMarco*, *Lister*) na jednom sastanku na kojem bi celi tim bio prisutan. Vođa sastanka treba da podeli listiće prisutnima, a zatim da napiše sledeća pitanja na tablu:

1. Koji deo projekta Vas noćima drži budnim?
2. Zamislite da je prošlo godinu dana od katastrofalnog propadanja projekta i neko Vas intervjuiše o razlozima propasti. Šta biste rekli?
3. Zamislite najlepše snove o Vašem projektu. Napišite na papir sve suprotno od njih.
4. Na koji način bi mogao, bez ičije krivice, da propadne projekat?
5. Na koji način bi mogao da propadne projekat zbog zainteresovanih strana? Zbog klijenata? Zbog programera? Zbog test-osobe? Zbog uprave? Zbog Vas?
6. Kako bi mogao da uspe projekat, ali da jedna zainteresovana strana bude nezadovoljna ili ljuta?

Prisutni treba kroz *brainstorming* da napišu svoje odgovore na listiće. Negativno, pesimistično razmišljanje nije poželjno – *obavezno* je. Kad završe, treba odgovore glasno da pročitaju, a zatim kroz novu sesiju *brainstorming*-a da izmisle scenarije koji bi vodili do tih katastrofa. Posle toga treba iskonskom analizom da nađu uzroke tih scenarija. Ti uzroci su *rizici projekta*. Nakon pronalaska rizika većina tima bi mogla da se vrati svom poslu, a jedna mala grupa bi nastavila sa razmišljanjem. Za svaki rizik treba navesti *verovatnoću pojavljivanja* (na primer: velika, srednja, mala) i *pravljenu štetu na projekat ukoliko se desi* (na primer: novčani troškovi, kašnjenje u danima, gašenje projekta). Nakon klasifikacije svakog rizika, treba izfiltrirati one koji imaju malu verovatnoću i/ili malu štetu. Kod preostalih rizika treba odrediti načine rukovanja: *izbeći* (ne izvršiti rizičnu akciju), *sadržati* (rezervisati dodatni novac i vreme), ili *ublažiti* (smanjiti napravljenu štetu). Ovi načini rukovanja se mogu i kombinovati. Preciznije rečeno, kod svakog rizika treba navesti:

- **Indikatore prelaza** (eng. *Transition Indicators*) – označavaju kad će rizik postati realnost.
- **Aktivnosti ublažavanja** (eng. *Mitigation Activities*) – aktivnosti koje smanjuju napravljenu štetu. One se izvršavaju bez obzira da li je rizik postao realnost ili ne.
- **Aktivnosti neizvesnosti** (eng. *Contingency Activities*) – aktivnosti koje smanjuju napravljenu štetu. One se izvršavaju samo kad je rizik postao realnost.
- **Izloženost prema riziku** (eng. *Risk Exposure*) – određuje preporučljivu sumu novca ili dodatnog vremena za optimalno zadržavanje rizika. Ovo se izračunava tako što se verovatnoća rizika pomnoži sa napravljenom štetom. Na primer: ako je verovatnoća rizika 20%, a napravljena šteta 5000€ i 4 dana, onda je izloženost prema riziku 1000€ i 0.8 ≈ 1 dan.

Rizici se moraju aktivno nadgledati. Preporučuje se da se svake nedelje ažurira lista rizika i njihova šteta.

Pomoću izloženosti prema riziku i množioca rizika može se predvideti koliko stori-poena će još tim moći da postigne do datuma izdanja. Kod *timebox*-ovanog plana izdanja koristi se sledeća formula:

$$\text{preostalih\_poena\_usaglašen\_sa\_rizikom} = \frac{(\text{preostalo\_iteracija} - \text{izloženost\_prema\_riziku}) \cdot \text{brzina}}{\text{množioc\_rizika}}$$

Na primer: ako se koristi rigorozni pogled, do datuma izdanja ima još 12 iteracija, brzina tima je 14, a izloženost prema riziku je jedna iteracija (treba izostaviti novčani deo, a dane treba pretvoriti u iteracije), onda se može izračunati sledeće:

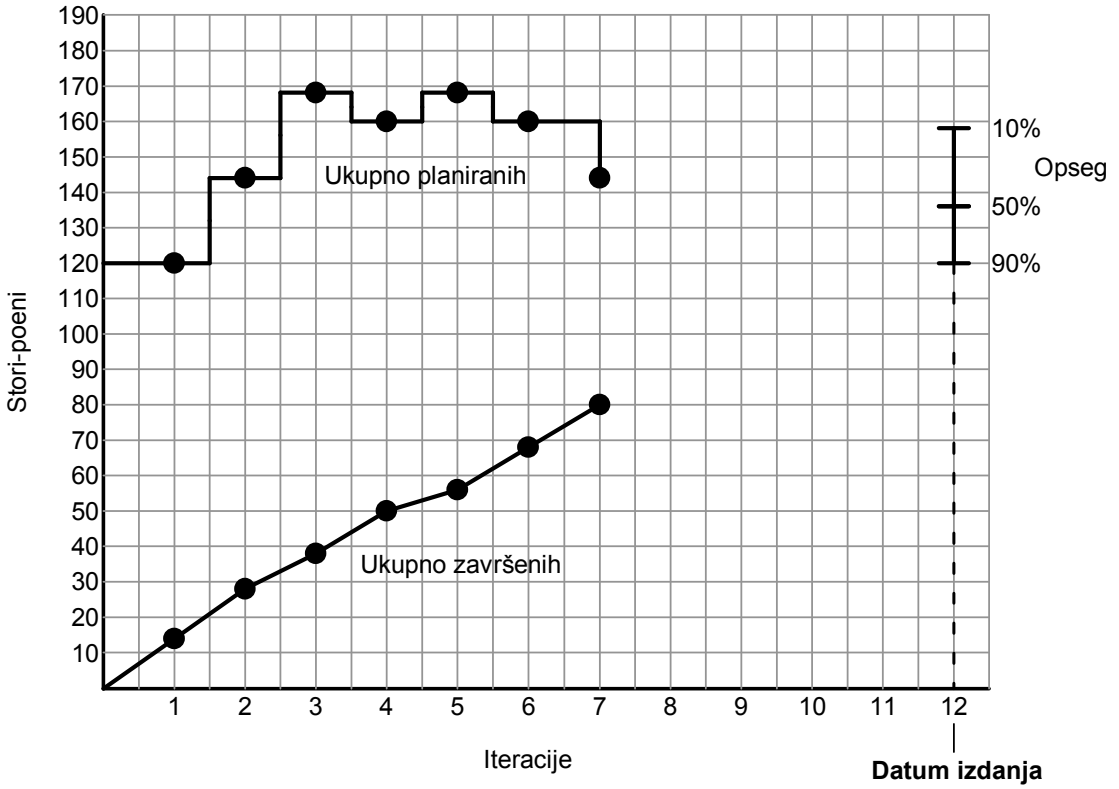
$$\text{preostalih\_poena} = (12-1)\cdot 14 = 154 \text{ poena}$$

$$10\% = 154/1 = 154 \text{ poena} \rightarrow \text{verovatnoća da se kompletira još } 154 \text{ poena do izdanja je } 10\%$$

$$50\% = 154/1.4 = 110 \text{ poena} \rightarrow \text{verovatnoća da se kompletira još } 110 \text{ poena do izdanja je } 50\%$$

$$90\% = 154/1.8 = 86 \text{ poena} \rightarrow \text{verovatnoća da se kompletira još } 86 \text{ poena do izdanja je } 90\%$$

Sve ovo se grafički može prikazati na tzv. **Burn-up dijagramu** (eng. *Burn-Up Chart*) (slika 8). Tokom svake iteracije treba navesti koliko stori-poena je postignuto, procenu ukupnog broja stori-poena na datumu izdanja i opseg preostalih poena računajući i rizik. Ovaj dijagram omogućava timu da da obećanja upravi firme. Treba upravi *obećati* da će implementirati one funkcije za koje je verovatnoća uspešnog završetka do datuma izdanja 90% ili više, a funkcije sa verovatnoćom između 50% i 90% treba navesti kao *optimističke ciljeve*.



Slika 8: Jedan Burn-up dijagram posle 7. iteracije (još 5 iteracija do datuma izdanja)

2.5.5. Planiranje iteracije

Planiranje iteracije (eng. *Iteration Planning*) pomaže razvojnom timu da odredi strukturu dnevnih aktivnosti unutar jedne iteracije. Kao što je već poznato, iteracije predstavljaju osnovu skoro svakog metoda razvoja softvera. Jedna iteracija kod *XP*-a obično ima sledeći raspored aktivnosti:

1. Demonstracija prethodne iteracije (oko trideset minuta)
2. Retrospektiva o prethodnoj iteraciji (jedan sat)

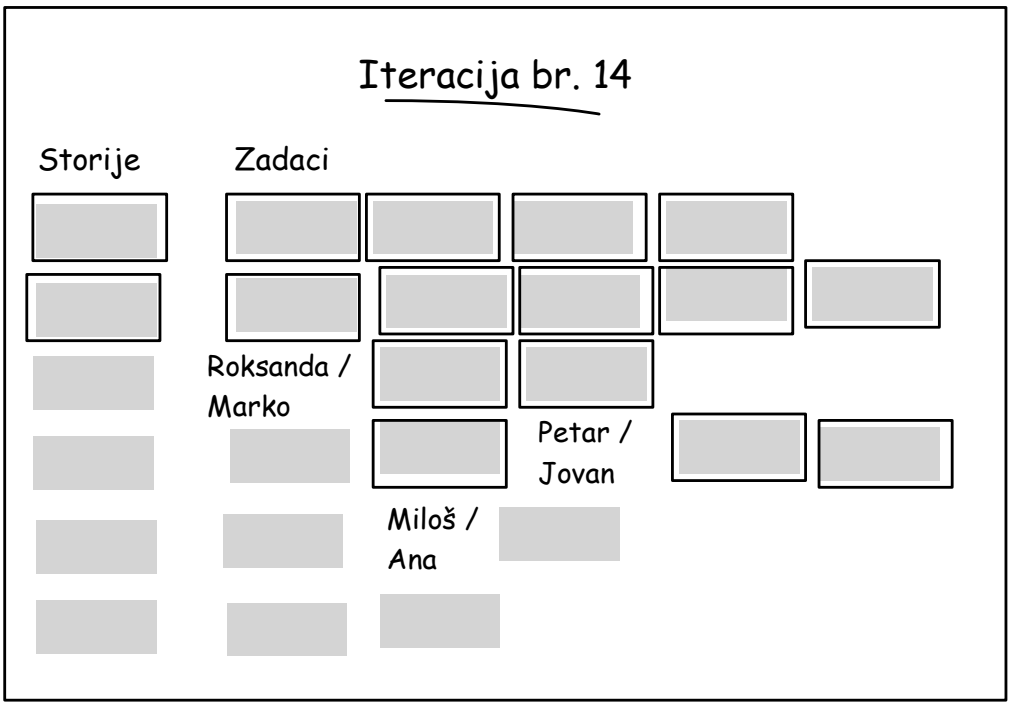
3. Planiranje trenutne iteracije (od pola sata do četiri sati)
4. Obećanje da će plan uspeti (pet minuta)
5. Razvoj storija (ostatak iteracije)
6. Izgradnja proizvoda (manje od deset minuta)

O demonstraciji iteracije i o retrospektivi je već bilo reči (odeljci 2.3.7. i 2.2.5.). Planiranje iteracije počinje izračunavanjem *brzine* prethodne iteracije. Izračunata brzina je broj stori-poena koje tim može da potroši u aktuelnoj iteraciji (više o brzini i o stori-poenima će biti u odeljku 2.5.8.). Treba izabrati storije iz napravljenog plana izdanja (pogledati odeljak 2.5.2.) tako da njihova suma u smislu stori-poena bude ista sa polaznim. Pošto su storije na planu izdanja već procenjene, a određen je i njihov prioritet, ovaj postupak ne bi trebao da traje duže od nekoliko minuta.

Posle ovoga dolazi glavni deo plana iteracije na kojem ostaju samo programeri i jedan-dva klijenta (u slučaju da treba nešto objasniti). Glavni deo plana je razbijanje storija na inženjerske zadatke. **Inženjerski zadaci** (eng. *Engineering Tasks*) su konkretni zadaci namenjeni isključivo programerima. Za razliku od storija koje su orijentisane ka klijentima i napisane su jezikom domena, inženjerski zadaci su orijentisani ka programerima i zato su napisani jezikom implementacije, na primer: „*proširiti bazu podataka*“, „*sinhronizovati dva objekta*“, „*modifikovati algoritam da bi mogao primiti JPG slike umesto BMP*“, itd. Inženjerski zadaci se isto pišu na listiće. Neki od njih će se odnositi samo na jednu storiju, dok će se neki odnositi na više od njih. Razbijanje storija na inženjerske zadatke je dizajnerska aktivnost i radi se kroz *brainstorming*. U zavisnosti od toga, koliko su programeri složni, ova sesija može da traje od pola sata do četiri sata. Nakon određivanja ovih zadataka treba ih proceniti. Za ovo su isto zaduženi programeri. Zadatke treba proceniti u *idealnim* satima, tj. koliko vremena bi zahtevala implementacija nekog zadatka u savršenim uslovima. Takođe, rezultati moraju biti navedeni u **čovek-satima** (eng. *Man-Hour*, *Person-Hour*) koji označavaju ukupan trud jednog programera da završi zadatak. Na primer: ako je nekom programeru potrebno 4 sati da završi neki zadatak, to se programiranjem u parovima (dva programera) može obaviti za dva sata. Svaka procena inženjerskih zadataka bi trebala da bude između jednog sata i šest sati: kraće zadatke treba kombinovati, dok duže zadatke treba podeliti. Kad su svi zadovoljni sa procenom, treba sabrati ove procene i uporediti dobijen broj sa istim brojem iz prethodne iteracije. Iako oni ne moraju da budu isti, trebalo bi da budu slični. S ovim je plan iteracije završen. Slika 9 pokazuje jedan mogući oblik plana.

Posle plana iteracije sledi formalno obećanje da će napravljen plan uspeti. Treba okupiti sve članove tima, i pitati ih da li mogu obećati da će plan uspeti. Očekuje se verbalno „*Da*“ od svih prisutnih. Dozvoljeno je reći „*Ne*“. Negativan odgovor je znak straha, i to je prilika za dalju diskusiju sa celim timom.

Posle ovog formalnog obećanja, svi počinju da rade. Plan iteracije se stalno ažurira tokom iteracije. Kad programeri počnu da rade na nekom inženjerskom zadatku (u paru), skidaju listić sa plana i na mesto plana napišu svoja imena da bi i drugi znali, kuda je otišao zadatak. Kad se završi zadatak, listić se vraća na svoje mesto i zaokruži se zelenim pravougaonikom. Na kraju iteracije se vrši izgradnja celog projekta koja bi trebala da traje manje od deset minuta.



Slika 9: Plan iteracije

2.5.6. Lenčarenje

Lenčarenje (eng. *Slack*) omogućava timu da pouzdano prikaže rezultate na kraju svake iteracije. U nekim literaturama je ovaj termin poznat pod imenom **buffer vreme** (eng. *Buffer Time*). Lenčarenje je u suštini dodatno vreme koje u normalnim uslovima nije potrebno da bi tim mogao da isporuči softver po planu. Međutim, uslovi nikad nisu idealni. Prema tome, i to dodatno vreme se mora iskoristiti „pametno“. Lenčarenje ima dve svrhe: može da se iskoristi za nekritične, ali bitne poslove (kad projekat napreduje po planu), i može da se iskoristi da bi tim mogao da isporuči softver po planu (u slučaju uzbune).

Postavlja se pitanje, šta se može uraditi u ovom dodatnom vremenu ukoliko projekat napreduje po planu. Dve najbolje ideje su: smanjenje tehničkog duga i istraživačko vreme. O tehničkom dugu je već bilo reči u prethodnim odeljcima. Bez obzira koliko čovek pazi, tehnički dug se vremenom nagomilava. Preporučuje se da se svi programeri bave smanjenjem tehničkog duga nekoliko sati nedeljno. **Istraživačko vreme** (eng. *Research Time*) je, međutim, nešto sasvim drugo. Naime, programeri uvek moraju biti u toku sa novim tehnologijama i tendencijama. Drugim rečima, potrebno je stalno unapređivati postojeće veštine. Istraživačko vreme je vreme koje programeri mogu iskoristiti za ove svrhe. Dužina trajanja ovog vremena je pola dana po iteraciji za svakog programera posebno. U jednom trenutku se samo jedan programer može baviti istraživanjem.

Kad postoji rizik da tim možda neće moći da isporuči softver po planu (tj. neće moći da ispuni svoje obećanje), lenčarenje se može iskoristiti kao dopunsko vreme. U zavisnosti od ozbiljnosti rizika mogu se izostaviti smanjenje tehničkog duga,

istraživačko vreme ili obe aktivnosti. Ukoliko ni to nije dovoljno, dozvoljava se i prekovremeni rad, ali samo u umerenim količinama (pogledati princip Stimulisan rad u odeljku 2.2.2.).

2.5.7. Storije

Storije (eng. *Stories*) predstavljaju glavne stavke u planu razvojnog tima. Storije nisu zahtevi, niti slučajevi korišćenja kako bi neki pomislili. One su jednostavni zadaci koji čekaju da se implementiraju. One su veoma sažeto napisane, i zato nisu dovoljne za uspešnu implementaciju zadataka. Umesto toga, one predstavljaju osnovu za dalju diskusiju o njima. Zato klijenti sede zajedno sa ostatkom tima jer imaju najviše znanja o njima. Dve najvažnije karakteristike svake storije su:

- One predstavljaju vrednost za klijente i zato su napisane u jeziku domena, a ne na jeziku implementacije. Najbolje storije su upravo one koje su napisane od strane klijenata.
- One moraju imati jasne kriterijume završetka. Naime, klijenti moraju imati ideju o tome, kad se smatra da je storija završena.

Storije se pišu na listiće od papira ili kartona, i zovu se stori-listići. Veličina listića može da varira, neki preporučuju da budu dovoljno mali da bi stali u džep (8cm x 12cm). Postavlja se pitanje, zašto se koriste listići. Njihova najveća prednost je da su opipljivi i mobilni. Nije ni čudo što skoro svi principi *XP*-a koriste ove listiće na neki način. Oni se koriste na svim sastancima (bez obzira da li je reč o sesiji *brainstorming*-a ili o grupisanju srodnih ideja u vidu listića na stolu ili na magnetnim tablama), koriste se u okviru informativnog okruženja, kod pravljenja plana, itd.[8] Baš zbog ovih razloga nema smisla kompjuterizovati ove listiće.

Pored uobičajenih storija postoje i specijalizovane storije. One uključuju: storije dokumentacije (odeljak 2.4.7.), nefunkcionalne storije (odeljak 2.6.8.), storije o greškama (odeljak 2.4.2.), *spike*-storije (odeljak 2.6.7.), itd.

2.5.8. Procenjivanje

Procenjivanje (eng. *Estimating*) omogućava timu da predvidi koliko dugo će nešto trajati. U igri planiranja, ali i pri planiranju iteracije, programeri su zamoljeni da procene koliko će vremena nešto zahtevati. Po mnogima, ovo je jedan od najtežih principa jer se zasniva na predosećanju. Jedan od najvećih problema je to da se razne smetnje ne mogu predvideti. Međutim, ovo nije tako velik problem. Umesto toga, programeri bi trebali da zamisle da rade pod idealnim uslovima. Drugim rečima, oni bi trebali da daju svoje procene u vidu **idealnih inženjerskih dana** (eng. *Ideal Engineering Days*). Idealni inženjerski dani su poznati i pod nazivom **stori-poeni** (eng. *Story Points*). Na primer, ako jedna storija „košta“ dva stori-poena, onda to znači dva idealna dana (u savršenim uslovima)[9]. Međutim, programeri će u većini slučajeva

[8] Neki ljudi čak nose nekoliko praznih listića u džepu kad su van prostorije razvojnog tima. Možda će se sresti sa nekim važnim zainteresovanim stranama i razgovarati o projektu. Ukoliko zainteresovana strana ima neku preporuku ili ideju, dovoljno je samo da izvadi listić da bi sama napisala tu ideju u obliku storija. Ovo povećava poverenje između zainteresovanih strana i tima, jer se vidi da tim daje sve od sebe da zadovolji njihove zahteve.

[9] Storije se procenjuju u obliku idealnih inženjerskih dana, dok se inženjerski zadaci procenjuju u obliku idealnih sati (pošto su manji od storija).

pogrešiti, jer idealni dani nisu realni dani. Ovo nije problem jer *XP* ima jedan jednostavni ali moćni alat: *brzinu*.

Kao što je već rečeno, procene programera uglavnom nisu tačne. Međutim, ono što je jako važno je to da su ove procene *konstantno netačne*. S druge strane, količina smetnji je u većini slučajeva takođe *konstantna* kroz iteracije. Kao rezultat, moguće je procene pouzdano transformisati u kalendarsko vreme korišćenjem nekog faktora skaliranja. Iz prethodno iznetih informacija se već lako definiše pojam brzine. **Brzina** (eng. *Velocity*) je broj stori-poena koji se može kompletirati u jednoj iteraciji. Brzina pruža dobru povratnu spregu o stvarnom uspehu tima u prethodnoj iteraciji (pogledati princip planiranja iteracije u odeljku 2.5.5.). Naime, kad tim *potceni* količinu posla u tekućoj iteraciji, neće moći da završi posao do kraja iteracije, i kao rezultat, brzina tima će se smanjiti. Zbog smanjene brzine, tim će imati manju količinu stori-poena za novu iteraciju, tj. imaće manje posla, a to će omogućiti uspešno okončanje iteracije i – ako količina vremena za lenčarenje to dopušta – završetak poslova iz prethodne iteracije. S druge strane, ako tim *preceni* količinu posla u tekućoj iteraciji, završiće više od potrebnog za tu iteraciju, i kao rezultat, brzina tima će porasti. Zbog veće brzine, tim će moći da uvrsti više posla u plan nove iteracije. Kod iskusnih *XP*-timova brzina je konstantna kroz iteracije. Neke ideje za poboljšanja brzine mogu biti:

- **Smanjiti tehnički dug** – o tome je već bilo više reči u prethodnim odeljcima
- **Više angažovati klijente** – treba klijente zamoliti da budu *on-site* klijenti
- **Podržati stimulisan rad** – prekovremeni rad treba da se izbegava
- **Rasteretiti programere od nepotrebnih poslova** – treba zamoliti menadžera projekta da zaštiti programere od suvišnih sastanaka, smetnji, itd.
- **Obezbediti potrebne resurse** – ako se programeri žale da računari nisu dovoljno brzi ili zahtevaju brži internet, treba uvažiti njihove želje, naročito ako će to povećati njihovu produktivnost.

2.6. Grupa principa broj 5: Razvijanje

Za uspešan razvoj softvera potrebna je međusobna saradnja između svih članova tima. Programeri implementiraju zadatke, klijenti razmišljaju o novim zahtevima i pripremaju se da budu dostupni kad neko zahteva njihovu pomoć, a test-osobe testiraju implementirane funkcije. Međutim, razvoj softvera je skup. *XP* smatra da je poboljšanjem unutrašnjeg kvaliteta koda i dizajna moguće smanjenje troškova. Sledećih devet principa pomažu u pisanju kvalitetnog softvera: inkrementalni zahtevi, klijentski testovi, razvoj usredsređen ka testiranju, refaktorisanje, jednostavan dizajn, inkrementalni dizajn i arhitektura, *spike*-rešenja, optimizacija performansi i istraživačko testiranje.

2.6.1. Inkrementalni zahtevi

Inkrementalni zahtevi (eng. *Incremental Requirements*) omogućavaju timu da počne sa radom dok klijenti određuju detalje zahteva. Kod uobičajenih metoda postoji posebna faza analize i definisanja u kojoj se pravi detaljan dokument specifikacije zahteva. *XP* nema ovu fazu, ali to ne znači da detaljni zahtevi nisu potrebni. Umesto posebne faze, *XP* koristi „*živu*“ specifikaciju zahteva, tj. klijenti sede zajedno sa timom i vrši se verbalna komunikacija.

Da bi tim mogao da počne sa radom pre nego što su svi zahtevi određeni, koristi se *inkrementalni* rad na zahtevima. Ovo omogućava da svi članovi tima rade paralelno. Rad na zahtevima počinje pravljenjem deklaracije vizije, a zatim plana izdanja. Tokom planiranja izdanja klijenti i programeri igraju igru planiranja u kojoj svaka storija dobija svoj prioritet i svoju procenu, na osnovu kojih se pravi plan iteracije. Nakon ovoga, programeri već mogu da počnu sa implementacijom izabranih storija, dok klijenti formulišu klijentske testove za trenutne storije i određuju kad će se smatrati da su „gotove-gotove“, a istovremeno razmišljaju i o budućim storijama.

2.6.2. Klijentski testovi

Klijentski testovi (eng. *Customer Tests*) pomažu u pravilnom implementiranju domenskih koncepata. Pošto klijenti imaju najviše znanja o domenu, jasno je da su oni zaduženi za pisanje ovih testova.

Pravljenje klijentskih testova prolazi kroz tri faze, a to su: opis, demonstracija i razvoj. U fazi *opisa on-site* klijent treba da pogleda storije i da predvidi one aspekte za koje postoji velika verovatnoća da će ih programeri pogrešno shvatiti. Znači, ova faza služi za eliminisanje nesporazuma, zato je važno da se ona održi još na početku iteracije. Kad je klijent identifikovao potencijalne opasnosti u storijama, treba okupiti tim i održati kratko predavanje o njima. U drugoj fazi (*demonstracija*) treba demonstrirati te scenarije celom timu kroz konkretne primere. Ove primere je najbolje prikazati u obliku tabela. U trećoj fazi (*razvoj*) programeri implementiraju te testove da budu deo deseto-minutne izgradnje.

Kao primer, može se navesti pravljenje softvera koji simulira digitron. Jedna storija softvera je „*zaokruživanje*“. Klijent pretpostavlja da svi znaju kako se vrši zaokruživanje kad je poslednja cifra veća od 5 ili manje od 5, ali možda ne znaju pravila kad je poslednja cifra *jednaka* sa 5. Zato on okupi tim i objasni im: „*Ako je prva cifra koja se želi odbaciti jednaka sa 5, a iza te cifre još postoje decimale različite od nule, onda se vrši zaokruživanje nagore. Ukoliko je prva cifra koja se želi odbaciti jednaka sa 5 i nema više decimala iza te cifre, onda se vrši zaokruživanje nagore, ako je poslednja*

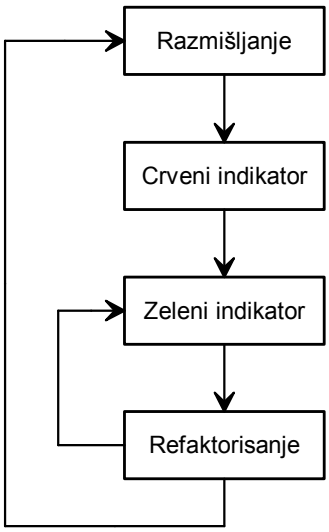
cifra koja se želi zadržati neparna, a u suprotnom se vrši zaokruživanje nadole.“ Posle ovoga klijent još demonstrira ova pravila konkretnim primerima na tabli u obliku tabele (tabela 4).

Broj	Zaokruživanje	Rezultat
4.2352	nagore	4.24
4.235	nagore	4.24
4.225	nadole	4.22

Tabela 4: Klijentski test

2.6.3. Razvoj usredsređen ka testiranju (TDD)

Razvoj usredsređen ka testiranju (eng. *Test-Driven Development*, *TDD*) omogućava programerima da budu sigurni da njihov kod radi onako kako treba. Ljudi često greše, i to se vidi i na softverskim proizvodima. *TDD* je izmišljen sa ciljem da odmah obavesti programera kad napravi neku grešku. Zapravo, većina razvojnih alata već ima mogućnost provere sintakse programskog koda u letu, ali *TDD* ide korak dalje. On je zapravo jedna tehnika koja se sastoji iz rapidnih ciklusa testiranja, kodiranja i refaktorisanja. Po studiji *Janzen*-a i *Saiedian*-a, *TDD* znatno smanjuje broj grešaka u kodu (a takođe poboljšava dizajn). U slučaju da se pronađe neka greška tokom *TDD*-a, ona se lako ispravlja jer *TDD* promoviše napredak u malim koracima (tj. *inkrementalno*).



Slika 10: Koraci TDD-a

Suština *TDD*-a leži u njenim kratkim ciklusima. Kad se jedan ciklus završi (koji traje uglavnom pet minuta), celi postupak se ponavlja, sve dok implementacija nije završena. U svakom ciklusu se vrši testiranje, kodiranje i refaktorisanje. Svaki ciklus se sastoji iz sledećih koraka (slika 10):

1. **Razmišljanje** – svaki ciklus počinje sa ovim korakom. Prvo treba smisliti, kako bi trebao da se ponaša kod koji se želi razviti. Zatim treba smisliti mali korak

(inkrement) koji će realizovati to ponašanje. Na kraju, treba formulisati jedan test koji će pokazati da inkrement funkcioniše na odgovarajući način. Ovaj korak je možda najteži deo *TDD*-a jer zahteva razmišljanje u malim koracima, a ne u vidu kompletnih rešenja.

2. **Crveni indikator** – sad treba napisati test. On bi trebao da bude veoma kratak, pošto je i celi inkrement kratak. Kad je test napisan, treba ga pokrenuti. Pošto još inkrement nije napisan, test neće proći što će rezultovati crvenim indikatorom.
3. **Zeleni indikator** – sledi ubacivanje samog inkrementa. On bi trebao da bude jako kratak (otprilike pet linija koda) – jedva dovoljno da prođe test. Nakon toga treba ponovo pokrenuti test: sad bi trebalo da bude sve u redu, što se označava zelenim indikatorom.
4. **Refaktorisanje** – sad je odlična prilika da se refaktoriše napisan kod (uključujući kod iz prethodnih ciklusa). Posle svakog refaktorisanja treba ponovo pokrenuti testove. Više o ovome će biti reči u odeljku 2.6.4.
5. **Ponavljanje** – kad je programer spreman da doda novo ponašanje, trenutni ciklus se završava i počinje novi. Programski kod se svakim uspešnim ciklusom proširuje jednim malim inkrementom.

Teško je raditi *TDD* bez specijalizovanih alata. Najpopularniji alati su iz slobodne *xUnit* familije, kao što su *JUnit* (za *Javu*) i *NUnit* (za *.NET*).

2.6.4. Refaktorisanje

Refaktorisanje (eng. *Refactoring*) omogućava programerima da poboljšaju kvalitet napisanog koda bez promene ponašanja istog. Refaktorisanjem se poboljšava dizajn programa. Ono se vrši u malim koracima radi lakše identifikacije novih grešaka tokom refaktorisanja. Zato je važno ovo vršiti zajedno sa testovima. Razvoj usredsređen ka testiranju (*TDD*) zajedno sa refaktorisanjem predstavlja moćnu tehniku koja poboljšava kvalitet koda smanjenjem broja grešaka i poboljšanjem dizajna.

Refaktorisanje pruža jedan novi pristup dizajniranju: **reflektivno dizajniranje** (eng. *Reflecting Desing*) koje pored kreiranja novog dizajna omogućava analiziranje postojećeg dizajna radi poboljšanja istog. Refaktorisanje je danas već napredna disciplina programiranja, i ovde će biti opisano samo kratko. Neke situacije u kojima, po *Flowler*-u, *Beck*-u, *Shore*-u i *Warden*-u, refaktorisnje može biti korisno su (važi za objektno-orijentisane jezike):

- **Postoji problem kohezije u kodu.** Na primer, razne nepovezane promene uslovljavaju izmene koda u jednoj istoj klasi. Ovo je znak da ta klasa možda ima previše koncepata, i rešava se razbijanjem te klase na manje celine (postupak je poznat pod imenom ***Divergent Change***). Moguća je i obrnuta situacija: potrebno je menjati kod u više klasa, iako je reč samo o jednom konceptu. Ovo je znak da je taj koncept rascepkan u više klasa i potrebno je spojiti tu ideju u jednu klasu (postupak se zove ***Shotgun Surgery***).
- **Predstavljanje koncepata sa primitivnim tipovima.** Na primer: valuta (kao što je evro) se može predstaviti primitivnim tipom, na primer `integer`-om. Međutim, bolje je ovaj koncept predstaviti u obliku posebne klase (postupak se zove ***Primitive Obsession***). Problem je sličan i u slučaju da se neki koncept predstavlja sa više primitivnih tipova (na primer: adresa). Ukoliko se neka polja uvek koriste zajedno, to je znak za ovaj problem, i takođe se rešava enkapsulacijom koncepta u jednu klasu (postupak se zove ***Data Clumps***).

- ***Null*-reference.** Iako služe za predstavljanje nepostojećih podataka i grešaka, one predstavljaju izvor velikog broja budućih grešaka zato što programeri često ne znaju šta da rade sa njima. Zato se preporučuje da se zabrani prosleđivanje *null* vrednosti parametrima metoda, konstruktorima i poljima (sem ako neka korišćena biblioteka to eksplicitno ne zahteva). Za signalizaciju grešaka bolje je koristiti izuzetke. Ovaj postupak je poznat pod imenom ***Coddling Nulls***.

2.6.5. Jednostavan dizajn

Jednostavan dizajn (eng. *Simple Design*) omogućava da se dizajn menja u skladu sa novim zahtevima. Kod *XP*-a glavno pitanje u vezi dizajna je: „*Šta je najjednostavnija stvar koja bi proradila?*“ Umesto da dizajn bude spreman za promene pomoću raznih proširenja, *XP* smatra da je bolje imati takav dizajn koji je tek dovoljan za trenutne potrebe. Međutim, u ovakvom dizajnu se isto krije spremnost za kasnije promene. Naime, kod uobičajenih metoda sa posebnom fazom dizajna, dizajnira se tako da se bori protiv promena, praveći robusnu arhitekturu sa proširenjima. Ali ovakav dizajn će moći da se bori samo sa predviđenim promenama. Po *XP*-u, predviđanjem se ne mogu identifikovati sve promene. Zato bi bilo najbolje da dizajn bude toliko fleksibilan da bi mogao lako da uvrsti sve promene. Po *Beck*-ovom mišljenju, ovo se najbolje postiže čistim i elegantnim, tj. jednostavnim dizajnom.

Pri nastojanju da se postigne jednostavan dizajn mogu se uvažiti sledeće preporuke:

- **„Neće biti potrebno“** – kad se želi ubaciti nešto novo, treba se prvo zapitati da li trenutne storije to zahtevaju. Ako je odgovor negativan, ne treba ništa ubaciti. Takođe, ovo podrazumeva i to da kad nešto postane suvišno (na primer, korisnici nisu koristili neku mogućnost, pa je doneta odluka da se ona odstrani), onda to treba izbrisati i iz programskog koda. Ako kasnije ponovo zatreba, stara verzija softvera je dostupna iz repozitorijuma.
- **Jednom i samo jednom** – treba svaki koncept izražavati jednom i samo jednom. Ovo ne podrazumeva samo brisanje dupliranog koda, nego i težnju da svaki koncept ima svoj „*dom*“, tj. svoju klasu. Ovo je u tesnoj vezi sa postupkom *Primitive Obsession* iz refaktorisanja.
- **Samo-dokumentujući kod** – predstavlja težnju da se kodira bez komentarisanja. Ukoliko programer smatra da je potrebno nešto komentarisati, onda je to znak da je to parče koda komplikovano. Takođe, ako neki drugi članovi tima (ili neki drugi tim) smatraju da je nešto komplikovano, onda to verovatno i jeste. Naravno, to ne znači da je komentarisanje nepotrebno, ali – ukoliko je to moguće – treba kodirati tako da to bude dovoljno jasno i bez komentara.

2.6.6. Inkrementalni dizajn i arhitektura

Inkrementalni dizajn i arhitektura (eng. *Incremental Design and Architecture*) omogućavaju programerima da rade paralelno i na storijama i na dizajnu. Naime, klijenti zahtevaju da softver ima tržišnu vrednost već na početku projekta. Inkrementalni i evolutivni dizajn omogućavaju programerima da završe storije na vreme, a da istovremeno imaju i arhitekturu koja zadovoljava trenutne potrebe.

Inkrementalni dizajn koristi koncepte predstavljene kod *TDD*-a, tj. sve se odvija u malim koracima. Inkrementalni dizajn se odvija u najmanje tri faze. U prvoj fazi se kreira najjednostavniji dizajn koji je tek dovoljan da zadovolji trenutne potrebe (tj. da reši trenutni problem). Znači, bez obzira o čemu je reč – o metodi, klasi ili o arhitekturi

– treba biti što konkretniji. Ovo je u suprotnosti sa filozofijom da programeri moraju već od početka da misle u apstrakcijama. *XP*, međutim, ne odbacuje potrebu apstrakcije, nego je samo odlaže. U drugoj fazi već počinje inkrementalno usavršavanje početnog dizajna, u kojoj on postaje malo više uopšten, ali samo do tog nivoa da reši novonastale probleme. U trećoj fazi uopštenje, tj. generalizacija postaje još veća, ali ponovo samo do potrebnog nivoa. Ove faze se ponavljaju kad god je to potrebno.

2.6.7. Spike-rešenja

***Spike*-rešenja** (eng. *Spike Solutions*) koriste istraživanja radi sakupljanja informacija. Kad neki programer naiđe na neki specifični problem, umesto da nagađa o pravom rešenju, pita *on-site* klijente. Međutim, neki problemi su u programerskom domenu. U tom slučaju treba sprovesti jedan eksperiment koji se zove *spike*-rešenje. Cilj ovog eksperimenta je da se nađe odgovor na jedno specifično pitanje. Rezultat istraživanja je jedan mali samostalni program koji sadrži odgovor na postavljeno pitanje (mada može biti i jedan metod unutar samog projekta). Na kraju eksperimenta dobijen rezultat (tj. taj mali program) se odbacuje ili se stavlja u repozitorijum (može se kreirati folder pod nazivom `Spike` za ove svrhe). *Spike*-rešenja se mogu koristiti i u slučaju da programer ima neko pitanje u vezi korišćenog dizajna.

Spike-rešenja su odlična i kada se dva programera ne slažu oko nečega. U tom slučaju je najbolje da oba programera naprave svoje *spike*-rešenje. Na kraju uporede dobijene rezultate i izvuku zaključak. Po nekima ovo je najveća korist *spike*-rešenja. Čak se mogu napraviti posebne storije zvane *spike*-storije, kad razvojni tim mora da donese neke odluke u vezi dizajna i slično.

2.6.8. Optimizacija performansi

Optimizacija performansi (eng. *Performance Optimization*) omogućava brže izvršavanje softverskog proizvoda. Kad se korisnici žale na sporost proizvoda ili kad sam tim proceni da brzina izvršavanja nije na odgovarajućem nivou, to je znak da se uradi optimizacija performansi.

Najbolji način merenja performansi je pisanje tzv. **testova performansi** (eng. *Performance Tests*) koji bi trebali da koriste baze podataka (takođe preko mreže) ili da dotaknu fajl-sistem. Oni pored testiranja ispravnosti softvera prikazuju i neke statističke podatke o merenju. Ovi testovi su relativno spori u odnosu na testove modula i integracione testove. Oni u suštini predstavljaju specijalan slučaj ***end-to-end* testova** (specijalan slučaj ovih testova su i *testovi prihvatanja*). Posle optimizacije treba ponovo pokrenuti ove testove radi upoređivanja izmerenih rezultata.

Treba optimizovati samo kad je to stvarno potrebno. Naime, optimizacija ima dva nedostatka: često može dovesti do složenijeg koda koji je više ranljiv na greške, a s druge strane odvraća pažnju od implementacije storija. U tom slučaju pravljenje posebne storije – nefunkcionalne storije – može pomoći.

2.6.9. Istraživačko testiranje

Istraživačko testiranje (eng. *Exploratory Testing*) omogućava test-osobama da identifikuju nedostatke u načinu razmišljanja tima. Kod uobičajenih metoda postoji posebna faza testiranja, a verovatno firma ima i odvojen departman za testiranja softvera. *XP*-tim radi testiranje kroz celi životni vek projekta i svi članovi tima su odgovorni za kvalitet isporučenog softvera. Međutim, ovo ne znači da test-osobe nisu potrebne. Iako *XP* teži ka razvoju softvera bez grešaka (i zato forsira *TDD*, kao i sve ostale principe), to još ne garantuje da će proizvod biti bez grešaka. Test-osobe koriste

istraživačko testiranje da bi pronašle one greške koje su ostale neprimećene i od strane programera i od strane klijenata. Međutim, glavni cilj test-osoba nije pronalaženje grešaka. Naime, smatra se da su te neprimećene greške rezultat nekih problema u samom procesu razvoja. Prema tome, glavni cilj test-osoba je rešavanje ovih problema, ispravljajući proces tako da teži ka razvoju bez grešaka.

Istraživačko testiranje veoma liči na *TDD* i inkrementalni dizajn: umesto pravljenja kompletnih test-sistema, testiranju se pristupa inkrementalno. Na početku, test-osoba napiše jedan jednostavan test. Ovaj mali test pruža neku ideju za pravljenje novog testa koji će biti malo komplikovaniji od prethodnog. Na osnovu drugog testa pravi se još komplikovaniji test. U međuvremenu test-osoba dobija jednu novu ideju i počinje je graditi na isti način kao i prethodnu. Testira se samo završen proizvod, tj. onaj proizvod, čije su storije „gotove-gotove". Pri testiranju koriste se četiri sledeća alata:

- **Dozvole** (eng. *Charters*) – kad test-osobe počnu sa testiranjem, već imaju neku ideju o tome, šta žele testirati. Jedna takva ideja se zove dozvola (da se nešto testira). Jedna takva dozvola može biti neka storija (na primer, testirati storiju „*Registracija novih članova*"), veza između raznih storija (na primer, ako su „*Registracija novih članova*" i „*Mailing-lista*" dve storije, onda se može testirati ispravno dodavanje novih članova na *mailing*-listu), itd. Dozvole su praktično ekvivalentne storijama (ali imaju značaj samo za test-osobe) i mogu se napisati na listiće.
- **Posmatranje** (eng. *Observation*) – neke greške se ne mogu primetiti automatizovanim testovima. U tom slučaju živo posmatranje rada softvera je najbolje. Na primer, test-osoba može primetiti da jedno polje za unos dozvoljava i unos slova, iako bi trebalo da dozvoli samo unos brojeva, ili test-osoba primećuje da pod nekim uslovima hard disk uvek počinje intenzivno da radi iako ne bi trebao, itd. Ovo test-osobama može biti novi izvor ideja za dalje testiranje.
- **Beleženje** (eng. *Notetaking*) – tokom testiranja lako se može zaboraviti gde se test-osoba trenutno nalazi i u kom pravcu ide. Takođe, često se dešava da test-osoba pronađe neku grešku i zaboravi kako je došla do te greške. Zato je važno voditi beleške ili koristiti sofisticirani softver za snimanje ekrana.
- **Heuristike** (eng. *Heuristics*) – na početku ovog principa je rečeno da jedan test uslovljava sledeći na inkrementalni način. Heuristike predstavljaju tehnike koje pomažu u pronalaženju inkrementa. Postoji veliki broj heuristika, jedna od njih je i **granično testiranje** (eng. *Boundary Testing*): ako se na primer testira polje za unos koje dozvoljava unos brojeva između 0 i 500, onda treba testirati granične brojeve (0, 1, 499, 500), neke brojeve na sredini (250, 251), ali i nedozvoljene brojeve blizu granice (-1, 501).

Neki doživljavaju testiranje kao način javnog sramoćenja truda programera, ali ne treba zaboraviti da je bolje greške primetiti pre isporuke softvera nego posle. Mada, mora se priznati da je rad test-osoba ponekad dosta „ekstreman" jer u najgorem slučaju uključuje „sedenje" na tastaturi, čupanje mrežnog kabla usred pisanja u bazu podataka, resetovanje računara tokom snimanja podataka na hard-disk ili namerni hakerski napad na server. Bez obzira na sve ovo, cilj test-osoba je da poboljšaju proces razvoja softvera, da se nađene greške ne ponavljaju sledeći put. Za identifikaciju stvarnih uzroka problema u procesu najbolje je koristiti iskonsku analizu. Zatim treba okupiti celi tim i diskutovati o mogućim rešenjima.

Zaključak

Ekstremno programiranje, izmišljeno od strane *Kent Beck*-a, je napravilo veliki bum u 90-im godinama prošlog veka koji traje još i danas. *XP* je doveo agilne principe do ekstrema napravivši metod koji se radikalno razlikuje u odnosu na uobičajene metode.

Odmah pri nastanku metoda, skeptici su počeli uveliko da ga kritikuju zbog njegove radikalne ideologije. Najčešće mete negodovanja su bile nedovoljno dokumentovanje (uključujući nedostatak dokumenata o zahtevima i o dizajnu), problem skalabilnosti (teško se primenjuje kod velikih projekata sa velikim timovima) i čudan pristup programiranju i testiranju. Branioci *XP*-a su na ove primedbe odgovarali tako, da bi skeptici trebali konačno da uvide da veliki broj projekata upravo propada zbog onih principa koje ti skeptici brane, tj. da ovaj metod samo pokušava na neki način da reši „očigledne“ probleme. Nastali su posebni članci, pa čak i cele knjige, posvećene kritikama *XP*-a. Dok su stručnjaci nastavili sa međusobnim kritikovanjem, neke velike kompanije poput *Microsoft*-a, *Google*-a, *Yahoo*-a i *Symantec*-a su već uveliko počele da primenjuju agilne metode poput *XP*-a i *Scrum*-a, postizući dobre rezultate.

Nakon određenog vremena se ipak iskristalisalo da *XP* više odgovara samo nekim određenim projektima. Smatra se da ga je najbolje koristiti kod onih projekata kod kojih se zahtevi često menjaju, i kod relativno malih projekata (iako je konsultantska kuća *ThoughtWorks* počela da eksperimentiše sa primenom *XP*-a kod velikih projekata). U svakom slučaju ostaje činjenica da je *XP* relativno mlad metod razvoja softvera i da je još u fazi usavršavanja. Na kraju krajeva, samo će vreme pokazati da li su primedbe *XP*-a opravdane ili ne.

Literatura

1. *James Shore, Shane Warden*: **The Art of Agile Development**; O'Reilly Media, Inc; Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo, 2007, br. str. 438
2. *Craig Larman*: **Agile and Iterative Development: A Manager's Guide**; Addison Wesley, 2003, br. str. 368
3. *Kent Beck, Martin Fowler*: **Planning Extreme Programming**; Addison Wesley, 2000, br. str. 160
4. *Kent Beck*: **Extreme Programming Explained (First Edition)**; Addison Wesley, 1999, br. str. 224

Kratka biografija

Rođen sam 11. aprila 1984. godine u Bačkoj Topoli (Vojvodina, Srbija) od oca Lasla i majke Katalin. Imam jednog starijeg brata, Zoltana. Osnovnu školu „Čaki Lajoš“ u Bačkoj Topoli sam završio 1999. godine sa odličnim uspehom i Vukovom diplomom, a 2003. godine i gimnaziju „Dositej Obradović“ u Bačkoj Topoli isto sa odličnim uspehom i Vukovom diplomom. Jula 2003. godine sam upisao Prirodno-matematički Fakultet u Novom Sadu, smer diplomirani informatičar – poslovna informatika. Do septembra 2008. godine položio sam sve ispite predviđene planom i programom. Prosek ocena mi je 8.90.

Novi Sad, oktobar, 2008.

Robert Pap

_____

UNIVERZITET U NOVOM SADU
PRIRODNO - MATEMATIČKI FAKULTET
KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Redni broj:
RBR
Identifikacioni broj:
IBR

Tip dokumentacije:	Monografska dokumentacija
TD	
Tip zapisa:	Tekstualni štampani materijal
TZ	
Vrsta rada:	Diplomski rad
VR	
Autor:	Robert Pap
AU	
Mentor:	dr Zoran Budimac
MN	
Naslov rada:	Ekstremno programiranje kao metod agilnog razvoja softvera
MR	
Jezik publikacije:	Srpski (latinica)
JP	
Jezik izvoda:	s / en
JI	
Zemlja publikovanja:	R Srbija
ZP	
Uže geografsko područje:	Vojvodina
UGP	
Godina:	2008
GO	
Izdavač:	Autorski reprint
IZ	
Mesto i adresa:	Novi Sad, Trg D. Obradovića 4
MA	
Fizički opis rada: (*broj poglavlja/ broj strana/ broj lit. citata/ broj tabela/ broj slika/ broj grafika/ broj priloga*)	(11/82/4/4/10/0/0)
FO	

Naučna oblast: Računarske nauke

NO

Naučna disciplina: Softversko inženjerstvo

ND

Predmetne odrednice, Ključne reči: ekstremno programiranje, XP, agilni

PO

UDK:

Čuva se:

ČU

Važna napomena: nema

VN

Izvod:

Razvoj softvera je jedan veoma važan i ozbiljan proces koji u većini slučajeva zahteva saradnju više ljudi. Jedan od glavnih ciljeva softverskog inženjerstva je proučavanje modela za sistematski razvoj softvera. Model procesa je razvojni plan koji definiše opšti proces razvoja softverskog proizvoda. Jedan od najranijih modela procesa je bio klasični vodopadni model. Vremenom su nastale razne varijante ovog osnovnog modela koje su već uvažile njegov najveći nedostatak: nedostatak iteracije. Ovako je nastao iterativni i evolutivni razvoj, a iz jedne radikalne ideologije koja se dosta razlikovala od dosadašnje prakse, nastali su agilni metodi. Cilj ovog rada je da prikaže osnove jednog poznatog agilnog metoda razvoja softvera: ekstremno programiranje.

IZ

Datum prihvatanja teme od strane NN veća: septembar, 2008.

DP

Datum odbrane: oktobar, 2008.

DO

Članovi komisije:

KO

Predsednik: dr Mirjana Ivanović, redovni profesor Prirodno-matematičkog Fakulteta u Novom Sadu

Član: dr Zoran Budimac, redovni profesor Prirodno-matematičkog Fakulteta u Novom Sadu

Član: dr Dragan Mašulović, vanredni profesor Prirodno-matematičkog Fakulteta u Novom Sadu

UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCES
KEY WORDS DOCUMENTATION

Accession number: **ANO**	
Identification number: **INO**	
Document type: **DT**	Monograph documentation
Type of record: **TR**	Textual printed material
Contents Code: **CC**	Graduation Thesis
Author: **AU**	Robert Pap
Mentor: **MN**	Dr Zoran Budimac
Title: **TI**	XP as a Method for Agile Software Development
Language of text: **LT**	Serbian (Latin)
Language of abstract: **LA**	en / s
Country of publication: **CP**	Serbia
Locality of publication: **LP**	Vojvodina
Publication year: **PY**	2008
Publisher: **PU**	Author's reprint
Publ. place: **PP**	Novi Sad, Trg D. Obradovica 4
Physical description: **PD**	(11/82/4/4/10/0/0)

Scientific field: Computer Science

SF

Scientific discipline: Software Engineering

SD

Key words: extreme programming, XP, agile

SKW

UC:

Holding data:

HD

Note:

N

Abstract:

Software development is a very important and serious process that – in most cases – requires teamwork. One of the central goals of software engineering is to study the models for systematic software development. A process model is a development plan that defines a general process for developing software products. One of the earliest process models was the classic waterfall model. In time, different varieties of this basic model were born that already solved it's biggest weakness: the absence of iterations. This was the start of iterative and evolutionary development, and from a radical ideology that was quite different from the current practices, the agile methods were born. The goal of this work is to illustrate the basics of a well-known agile method for software development: extreme programming (XP).

AB

Accepted by the Scientific Board on: September 2008.

AS

Defended: October 2008.

DE

Thesis defend board:

DB

President: Dr Mirjana Ivanović, full professor, Faculty of Sciences, Novi Sad

Member: Dr Zoran Budimac, full professor, Faculty of Sciences, Novi Sad

Member: Dr Dragan Mašulović, associate professor, Faculty of Sciences, Novi Sad