Trendovi razvoja aplikacija

Tihomir Katić
Mrežne tehnologije Verso d.o.o.
Freudova 5, HR – 10000 Zagreb, Hrvatska
tihomir.katic@verso.hr

1. Uvod

Razvoj aplikacija u brojnim današnjih organizacijama sličan je kao i u početcima razvijanja softverskih proizvoda. Istina, programski alati i jezici su napredniji, tehnička oprema i informacije lakše dostupni, i ukupno gledajući mogućnosti su šire, ali unatoč tomu svemu ne primjenjuju se konkretne metodologije razvoja ili su ti modeli razvoja jednaki prvim osnovnim modelima. Zbog toga ne čudi da većina organizacija i danas najviše primjenjuje metodologiju vodopada koja je ukratko i u ovom dokumentu opisana.

Ipak, brojne organizacije svjesne su nedostataka takvih osnovnih metodologija razvoja te da im one ne mogu pomoći u postizanju konkurentnih prednosti kako na regionalnom, a tako i na svjetskom tržištu. Takve organizacije svjesne su da konkurentne mogu postati ili ostati samo ako budu fleksibilni na promjene korisničkih zahtjeva tijekom razvoja, i brze u ispunjavanju tih zahtjeva tj. da se pridržavaju dogovorenih rokova i budžeta, a opet da razvijani sustavi budu visokokvalitetni. Da bi se ti zahtjevi ispunili, organizacije moraju primjenjivati netradicionalne metodologije razvoja, a u današnje doba najpopularniji su tzv. agilni modeli razvoja.

Važno je spomenuti kako novi trendovi razvoja aplikacija sve više uključuju i potrebu za izradom sigurnijeg programskog koda. Broj zlonamjernih korisnika na Internetu svakim je danom sve veći, a da bi uopće mogli širiti svoje djelovanje, moraju pronalaziti i iskorištavati sigurnosne propuste u korištenim softverskim proizvodima. Stoga danas postoje brojni statički i dinamički kontrolori programskog koda, a u ovom dokumentu opisan je i Microsoftov model sigurnog razvoja.

2. Povijesni pregled razvoja aplikacija

2.1. Osnovni stilovi programiranja

Unatrag pedeset godina povijesti programiranja razvijeno je nekoliko osnovnih stilova (ili paradigmi, ili modela) programiranja – proceduralno, funkcijsko, logičko i objektno orijentirano. Svaki od tih stilova zasnovan je na specifičnim algoritamskim apstrakcijama podataka i operacija te predstavlja posebni oblik razmišljanja o tome što je program i kako se treba izvoditi. Različite programske tehnike (uključujući i podatkovne strukture te kontrolne mehanizme) razvijene su nezavisno unutar svakog stila programiranja te su time formirani različiti oblici njihove primjenjivosti. Npr. objektno orijentirani stil i povezane tehnike, pogodni su za programe s kompliciranim podacima i logikom, dok je logičko programiranje pogodnije za jednostavnije logičke provjere zasnovane na pravilima. [15]

Ipak, moderniji jezici programiranja obično uključuju programske tehnike iz različitih stilova pa se na taj način različiti stilovi isprepliću. To ispreplitanje potrebno za potrebe implementacije velikih programskih projekata, u pravilu zbog kompleksnosti i heterogenosti projektnih zahtjeva.

Proceduralno programiranje najstarije je i najtradicionalnije. Ono je raslo od strojnih i asemblerskih jezika, čije glavne osobine odražavaju osnovna svojstva John von Neumanovih principa računalne arhitekture. Proceduralni program sastoji se od niza eksplicitnih naredbi (instrukcija) te poziva procedura (podprograma) koji se slijedno izvode. Time se obavljaju definirane operacije nad podacima i mijenjaju se vrijednosti programskih varijabli, ali i vanjsko okruženje. Programske varijable mogu se promatrati kao spremnici za podatke, slično kao i memorijske adrese u memoriji računala.

Funkcijsko programiranje je isto star stil koji je nastao iz evolucije algebarskih formula te su i njegovi elementi korišteni u prvim proceduralnim jezicima kao što je Fortran. Čisti funkcijski program je skupina međusobno povezanih (moguće i rekurzijama) funkcija. Svaka funkcija je zapis za izračun vrijednosti te je definirana kao kompozicija standardnih (ugrađenih) funkcija. Izvođenje funkcijskog programa zasniva se na pozivanju funkcija te na izračunavaju definiranih vrijednosti.

S **logičkom** paradigmom, program je predstavljen kao niz logičkih formula – aksioma (pravila i činjenica) opisujući svojstva pojedinih objekata i teorema koje je potrebno dokazati ili opovrgnuti. Programsko izvođenje je proces logičkog dokazivanja teorema preko konstrukcije objekata s opisanim svojstvima. Glavna razlika u odnosu na prethodna dva stila nije samo u konceptu programa i njegovom izvođenju već i u konceptu programskih varijabli. Za razliku od proceduralnih jezika, kod funkcijskih i logičkih programa ne postoje naredbe s eksplicitnim pridjeljivanjem pa se stoga smatraju netradicionalnim stilovima.

Kod **objektno-orijentiranih** paradigmi, program opisuje strukturu i ponašanje tzv. objekata i klasa objekata. Objekt enkapsulira pasivne podatke i aktivne operacije koje se provode nad tim podacima. Podaci određuju njegovo stanje, a skup metoda opisuje ponašanje objekta. Klase predstavljaju set objekata koji imaju istu strukturu i isto ponašanje. U pravilu opis klasa uključuje naslijeđenu hijerarhiju uključujući polimorfizam. Izvođenje objektno orijentiranog programa može se opisati kao razmjena poruka između objekata čime se stanja tih objekata modificiraju.

Objektno orijentirana paradigma je najapstraktnija budući da njezina osnovna ideja može jednostavno biti kombinirana s principima i programskim tehnikama ostalih stilova. U pravilu, objektno orijentirana metoda može biti predstavljena kao procedura ili funkcija, gdje slanje poruka može biti predstavljen kao poziv funkcija ili procedura.

2.2. Modeli životnog ciklusa razvoja

[1]

Model životnog ciklusa proizvoda definira kako bi se aktivnosti trebale provoditi u izradi proizvoda – određuje poredak kojim se obavljaju aktivnosti na projektu (dizajn, izrada prototipova, oblikovanje, implementacija, testiranje, revizija, i dr.).

Prvi dokumentirani model je tzv. model **vodopada** koji se po prvi put spominje 50ih godina 20 st., a kojeg je Winston W. Royce 1970. g. popularizirao u jednom članku iako tad nije upotrijebio naziv vodopad. Model vodopada sastoji se od nekoliko osnovnih faza, a u originalnom modelu to su: specifikacija zahtjeva, dizajn, konstrukcija, integracija, testiranje i detaljna analiza (eng. *debugging*), instalacija i održavanje. Na kraju svake faze provodi se kontrola kako bi se utvrdilo da li je projekt spreman za sljedeću fazu, a ako nije zadržava se na trenutnoj. Iako najstariji, ovaj model i danas se koristi, a služi i kao osnovica mnogim drugim modelima. Koristan je kod detaljno i stabilno definiranih proizvoda te pri radu s dobro poznatom tehnologijom, dok se nedostaci otkrivaju kod slabo definiranih i promjenjivih projekata kao i kod potrebe za brzim razvojem.

Kasnije je iz modela vodopada nastalo nekoliko varijacija. Jedna od prvih varijacija izvedena iz modela vodopada jest **inkrementalni** model. Tu se radi o kružnom modelu razvoja koji je razvijen kako bi otklonio nedostatke svog prethodnika. Osnovna ideja ovog modela jest tzv. inkrementalni razvoj sustava gdje se na početku razvije jednostavni sustav koji zadovoljava samo dio specifikacija. Naknadno se iterativno povećava skup ciljanih zahtjeva sve dok sustav ne bude u potpunosti završen. Drugi naziv za opisani način rada je i model postupne isporuke (eng. *Staged Delivery*).

Korištenje prototipa za razvoj ili tzv. **evolucijsko prototipiranje** također koristi inkrementalno povećavanje funkcionalnosti kod traženog sustava, ali za razliku od prethodno opisanog modela nije poznata konačna „slika“ ciljanog proizvoda već se nakon svakog završenog prototipa specificiraju potrebne funkcionalnosti sljedeće verzije sustava. Takav način rada pogodan je kad se zahtjevi brzo mijenjaju, ali se zbog toga ne može dobro procijeniti trajanje i cijena projekta. Ovaj model zasniva se na „*Do it twice*“ principu koji je 1975. predstavio F. Brooks.

Kombinacija prethodna dva modela naziva se modelom **evolucijskih isporuka**, a zasniva se na postupnoj izgradnji sustava gdje se u svakoj iteraciji dodaju određene funkcionalnosti, ali se pri tome u sam postupak uključuju i klijenti koji kontroliraju pojedine inačice sustava te „u hodu“ mijenjaju tj. prilagođavaju specificirane funkcionalnosti.

Još jedan model zasnovan na iteracijama je i **spiralni** model kojeg je 1988. B.Boehm predstavio u jednom članku. Ovaj model sličan je prethodno opisanim modelima zasnovanim na iteracijama i inkrementalnom povećavanju funkcionalnosti sustava, a osnovna je razlika u tome što je svaka iteracija usredotočena na rizike. Koncept spiralnog razvoja zasniva se na tome da se za svaku iteraciju istraže rizici koji se mogu pojaviti, izrade planovi za njihovo rješavanje te se nastavlja sa sljedećom iteracijom. Osnovne 4 faze ovog procesa bile bi određivanje ciljeva, identifikacija i rješavanje rizika, razvoj i testiranje te na kraju planiranje sljedeće iteracije. Iako je ovaj model prilično složen, zbog čega se koristi samo na velikim projektima, njegovo korištenje olakšava rano prepoznavanje i rješavanje mogućih rizika.

Od ostalih modela potrebno je spomenuti i model **razvoja do roka** (eng. *Design to Schedule*) te model **razvoja prema korištenim alatima** (eng. *Design to tools*). Prvi planira razvoj proizvoda samo do određenog roka, što znači da tad projekt završava, ali da ne mora imati implementirane sve funkcionalnosti budući da se poredak razvijanih funkcionalnosti kreće od važnijih prema manje važnim. Drugi model pretpostavlja razvoj proizvoda samo s onim funkcionalnostima koje su podržane od strane korištenih tj. raspoloživih razvojnih alata.

Na temelju procesa koje je Microsoft koristio pri izradi Internet Explorer preglednika i procesa koje je Netscape koristio pri izradi svog Netscape Communicator preglednika, 1996. godine David Yoffie (Harvard) i Michael Cusumano (MIT) razvili su **sinkroniziraj-i-stabiliziraj** (eng. *sync-and-stabilize*) model. Cijeli projekt dijeli se u nekoliko faza (3-4) pri čemu svaka dodaje nove funkcionalnosti, ali prva daje najosnovnije. U tom modelu timovi rade paralelno na individualnim aplikacijskim modulima, periodički sinkronizirajući svoj programski kod s ostalim timovima (testiranjem i analizom pogrešaka – eng. *debug*) te stabilizirajući (uklanjanje pogrešaka i završavanje pojedinih verzija) tijekom cijelog razvoja. Na kraju je cilj izdati stabilan sustav. Budući da navedeni model omogućava promjene u svakoj točci razvoja, on može biti vrlo fleksibilan te omogućavati brže odgovore na zahtjeve tržišta, osobito kad se usporedi s sekvencijalnim modelom vodopada.

Kao odgovor na prve strukturirane modele razvoja aplikacija (vodopad modeli i sl), koje su imale taj nedostatak što bi razvoj ponekad trajao toliko dugo te bi se početni zahtjevi previše promijenili učinivši dobiveni sustav zastarjelim ili beskorisnim, 1991. godine predstavljen je **RAD – Rapid Application Development**. Ta metoda razvijena je u IBM-u, a navedene godine ju je James Martin predstavio u knjizi istoimenog naziva, kod koje je ključ u brzom razvoju i isporuci visoko kvalitetnog sustava pri relativno niskim investicijskim troškovima. Ovo je jedna od prvih agilnih metoda razvoja, koja je zasnovana na iterativnom razvoju te izgradnji prototipova kako bi se definirali korisnički zahtjevi i dizajnirao konačni sustav. Korištenjem strukturiranih tehnika, u razvoju se prvo izrađuje početni podatkovni i poslovni model na temelju kojeg se potom izrađuje(u) prototip(ovi). Dobiveni prototipovi koriste se za provjeru korisnikovih zahtjeva te za reformuliranje podataka i procesnih modela. Opisani postupak se ponavlja dok se ne dobije konačna kombinacija poslovnih zahtjeva i tehničke specifikacije dizajna budućeg sustava.

Iako bi se RAD u suštini mogao poistovjetiti s evolucijskim prototipiranjem, osnovna razlika je u tome što je kod RAD metode naglasak na agilnosti i brzom završetku sustava, dok je kod evolucijskog protipiranja naglasak na postupnom dodavanju funkcionalnosti počevši od jednostavnog prema složenom pri čemu brzina nije ključna, već samo kvaliteta.

Još jedna od starijih agilnih metoda je i **Scrum**, koja je pod tim imenom službeno predstavljena 1995. godine, iako su još 1986. godine Hirotaka Takeuchi i Ikujiro Nonaka opisali novi holistički pristup koji je ubrzavao brzinu i fleksibilnost u komercijalnom razvoju novih proizvoda te koji je na posljetku predstavljao temelje nove metodologije. Scrum je model razvoja koji uključuje skup preporuka i predefiniranih uloga. U ovom novom modelu faze se uvelike preklapaju, a cijeli projekt se izvodi od strane multi-funkcionalnog tima kroz različite faze. Tim izrađuje proizvod u tzv. sprintovima tijekom kojih se implementiraju točno određene funkcionalnosti te se tijekom trajanja pojedinog sprinta (2-4 tjedan) postavljeni zahtjevi ne mogu mijenjati. Na kraju sprinta prezentira se korištenje razvijenih funkcionalnosti, od strane klijenta postavljaju se novi zahtjevi te tim procjenjuje koliko će im trebati vremena za njihovo ispunjenje tijekom novog sprinta. Ovaj model donosi dugoročne koristi jer omogućava kreiranje samostalnog tima potičući komunikaciju između članova te razmjenu znanja iz različitih područja koji su

uključeni u projekt. Kao i kod brojnih drugih modela, tako je i kod ovog modela od presudne važnosti uključenost krajnjih korisnika i njihovih zahtjeva. Ali važno je i prihvaćanje učestalih korisničkih promjena zahtjeva zbog nemogućnosti potpunog definiranja problematike projektnog zadatka. Stoga se Scrum fokusira na razvijanje sposobnosti članova tima kako bi mogli brzo odgovoriti na rastuće i promjenjive korisničke zahtjeve.

3. Novi trendovi razvoja aplikacija

3.1. Agilni razvoj

[8][9][10][11][12][14]

Agilni razvoj aplikacija predstavlja grupu različitih razvojnih metoda koje su sve zasnovane na sličnim principima. U pravilu, agilne metode promiču procese upravljanja projektima koji potiču ispitivanje i prilagodbu, timski rad, organizaciju po potrebi i po vlastitoj odgovornosti članova, razvoj u potpunosti prilagođen korisničkim potrebama i ciljevima, korištenje inženjerskih najboljih praksi koje omogućavaju brzu isporuku visoko kvalitetnih programskih proizvoda.

Do pojma agilnog razvoja došlo je tek sredinom 90ih godina kao odgovor na „teške“ metode kao što je vodopad, koji je viđen kao previše birokratski, spor, i neusklađen s načinom na koji programeri rade. Zbog toga su se agilne metode na početku nazivale i „laganim“ metodama. 2001. godine formirana je Agile Allianze, neprofitna organizacija koja promovira agilni razvoj. Iste godine objavljen je i tzv. Agilni Manifest sa sljedećih 12 naputaka:

- Najveći prioritet je zadovoljiti klijenta pomoću ranih i kontinuiranih isporuka korisnih aplikacija.
- Prihvaćati promjene u zahtjevima čak i u kasnim fazama razvoja. Promjene su potrebne za potrebe postizanja konkurentnih prednosti.
- Isporučivati funkcionalne aplikacije često, u razmacima od nekoliko tjedana do nekoliko mjeseci, s prioritetom da se razmak smanji.
- Menadžeri moraju surađivati i raditi zajedno s programerima svakodnevno tijekom projekta.
- Projekte treba graditi na motiviranim pojedincima te im je potrebno dati povjerenje i potrebno okruženje.
- Najefikasniji i najefektivniji metoda prijenosa informacija prema i unutar razvojnog tima je osobno, licem u lice.
- Primarna mjera napretka su funkcionalne aplikacije.
- Agilni proces promovira održivi razvoj. Sponzori (naručitelji, klijenti), programeri i korisnici bi trebali održavati konstantan tempo bez prekida.
- Kontinuirana pažnja na tehničko savršenstvo i dobar dizajn unaprjeđuju agilnost.
- Jednostavnost, umjetnost maksimiziranja količine posla koji nije obavljen, je ključna.
- Najbolje arhitekture, zahtjevi i dizajn, proizlaze iz samoorganiziranih tipova.
- Redovno, tim razmišlja kako postati može postati više efektivan, te u skladu s time prilagođava svoje ponašanje.

Postoji više specifičnih agilnih metoda razvoja aplikacija. U pravilu, one definiraju kako se stvari trebaju raditi s inkrementalnim poboljšanjima i s minimalnim planiranjem radije nego s dugotrajnim planiranjem. Iteracije su kratki vremenski odsječci koji uobičajeno traju od jednog tjedna do mjesec dana. Na svakoj iteraciji radi tim kroz cijeli tipični razvojni ciklus (planiranje, analiza zahtjeva, dizajn, implementacija, testiranje i prihvaćanje od strane korisnika). Time se smanjuje rizik, a i projekt se može brže prilagoditi promjenama. Dokumentacija se izrađuje u skladu sa zahtjevima korisnika. Svaka iteracija ne mora dodati velik set funkcionalnosti koji bi jamčio izlazak proizvoda na tržište, već je potrebno da završi izdanom verzijom s minimumom grešaka. Veći broj iteracija može biti potreban da bi se izdala verzija proizvoda s novim funkcionalnostima.

Timovi su obično više-funkcijski, samoorganizirani, bez zamaranja s postojećom hijerarhijom ili pozicijama članova tima. Članovi tima obično preuzimaju odgovornost za isporuku funkcionalnosti pojedine iteracije. Oni sami za sebe odlučuju što će raditi tijekom pojedine iteracije. Budući da je preporuka da komunikacija bude licem u lice, većina timova je smještena u jednom uredu koji to omogućuje, a isto tako i njihova veličina je relativno mala (5-9 članova) što olakšava komunikaciju i suradnju. U slučaju kad se radi o većim razvojnim projektima, formira se veći broj timova koji svi rade prema zajedničkom cilju, pa je stoga

potrebno koordinirati prioritete. Komunikacija može biti i formalna dnevna na kojoj može sudjelovati i predstavnik klijenta, ili druge zainteresirane strane. Na tom dnevnom sastanku, članovi u kratko iznesu što su radili jučer, što planiraju tog dana, te koje su specifičnosti. Tim oblikom komunikacije licem u lice sprečava se da problemi ostanu skriveni. Svrha predstavnika klijenta, koji treba biti dodijeljen timu, je da nadzire napredak te da procjenjuje prioritete u skladu s optimiranjem stope povratka na investicije, te da osigura usklađenost s potrebama klijenta i ciljevima organizacije.

Uz neke u nastavku su navedene neke od novijih metoda agilnog programiranja kao što je ekstremno programiranje (eng. *Extreme Programming*), agilni objedinjeni proces (eng. *Agile Unified Process*), razvoj prema funkcionalnostima (eng. *Features Driven Development*), naslonjeni razvoj (eng. *Lean Software Development*), dok je od „starijih" metoda potrebno spomenuti Scrum, RAD (eng. *Rapid Application Development*) te na kraju DSDM (eng. *Dynamic Systems Development Methodology*), za koju se može pronaći kako je to upravo izvorna agilna metoda.

Prednosti agilnog programiranja dolaze do izražaja kad se usporedi s tradicionalnim metodama:

- skraćeni razvoj sustava na 75-80%, kraće vrijeme isporuke,
- veća stabilnost i kvaliteta sustava zahvaljujući ranom dobivanju povratnih informacija od korisnika,
- bolja iskorištenost ljudi,
- veća fleksibilnost na promjene (korisnički zahtjevi i planovi menadžmenta).

Nedostaci agilnog programiranja proizlaze iz zahtjeva koji trebaju biti ispunjeni za uspješnu implementaciju agilnog programiranja, a ti nedostaci su sljedeći:

- ne funkcionira dobro s velikim timovima ljudi,
- ne funkcionira dobro kad je tim razdijeljen na više udaljenih lokacija, ili kad je slaba komunikacija,
- zahtjeva vrlo kvalitetne i iskusne članove tima,
- zahtjeva visoku razinu uključenosti krajnjih korisnika što nije uvijek moguće,
- iako uvelike doprinose kvaliteti konačnog proizvoda, troškovi testiranja mogu biti visoki,
- na početku ne postoji jasna slika konačnog proizvoda.

3.2. Ekstremno programiranje (XP)

[2][3]

Na temelju iskustava rada na projektu Chrysler Comprehensive Compensation 1999. godine Kent Beck je objavio knjigu „Extreme Programming Explained". Po toj knjizi, cilj ekstremnog programiranja je smanjiti troškove promjena zahtjeva, koji su pri tradicionalnim metodama razvoja relativno visoki. Ekstremno programiranje (eng. XP – *Extreme Programming*) postoji upravo zbog sustava koji se svakih nekoliko mjeseci trebaju mijenjati. To znači da je cilj isporučivati programe onda kad su potrebni i u obliku u kojem su potrebni.

Ekstremno programiranje stavlja naglasak na timski rad pa su tako i menadžeri, i programeri i klijenti dio tima. Programeri su ovlašteni samostalno odgovarati na promjene u korisničkim zahtjevima čak i u završnima fazama razvoja. Vezano uz članove tima nalazi se i jedno od ograničenja ekstremnog programiranja, a to je da je primjenjivo samo na projekte s manjim brojem zaposlenika. Idealna grupa kreće se od 2 do 12 zaposlenika, iako su prijavljeni uspjesi i kod projekata s 30 članova. U pravilu, kod projekata s dinamičnom promjenom zahtjeva ili visokim rizicima, manja grupa koja primjenjuje principe ekstremnog programiranja može biti učinkovitija nego veliki tim.

Originalno, ekstremno programiranje je osmišljeno za potrebe projekata s visokim rizicima. Npr. ako klijent zahtjeva novi sustav do određenog datuma, rizik je visok, ali ako je izrada tog sustava novi izazov za projektni tim, rizik je tim i veći, a najveći je ako je izrada tog sustava izazov ne samo za tim već i za cijelu industriju. Korištenjem ekstremnog programiranja smanjuju se rizici te povećava vjerojatnost uspjeha projekta.

U nastavku slijede osnovna pravila i preporuke podijeljene u 4 glavna procesa razvoja sustava: planiranje, dizajn, programiranje i testiranje.

3.2.1. XP planiranje

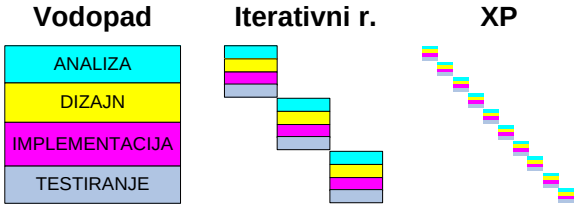
Planiranje se odnosi na dva glavna pitanja u procesu razvoja: što će biti napravljano do određenog dana te određivanje što treba sljedeće napraviti. Pri tome je naglasak na držanju osnovnog pravca razvoja, a ne na detaljnom predviđanju potreba korisnika i točno trajanje.

Kod planiranja se koriste **korisničke priče** (eng. *user stories*), koje su slične korisničkim scenarijima (eng. *user scenarios*) ali ne specificiraju samo sučelja sustava. Služe za potrebe planiranja izlazaka pojedinih inačica programa. Koriste se umjesto dugih dokumenata sa zahtjevima, a u njima korisnici opisuju što bi sustav trebao raditi za njih u par rečenica (korisnikova, a ne tehnička terminologija). Za svaku korisnikovu priču potrebno je izraditi najmanje jedan test kojim će se potvrditi ispunjenost korisnikove priče. Svrha ovih kratkih priča je samo da pruže dovoljno detalja kako bi se procijenilo trajanje implementacije korisnikovih zahtjeva. Prije same implementacije, programeri moraju dobiti detaljnije specifikacije. Svaka korisnička priča trebala bi prema procjeni trajati 3 tjedna, a ako traje dulje, treba ju razdijeliti na druge priče. Na kraju je potrebno imati 80 korisničkih priča (plus/minus 20) za planiranje.

Na zajedničkom sastanku kreira se **raspored verzija sustava**. Taj raspored služi da bi se kreirali planovi za svaku pojedinačnu iteraciju. Osnova za raspored su korisničke priče čija se implementacija mjeri u „idealnim“ tjednima (unutar kojih programeri ne dobivaju druge zadatke). Raspored definira koje korisničke priče će biti implementirane u sklopu koje verzije sustava te na koji datum će ta verzija biti izdana.

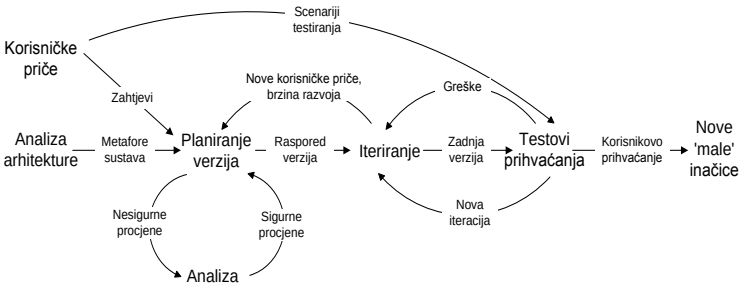
Preporuka je **često izdavanje verzija** koje ne trebaju biti značajno poboljšane. Važno je otkriti funkcionalnosti od kojih korisnik ima koristi za poslovanje te ih isporučiti čim prije u nekoj verziji te staviti u pogon. Time se dosta rano mogu dobiti povratni podaci o sustavu, a samim time ima više vremena za popravke.

Sustav se razvija u **iteracijama** trajanja od 1 do 3 tjedna pri čemu one trebaju biti približno iste duljine. Sve programerske aktivnosti nije potrebno planirati na samom početku projekta već se pojedine aktivnosti planiraju na početku pojedine iteracije. To je tzv. **JIT** (eng. *Just In Time*) **planiranje iteracija** koje omogućava brzo praćenje promjena korisničkih zahtjeva. U pojedinu iteraciju stavljaju se oni korisnički slučajevi koji se podudaraju s rasporedom verzija te koje su korisniku najvažnije da ih čim prije dobije. Svaki programer procjenjuje koliko mu idealnih radnih dana treba za ispunjenje kojeg zadatka, a ako je procjena veća od 3 radna dana, zadatak je potrebno raščlaniti u manje zadatke. U slučaju da unutar neke iteracije nije moguće izvršiti sve zadatke, rokovi se ne smiju pomicati te stoga treba izbaciti neke od zadataka. Stoga na kraju cijeli razvoj završi s velikim brojem iteracija kao što je prikazano na sljedećoj slici.



Slika 1. Usporedba XP-a s modelom vodopada i iterativnog razvoja

Potrebno je mjeriti i **brzinu razvoja projekta**. Pri tome se procjenjuje trajanje ispunjenja korisničkih priča u pojedinim iteracijama na temelju prijašnjih iteracija. Na temelju dobivene brzine razvoja, moguće je smanjiti ili povećati opseg pojedinih iteracija u skladu s važnosti zadataka za korisnika.



Slika 2. Osnovna shema XP procesa

Kako bi se izbjegla nepotrebna kašnjenja važno je **mijenjati zaposlenike** i njihove poslove. Jer ako samo jedna osoba zna raditi određene zadatke, tada ako ona ode ili jednostavno ima važnije zadatke za odraditi, napredovanje projekta će se znatno usporiti. U tu svrhu koristi se i programiranje u parovima, ali generalno važno je poticati zaposlenike da rade na novim stvarima.

Svaki dan potrebno je početi **brzim sastankom** koji treba odraditi stojeći da se ne bi nepotrebno gubilo vrijeme. Na njemu treba komunicirati o problemima, rješenjima te fokusirati tim.

Kad XP proces prestane funkcionirati potrebno ga je **„popraviti“**. Za očekivati je da XP treba prilagoditi konkretnim potrebama projekta i ljudi pa stoga treba komunicirati i o kvalitetama i nedostacima XP metode razvoja.

3.2.2. XP dizajniranje

Jednostavnost je jedan od glavnih ciljeva prilikom dizajniranja sustava jer je uz jednostavan dizajn projekt uvijek moguće brže završiti nego uz složeni. Stoga je preporuka da se uvijek radi najjednostavnije moguće jer je brže i jeftinije. U tu svrhu nikad se ne bi trebale dodavati nove funkcionalnosti prije za to predviđenog roka. Naravno, održavanje dizajna jednostavnim vrlo je težak posao.

Prilikom dizajniranja potrebno je koristiti **prikladne metafore** za konzistentno definiranje naziva objekata i klasa. To može olakšati razumijevanje sustava kao cjeline ili prilikom potrebe ponovnog korištenja već izrađenog programskog koda.

Za potrebe dizajniranje sustava poželjno je koristiti **CRC (eng. *Class Responsibility Collaborator*)** kartice. One se koriste za dizajniranje objektno orijentiranih programa i pomažu shvatiti koje su programske klase potrebne te kako će one međusobno djelovati. Prednost im je što ih je lako mijenjati, te pomažu izradi jednostavnijeg dizajna fokusiranjem samo na važne detalje što uključuje i definiranje samo osnovnih nadležnosti pojedinih kartica. Također, njihovim korištenjem ljudi se mogu udaljiti od proceduralnog i bolje prihvatiti objektno orijentirano razmišljanje.

U slučaju kompliciranih tehničkih ili dizajnerskih problema potrebno je kreirati **potencijalna zamjenska rješenja**. Cilj tomu je smanjiti rizik mogućih tehničkih problema ili povećati pouzdanost procjena korisničkih priča. I u slučajevima kad određeni tehnički problem zaprijeti normalnom razvoju sustava, potrebno je odrediti par programera koji trebaju na tom problemu raditi tjedan dva dok ne smanje taj potencijalni rizik.

Prilikom dizajna potrebno je paziti da se **ne dodaju nepotrebne funkcionalnosti**. Samo 10% tako ekstra dodanih funkcionalnosti će se ikad koristiti, pa stoga je 90% utrošenog vremena čisti gubitak. Prilikom dizajniranja i razvoja svi se nalaze u iskušenju da dodaju funkcionalnosti rano jer im je jasno kako ih dodati i jer bi sustav očito bio bolji s njima. Čini se da ih je brže dodati ranije. Stoga je potrebno konstantno podsjećati kako te funkcionalnosti ipak nisu potrebne i kako će one na kraju samo usporiti cjelokupni razvoj pa se stoga potrebno koncentrirati samo na one stvari koje su planirane.

I na kraju kod dizajna je potrebno primjenjivati nešto što će se u ovom tekstu nazvati **redizajn**, dok je engleski termin *refactoring*. Pod tim pojmom misli se na odbacivanje nepotrebnih funkcionalnosti, renoviranje zastarjelih dizajna, te uklanjanje svih redundancija. To je kod programera dosta teško za provesti budući da oni nisu skloni uklanjanju programa koji sasvim dobro funkcioniraju, a osobito ako su na njih potrošili puno vremena. Ali to je potrebno kako bi programski kod bio čim jednostavniji i lakši za razumjeti, modificirati te proširivati.

3.2.3. XP programiranje

Tijekom razvoja sustava važan preduvjet je **dostupnost krajnjeg korisnika** i to ne samo kako bi pomagao razvojnom timu već i da bi bio član tog razvojnog tima. Naime, budući da sve faze XP razvoja zahtijevaju komuniciranje s korisnicima, najbolje osobno licem u lice, najjednostavnije je jednog od korisnika pridružiti razvojnom timu. Budući da se tu radi o dugotrajnom procesu, potrebno je paziti da kao pridruženi član tima ne bude poslan neki novak, što će korisnikova organizacija prvo pokušati, već stručnjak za svoje područje. Korisnik na kraju treba biti uključen u sve faze, od kreiranja korisničkih priča, planiranja verzija i uključenih funkcionalnosti, pa do testiranja i prihvaćanja sustava.

Prilikom samog programiranja potrebno je držati se određenih **standarda programiranja**. To pomaže da programski kod ostane konzistentan i jednostavan za čitanje te eventualne modifikacije.

Preporuka je i da se **unaprijed kreiraju osnovni testovi** za provjeru funkcionalnosti, prije same izrade programskog koda. Time se ne produžuje izrada samog osnovnog programskog koda, a kasnije nije potrebno trošiti dodatno vrijeme na izradu testova. Također, kreiranje osnovnih testova pomaže programerima da shvate što je točno potrebno napraviti, programski zahtjevi postaju jasniji, a dobivaju i brže povratne informacije o uspješnosti razvoja pojedinih modula. Čak je jednostavnije i odrediti kad je programer završio sa svim potrebnim

funkcionalnostima – kad su svi osnovni testove zadovoljeni.

Još jedna od preporuka, koja programerima na početku može biti čudna, jest **programiranje u parovima**. To znači da dvije osobe (programera) sjede zajedno pred monitorom pri čemu jedna osoba piše programski kod dok ga druga kontrolira te razmišlja kakvi su sve efekti novog koda mogući na ostale dijelove sustava. Uloge se pri tome mogu mijenjati. Ta preporuka trebala bi se poštivati prilikom izrade programskog koda cijelog sustava. Na kraju bi trebala rezultirati mnogo višom razinom kvalitete sustava te jednakom količinom funkcionalnosti kao da su programeri radili odvojeno.

Vrlo je važno da se za uključivanje novog programskog koda u cjelokupni sustav koristi **sekvencijalna integracija**. To znači da se zabranjuju slučajevi u kojima bi različiti programeri istodobno uključivali svoj programski kod u sustav i testirali ga. Očito je da u sklopu takvog oblika integracije mogu nastati brojni problemi i pogreške, a dodatan je problem što nije moguće odrediti ni čistu razdiobu verzija koje u sebi imaju određeni skup novih funkcionalnosti. Stoga je potreban određena procedura zaključavanja, npr. korištenje posebnog računala (laptopa) namijenjenog samo za integraciju koji istodobno može biti samo kod jednog programera.

Što se tiče integracije programskog koda, tu postoji još jedno pravilo, a ono je da se **programski kod često integrira**. Preporuka je da se programski kod uključuje u sustav svakih nekoliko sata te da se ne čeka duže od jednog dana, ali po pravilu sekvencijalne integracije. Time i drugi programeri mogu koristiti već izrađene komponente čak ako komunikacija nije na prikladnoj razini.

Važno je da tijekom projekta postoji tzv. **zajedničko vlasništvo nad programskim kodom**, tj. svi programeri mogu slobodno mijenjati gotovi programski kod, koji nisu nužno oni sami napisali, kako bi dodali nove funkcionalnosti ili ispravili pogreške. Pri tome moraju paziti da novo napisani programski kod zadovoljava prethodno definirane osnovne testove. Prednost ovakvog rada je što u razvoju projekta ne postoje uska grla u obliku autora pojedinih dijelova sustava, ili glavnog arhitekta koji mora sve akcije odobriti. Umjesto takvih ograničavajućih uvjeta primjenjuje se zajedničko pravo mijenjanja sustava pri čemu su svi odgovorni za najbolju moguću implementaciju sustava.

Sve **optimizacije** treba ostaviti da budu **završne aktivnosti**. Nije potrebno nagađati što bi sve moglo postati usko grlo u sustavu. Na kraju kad je sustav gotov takve stvari treba mjeriti te tek onda optimizirati. Tijekom razvoja važnije je brinuti da sustav radi ispravno nego brzo.

Zaposlenici na projektu bi trebali **izbjegavati prekovremeni rad**. Prekovremeni rad uništava motivaciju i duh među članovima tima. Projekti koji zahtijevaju prekovremeni rad na kraju tako i tako ne završe u roku. Stoga zaposlenici trebaju biti angažirani „samo“ 40 sati tjedno. Isto tako nije uspješno ni dodavati nove ljude sa ciljem da se izbjegne kašnjenje projekta. Umjesto svega toga potrebno je promijeniti raspored izlazaka verzija sustava.

3.2.4. XP testiranje

[6] [7][13]

Jedan od temelja ekstremnog programiranja je korištenje **osnovnih testova** (eng. *unit tests*) za provjeru ispravnosti implementiranih funkcija. Za samo testiranje potrebno je ili uzeti (npr. JUnit za Java jezik) ili razviti vlastito okruženje pomoću kojeg će se moći kreirati automatizirani testovi. Okruženje pomaže formalizirati zahtjeve, pojednostaviti arhitekturu, te napisati, kontrolirati, integrirati i naravno testirati programski kod. Pomoću takvog okruženja potrebno je testirati sve klase u sustavu. Također, same testove je potrebno kreirati prije razvoja samog programskog koda sustava. Stoga razvijeni programski kod ne može biti integriran u cjelokupni sustav ako ga ne prate i popratni testovi.

Projektnim timovima se često čini kako nemaju vremena unaprijed izraditi testove jer su rokovi prekratki. Ali ako tijekom projekta već postoje pripremljeni testovi, pogreške se mogu na vrijeme uočiti i njihovo uklanjanje može biti mnogo „bezbolnije“. I kad bi se razvoj testova ostavio za zadnju fazu, sustav bi bio nekvalitetan i cjelokupni razvoj bi se samo produžio Stoga je vrlo važno da sav **programski kod zadovolji testove prije integracije** u sustav.

A u slučajevima kad se u programskom kodu otkrije neki tip pogreške (eng. *bug*), potrebno je kreirati **test(ove) za pogrešku** kako se ne bi kasnije u drugim modulima ponovila.

Iz definiranih korisničkih priča kreiraju se i **testovi za prihvaćanje** (eng. *acceptance tests*) sustava. Klijent specificira scenarije koje je potrebno testirati na temelju definiranih

korisničkih priča, a za jednu priču moguće je definirati jedan ili više testova kako bi se potvrdila ispravnost implementirane funkcionalnosti. Svaki takav test reprezentira određene očekivane rezultate sustava. Klijenti su odgovori za prihvaćanje ispravnosti testova prihvaćanja te kontrolu dobivenih rezultata testiranja kako bi odredili koji neispunjeni testovi imaju najviši prioritet.

Vezano uz testiranje kao osnovu razvoja programskih paketa, sve češće se spominje i princip **razvoja testiranjem** (eng. ***Test-Driven Development***). O tome je Kent Beck napisao i knjigu 2003. g. Test-Driven Development by Example, a principi su isti kao i kod testiranja kod ekstremnog programiranja. Prije samog razvoja, a na temelju definiranih funkcionalnosti, izrađuju se osnovni testovi koje razvijani sustav mora zadovoljiti. Nakon svake integracije provode se testiranja, a bitno je da svi testovi budu uspješno zadovoljeni. Osnovna premisa programiranja zasnovanog prvenstveno na testiranju jest da se programira samo programski kod potreban za zadovoljenje testova što rezultira jednostavnijim dizajnom.

Zabilježeno je da korištenje unaprijed definiranih testova povećava produktivnost pa stoga ne čudi primjena unaprijed definiranih testovima u većini agilnih metodologija. Ipak, važno je napomenuti kako neki ipak smatraju da to nije dovoljno dokazno.

3.2.5. Prednosti i nedostaci XP-a

[4][5]

Prednosti ekstremnog programiranja različito se očituju kod različitih interesnih grupa:

- **Programerima** XP olakšava fokusiranje na razvoj sustava, a ne na nepotrebnu papirologiju i sastanke. Zahvaljujući čestim izlascima novih inačica, dobiva se bolji osjećaj o postignućima. Također, zbog 40 satnog radnog tjedna, programeri imaju priliku u normalno vrijeme odlaziti s posla. Isto tako, zbog poticanja timskog rada pospješuju se i socijalni odnosi i komunikacija.
- **Korisnicima** ekstremno programiranje omogućava brže dobivanje završnog proizvoda, koji je zahvaljujući sustavnom testiranju i kvalitetniji. Pojedine komponente proizvoda mogu se i prije zadnje inačice staviti u rad. Također, ukoliko su potrebne, promjene u zahtjevima se mogu lako tražiti bez prigovora. Pri svemu tome korisnik se može pouzdati u realno zadani raspored izlazaka inačica.
- **Menadžmentu** XP omogućava dobivanje gotovog sustava uz manje troškove. Također, manji su i rizici.

Osnovni **nedostatak** XP-a je u tome što je težak. Težak je za programere koji trebaju prihvatiti načela XP-a te imati puno discipline za njihovo ispunjavanje. Težak je za korisnike koji moraju aktivno sudjelovati u razvoju cijelog sustava. A i menadžment može teško prihvatiti promjenu procesa rada. Stoga XP nije preporučljiv ukoliko ga sve tri grupe neće podržati i prihvatiti.

Osim toga XP nije prikladan za kompleksne projekte koji zahtijevaju velik broj sudionika. Tu je osnovni problem u nejasnom dizajnu finalnog proizvoda pa projekt koji se zasniva na promjenama može trajati unedogled.

Jedan od nedostataka je i manjak dokumentacije. Stoga je mala vjerojatnost iskorištavanja prijašnjih komponenti prilikom rada na novim projektima. Osobito jer se razvijene metode nikako ili samo ukratko opisuju.

Iako se u literaturi uglavnom navodi kako XP podiže kvalitetu finalnog proizvoda, XP ne predviđa nikakav plan za podizanje kvalitete sustava u obliku završnih testiranja pa stoga tradicionalnije metode razvoja mogu generirati kvalitetnije završne proizvode.

3.3. Objedinjeni proces

[23][24]

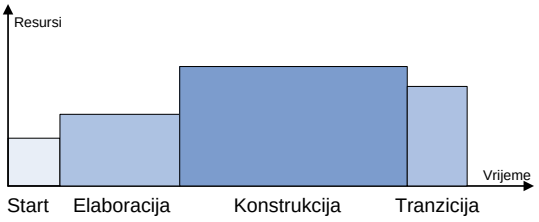
Objedinjeni proces (eng. *Unified Process*) je iterativni i inkrementalni razvojni proces koji nije samo proces već i jedno razvojno okruženje te ga je potrebno prilagođavati za specifične organizacije i projekte. To je ujedno i opći model koji uključuje druge specifične modele razvoja. Prvi put se spominje 1999. godine u knjizi The Unified Software Development Process.

Vrlo važna komponenta ovog modela su korisnički slučajevi (scenariji) koji se koriste da bi se odredili funkcijski zahtjevi te da bi se definirao sadržaj pojedinih iteracija.

Unified Process dijeli projekt u četiri osnove faze:

- start,
- elaboracija,
- konstrukcija i

- tranzicija.



Slika 3. Četiri faze objedinjenog procesa

Start je najkraća od svih faza projekta. U slučajevima kad je ta faza preduga, to predstavlja upozorenje da je početna specifikacija preopširna, a što je u suprotnosti sa duhom Unified Process modela. Tipični cilj ove faze je određivanje projektnog plana tj.:

- obuhvata projekta kao i ograničavajućih uvjeta,
- korisničkih specifičnih slučajeva i zahtjeva koji su ključni za dizajniranje sustava,
- jedne ili više mogućih arhitektura,
- mogućih rizika te
- preliminarnog vremenskog plan projekta te procjenu troškova.

Po završetku te faze moraju biti zadovoljeni određeni uvjeti:

- isplativost za korisnike,
- razumijevanje zahtjeva,
- vjerodostojnost procjene troškova, rokova, prioriteta, rizika, i razvojnog procesa,
- složenost (širina i dubina) do tad razvijenih prototipova,
- uspostava osnove odrednice prema kojoj će se uspoređivati ostvareni nasuprot planiranih troškova.

Ako neki od navedenih kriterija nisu ispunjeni, ova faza se može ponoviti kako bi se redizajnirala i bolje zadovoljila kriterije. Ali moguće je i odustati od projekta.

Tijekom **elaboracije** projekta projektni tim bi trebao odrediti većinu korisnih i bitnih među svim postavljenim zahtjevima. Ipak, osnovni cilj elaboracije je ustanoviti faktore poznatih rizika te uspostaviti i provjeriti arhitekturu sustava. Specifični procesi koji su dio ove faze su kreiranje dijagrama korisničkih slučajeva, konceptualnih dijagrama, te dijagrama arhitekture. Potom se arhitektura provjerava parcijalnom implementacijom sustava koji uključuje osnovne, arhitektonski najvažnije komponente. To parcijalno pokazno rješenje treba demonstrirati kako će arhitektura podržavati ključne sistemske funkcionalnosti i osigurati ispravno ponašanje u smislu performansi, skalabilnosti i troškova. Uz navedeno, elaboracija treba isporučiti i plan (uključujući troškovnu i vremensku dimenziju) za fazu konstrukcije.

Elaboracija predstavlja početak formiranja projekta gdje se provodi analiza problema te arhitektura projekta dobiva svoj osnovni oblik. Uvjeti koji trebaju biti zadovoljeni su sljedeći:

- razvijen model korisničkih scenarija gdje su svi uključeni identificirani te je čak 80% korisničkih scenarija razvijeno,
- opis arhitekture programa u procesu razvoja programa te izvršne arhitekture kod koje su u obzir uzeti specifični korisnički slučajevi,
- revidirana lista poslovnih slučajeva i mogućih rizika,
- sastavljen plan razvoja za cjelokupni projekt,
- razvijeni prototipovi koji dokazivo smanjuju svaki identificirani tehnički rizik.

U slučaju kad projekt ne zadovoljava navedene uvjete, još postoji mogućnost za redizajn ili za odustajanje od projekta. Ali kad se s ove prijeđe na sljedeću fazu, projekt prelazi u visoko rizične operacije gdje su promjene mnogo složenije i teže pa i štetnije za projekt u cijelosti.

Konstrukcija je najopsežnija faza u cijelom projektu. Zasniva se na temeljima postavljenima u elaboraciji. Sistemske funkcionalnosti se implementiraju u serijama kratkih vremenskih iteracija. Svaka iteracija rezultira izvršnom verzijom proizvoda. Obično se napišu i potpuni korisnički slučajevi (scenariji) te svaki postaje početak nove iteracije.

Glavni cilj konstrukcije je razviti dizajnirani sustav. Korištenje iteracija biti će tim vjerojatnije gdje je projekt složeniji te je potrebno ili jednostavnije razdvojiti korisničke zahtjeve u praktične segmente koji mogu rezultirati pokaznim prototipovima. Na kraju ova faza rezultira prvom verzijom razvijenog programa – sustava.

Završna faza projekta je **tranzicija**. Tijekom te faze sustav se postavlja kod krajnjih korisnika. Povratne informacije korisnika dobivene iz inicijalne verzije mogu rezultirati u daljnjim usavršenim verzijama tijekom jedne ili više iteracija u sklopu tranzicije. Također, tranzicija uključuje i zamjenjivanje korištenog sustava te

obuku korisnika. Aktivnosti se odnose na obuku korisnika te testiranje kako sustav zadovoljava korisnička očekivanja. Također, sustav se ispituje kako zadovoljava razinu potrebne kvalitete zadane u početnoj fazi. I ako su na kraju svi zahtjevi ispunjeni, razvoj je završen.

U nastavku su navedene neke specifične implementacije ovog općenitog modela.

3.3.1. Racionalni objedinjeni proces

[17][21][24]

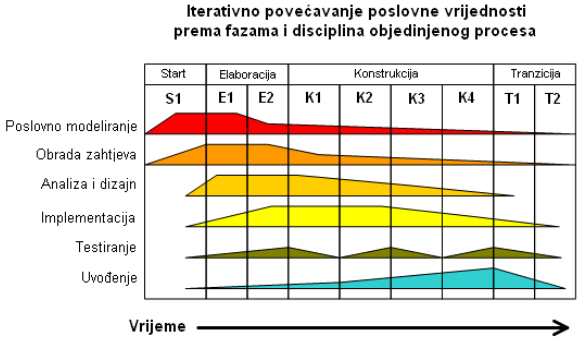
Racionalni objedinjeni proces (eng. ***Rational Unified Process***) razvijen je izvorno u Rational Software organizaciji te je pod navedenim imenom objavljen krajem 1998. godine. IBM je 2003. godine preuzeo navedenu organizaciju pa je tako danas RUP IBM-ov proizvod. Povećavanje

Autori ovog procesa fokusirali su se na dijagnosticiranje obilježja različitih propalih programerskih projekata kako bi otkrili glavne uzroke promašaja. Propast pojedinog projekta uzrokovana je kombinacijom različitih uzroka iako svaki projekt na kraju propadne na jedinstven način. RUP upravo predstavlja skup najboljih preporuka kako projekt ne bi propao.

Sam RUP zasniva se na objektno orijentiranom modelu korištenjem UML-a (*Unified Modeling Language*). RUP se sastoji od četiri osnovne faze (start, elaboracija, konstrukcija i tranzicija) čije su osnovne osobine prethodno opisane.

Posebnost RUP-a prema općenitom UP modelu, ali i svim ostalim specifičnim verzijama, jest u kategorizaciji procesa rada (tijeka poslova). Tako RUP razlikuje 6 osnovnih i 3 pomoćne aktivnosti - discipline. Osnovne su sljedeće:

- **Poslovno modeliranje** ima za cilj uspostaviti bolje razumijevanje između poslovnog i programskog inženjerstva. Programski inženjeri moraju bolje razumjeti strukturu i dinamiku klijentske organizacije, postojeće probleme i moguća poboljšanja.
- **Obrada zahtjeva** objašnjava kako izvući zahtjeve korisnika i uključenih te ih preoblikovati u detaljne specifikacije što bi novi sustava trebao biti i raditi.
- **Analiza i dizajn** trebaju pokazati kako će sustav biti realiziran. Cilj je izgraditi sustav koji će u sklopu određenog okruženja izvoditi definirane funkcije, zadovoljavati sve zahtjeve te biti lako promjenjiv u slučaju promjene funkcijskih zahtjeva.
- **Implementacija** ima zadatak definirati organizaciju programskog koda, programskih klasa i objekata, testirati razvijene komponente te integrirati pojedine rezultate u izvršni sustav.
- **Testiranje** služi da bi se provjerila interakcija među objektima, ispravnost integracije pojedinih komponenti, ispunjenje implementacije postavljenih zahtjeva. Također, služi i da bi se osiguralo identificiranje i uklanjanje problema prije razvoja programa, ali i tijekom razvoja. Testiranje se provodi u svim fazama kako bi se problemi identificirali čim ranije i time smanjili troškovi uklanjanja, a u tu svrhu različiti testovi su definirani za četiri dimenzije kvalitete: pouzdanost, funkcionalnost, aplikacijske i sistemske performanse**.**
- **Uvođenje** ima za cilj uspješno završiti verzije sustava te ih isporučiti krajnjim korisnicima. Ona pokriva različite aktivnosti izrada različitih verzija, kreiranje programskih paketa, distribucije i instalacije.



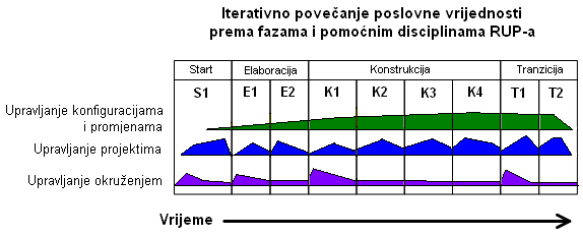
Slika 4. Porast poslovne vrijednosti po fazama i osnovnim disciplinama UP-a

Uz navedene osnovne, RUP definira i tri pomoćne aktivnosti:

- **Upravljanje konfiguracijama i promjenama** ima više zadataka. Potrebno je sistematizirati proizvod te brinuti da promjene budu dokumentirane (verzionirane) te povezana sa zavisnim elementima (modeli, dokumenti,...). Promjenama je potrebno pridjeljivati stanja (npr. novi, dozvoljeni, pridjeljeni, potpuni), te određene atribute kao što su

prioriteti, priroda promjene (npr. pogreška, promjena), uzrok promjene i sl. Ta stanja i atributi trebali bi biti pohranjeni u bazi podataka kako bi se mogli generirati različiti izvještaji.

- **Upravljanje projektima** odvija se na dvije razine. Prva je upravljanje na razini faza, a druga je upravljanje na razini iteracija. Ono obuhvaća samo ključna područja kao što je upravljanje rizicima, planiranje iteracija, praćenje završenosti, planiranje rješavanje problema i prihvaćanja pojedinih komponenti.
- **Upravljanje okruženjem** fokusira se na aktivnosti potrebne da bi se pružila potrebna podrška projektu. Uključuje sve aktivnosti potrebne za podršku projektu (osiguranje procesa i alata).



Slika 5. Porast poslovne vrijednosti po fazama i pomoćnim disciplinama RUP-a

3.3.2. Potpuno objedinjeni proces

[20]

RUP model razvoja aplikacija zasnovan na procesima, ima brojne prednosti - prvenstveno što je zasnovan na razvoju koji polazi od iterativnog, na zahtjevima i arhitekturi zasnovanom pristupu. Ipak, kod njega postoje i određeni nedostaci. RUP opisuje samo proces razvoja programa, ne i ostale procese mimo samog razvoja iako pravi fokus organizacija nije samo na razvoju jednog programa već na razvoju, podršci i održavanju više sustava. Također, RUP ne podržava višesistemske infrastrukturne razvojne procese.

Kako bi se uklonili navedeni nedostaci, IBM-ov stručnjak Scot W. Ambler je sa svojim timom do 2004. godine nadograđivao RUP model te je tako nastao potpuni (poduzetnički) objedinjeni proces (eng. ***Enterprise Unified Process***) koji je dostupan preko Ambysoft organizacije.

Ključna razlika u odnosu na prethodni RUP model jest u dvije dodatne faze koje slijede nakon tranzicije:

- **Produkcija** predstavlja onu fazu u životnom ciklusu programa, nakon što je novi sustav bio uveden kod korisnika. Njena svrha je održati razvijeni program u korištenju sve dok ne bude ili zamijenjen novom verzijom programa ili povučen iz korištenja. Uključuje pružanje pomoći korisnicima u svakodnevnom radu, upravljanje sustavom, praćenje i nadziranje rada sustava, pripreme za oporavak u slučajevima gubitaka podataka.
- **Povlačenje** je faza uklanjanja programskog sustava iz korištenja. Npr. jer više nije potreban ili je zamijenjen nekim drugim sustavom. Uključuje detaljnu analizu integracije sustava u odnosu na druge sustave, redizajn ili rekonfiguraciju tih drugih ovisnih sustava, transformaciju postojećih podataka, arhiviranje nepotrebnih podataka, pripremu za deinstalaciju te testiranje preostalih sustava.

EUP u odnosu na RUP ne definira nove osnovne aktivnosti - discipline, ali zato definira jednu novu pomoćnu disciplinu:

- **Operacije i podrška** se fokusiraju na aktivnosti koje uključuju nadzor i podešavanje sustava, nadogradnju korištenog *hardware*-a, arhiviranje podataka, odgovaranje na korisničke zahtjeve, rješavanje ozbiljnih problema, bilježenje mogućih poboljšanja te pripremu za oporavak u slučaju gubitaka podataka.

Ali to nisu sve promjene. EUP definira i jednu novu granu aktivnosti – tzv. poduzetničke discipline koje imaju za zadatak pokriti više područja unutar organizacije kako bi podržali rješavanje složenih među-projektnih problematika i izazova, a to su:

- **Potpuno (poduzetničko) poslovno modeliranje** ima za cilj istražiti poslovne procese i strukturu te omogućava razumijevanje poslovnih aktivnosti, klijenata, dobavljača i sl. Pomaže identificirati probleme i područja koja je moguće automatizirati.
- **Upravljanje portfeljem** aplikacija omogućava bolje planiranje cjelokupnog skupa aplikacija kao i individualnih programa. To pomaže kod implementiranja novih zahtjeva, ali i kod planiranja funkcionalnosti.

- **Potpuno (poduzetničko) upravljanje arhitekturom** bavi se svim pitanjima vezanim uz arhitekturu sustava te sadrži modele kako definirati arhitekturu, izraditi prototipe i pokazne modele, itd. Cilj je osigurati konzistentnost između različitih sustava.
- **Strategija ponovnog korištenja** promovira razvoj korištenjem postojećih elemenata čime se povećava kvaliteta razvijanih sustava budući da se koriste već testirane komponente.
- **Upravljanje ljudima** uključuje razvijanje komunikacija, treniranje, motiviranje, nadzor kako bi se osiguralo da djeluju zajednički te da doprinose svim projektima.
- **Potpuna (poduzetnička) administracija** uključuje uvođenje i administriranje alatima, procesima i drugim komponentama koji su ključne infrastrukturni elementi IT-a organizacije.
- **Poboljšavanje programskih procesa** odnosi se na potrebu upravljanja, poboljšavanja i podržavanja višestrukih procesa u sklopu organizacije.

3.3.3. Ubrzani objedinjeni proces

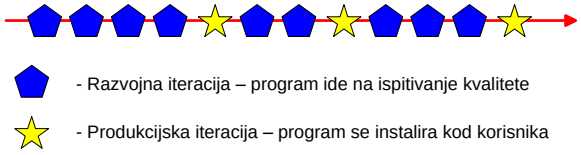
[16][25]

Ubrzani objedinjeni proces (eng. *Agile Unified Process*) pojednostavljena je verzija RUP modela. AUP opisuje jednostavni i razumljiv pristup razvoja poslovnih aplikacija korištenjem agilnih tehnika kombiniranih s RUP modelom. Prvi put predstavljen je 2005. godine od strane Scott W. Amblera.

U usporedbi s RUP modelom, AUP isto ima četiri osnovne faze (start, elaboracija, konstrukcija, tranzicija). Osnovna promjena je u pojednostavljenom prikazu osnovnih i pomoćnih disciplina (aktivnosti) pa su tako tri discipline poslovno modeliranje, obrada zahtjeva te analiza i dizajn zamijenjeni samo s jednom disciplinom – **modeliranjem**. Ono je važan dio AUP modela, ali ne dominira njime jer je potrebno da projekt ostane ubrzan razvijajući modele i dokumente koji su tek jedva dovoljno dobri.

Druga razlika u odnosu na RUP je u promijenjenoj pomoćnoj disciplini upravljanja konfiguracijama i promjena. Kod AUP modela ta se disciplina odnosi samo na **upravljanje konfiguracijama** budući da su kod ubrzanog razvoja aktivnosti vezane uz upravljanje promjenama tipično dio obrade zahtjeva što je kod AUP-a dio modeliranja.

Umjesto da se za isporuku programa koristi pristup gdje se sav program isporučuje u jednoj isporuci, kod AUP modela program se isporučuje u dijelovima (npr. verzija 1, verzija, 2, ..., verzija n). Pojedine razvojne verzije isporučuju se na kraju svake iteracije. Tu je potrebno razlikovati različite iteracije. Postoje razvojne iteracije na kraju kojih se razvijeni program koristi u procesu ispitivanja kvalitete, a postoje i produkcijske iteracije na kraju kojih se program stavlja u produkciju kod korisnika.



Slika 6. Produkcijske i razvojne iteracije

U pravilu vrijeme potrebno za isporuku prve verzije traje duže od ostalih. To je zbog toga što je u toj fazi potrebno napraviti određene administrativne poslove kako bi se razvoj uhodao te da bi ljudi postali efikasni i kooperativni. Ako bi razvoj prve verzije trajao npr. 12 mjeseci, razvoj druge mogao bi trajati npr. 9 mjeseci, a svih ostalih po 6 mjeseci. To može biti veoma korisno jer se dosta rano fokus usmjerava na uvođenje sustava kod korisnika, a to iskustvo pomaže kod izbjegavanja problema u kasnijim isporukama.

Ubrzani objedinjeni proces (AUP) zasniva se na nekoliko principa (filozofija):

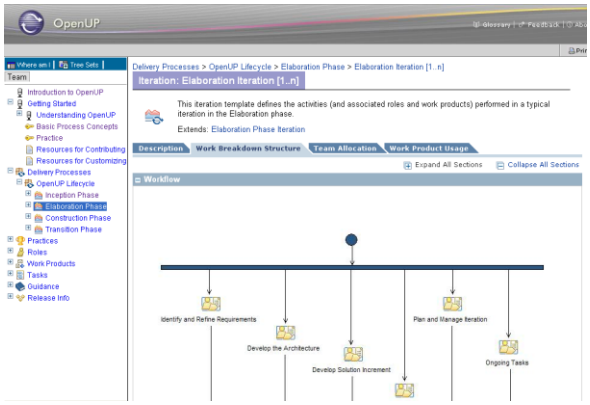
- **Radnici znaju što rade** tj. ljudima nije potrebno čitanje detaljnih dokumentacija o procesima već će im trebati neki osnovni dizajn te po mogućnosti trening.
- **Jednostavnost** opisa što znači da je sve opisano koncizno korištenjem manjih uputa, umjesto detaljnih priručnika.
- **Ubrzanost** je jedno od osnovnih načela ovog procesa.
- **Fokusiranost na važne aktivnosti**, a ne na ostale koje bi se možda mogle dogoditi tijekom projekta.
- **Nezavisnost o korištenim alatima** omogućava korištenje bilo kojih alata, tj. onih koji su najpogodniji za neki konkretni posao (najčešće jednostavni alati).

- **Prilagodljivost AUP modela** potrebama razvojnog tima.

3.3.4. Otvoreni (osnovni) objedinjeni proces

[19]

Otvoreni objedinjeni proces (eng. *Open Unified Process* - OpenUP) dio je Eclipse Process Framework razvojnog okruženja koji je stvoren kako bi pružio najbolje prakse različitih razvojnih i programerskih perspektiva i potreba. Razvijen je tijekom 2005. i 2006. godine, ali se i dalje razvija (zadnja verzija izdana 27.02.2009).



Slika 7. Eclipse OpenUP

OpenUP je agilna verzija UP modela koja sadrži minimalni skup najboljih praksi kako bi pomogao timovima da postanu efikasni u razvoju programa. Da ne bi postao kompliciran, sadrži samo osnovni sadržaj. Stoga ne posjeduje upute za mnoge slučajeve s kojima bi se projekt mogao susresti (npr. veliki timovi, specifične tehnologije, sigurnost aplikacija, kritiče aplikacije, eksternalizacija poslova i sl.).

OpenUP zasniva se na četiri osnovna principa:

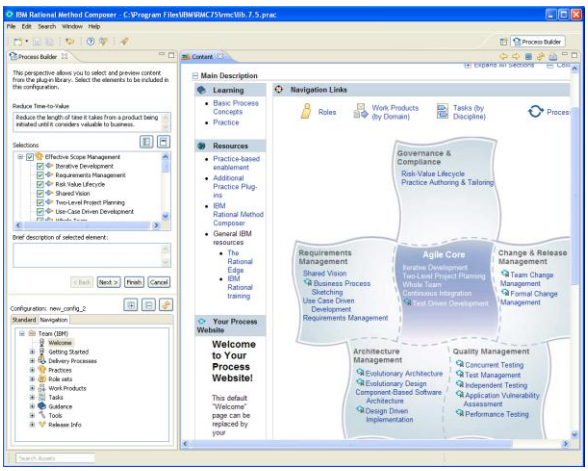
- **Poticanje suradivanja** kako bi se uskladili različiti interesi te da bi se omogućilo međusobno razumijevanje. je jedno od osnovnih načela ovog procesa.
- **Uravnoteženje konkurentnih prioriteta** čime bi se trebala povećati vrijednost konačnog proizvoda za korisnika, ali u skladu s nametnutim ograničenjima.
- **Rana usmjerenost na arhitekturu** kako bi se minimalizirali rizici te organizirao razvoj.
- **Postići neprestano dobivanje povratnih podataka** i to što ranije od zainteresiranih strana, kako bi se stvari mogle poboljšavati.

OpenUP razvijen je za potrebe malih timova (3-6 ljudi) i malih projekata (3-6 mjeseci). Stoga su neke od osnovnih i pomoćnih disciplina uklonjene, kao što su poslovno modeliranje i upravljanje okruženjem. Te aktivnosti su nepotrebne za tako male projekte i timove.

3.3.5. Prednosti i nedostaci modela objedinjenog procesa

[18][22][23]

Karakteristika objedinjenih procesa je njihova prilagodljivost za različite namjene. Stoga postoji nekoliko praksi korištenja ovisno o jednostavnim ili složenim potrebama. Tako npr. za manje projekte može biti pogodniji agilni model, a za neke racionalni od IBM-a. Tu se vidi i razlika u podršci pa su tako preporuke za agilni objedinjeni proces dostupan u datoteci veličine 0.5MB koja je zadnji put osvježena u svibnju 2006., za otvoreni objedinjeni proces izdan od strane Eclipse Proces Framework projekta 3.5MB, dok za potrebe korištenja probne verzije IBM-ovog racionalnog procesa (IBM Rational Method Composer 7.5) potrebno skinuti najmanje 0.5GB podataka.



Slika 8. IBM Rational Method Composer

Kad se promatra samo osnovni objedinjeni proces – RUP, **prednosti** su u tome što je dobro dokumentiran, koristi UML dijagrame za poboljšanje analize i dizajna, javan je, postoje treninzi pa i certifikacija. Jedna od prednosti je što RUP promiče proaktivno planiranje promjena u korisničkim zahtjevima u procesu razvoja. Također, RUP olakšava ponovno korištenje već

izrađenog koda te ubrzava integraciju novog koda.

Glavni **nedostatak** RUP-a je prevelika kompleksnost samog procesa. Pretežak je za naučiti, pretežak za ispravnu primjenu te zahtjeva prave stručnjake. Da bi se mogao koristiti potrebno ga je prvo prilagoditi konkretnim potrebama projekta, što dodaje novu dozu kompleksnosti razvoju, jer da bi se objedinjeni proces mogao uopće implementirati, potrebno ga je prvo modificirati. RUP ne obuhvaća ni sociološki aspekt razvoja te ne daje detalje kako istinski razvijati inkrementalno. Također, sam RUP može završiti neorganiziranim razvojem.

3.4. Kaubojsko programiranje

[26][27]

Kaubojsko programiranje je model koji pretpostavlja nepostojanje definirane metode rada. Članovi tima rade na poslovima za koje misle da su ispravni, na način na koji misle da je ispravan. Programeri imaju autonomiju nad razvojnim procesom, a to uključuje kontrolu vremenskih rokova, algoritama, alata i stila programiranja.

„Kauboji“ mogu biti pojedinci ili dio razvojnog tima, koji nemaju nadređene ili nadređeni kontroliraju samo aspekte koji nisu povezani s tehnikama programiranjem (priroda, okvir, skup funkcionalnosti željenog proizvoda). Nadređeni definiraju što se treba napraviti, ali ne definiraju kako se to treba napraviti.

Takav način programiranja može imati i pozitivne i negativne posljedice, što prvenstveno ovisi o stavovima o ulogama i formalnim procesima u razvoju programa. Također, uspjeh u pravilu mnogo ovisi i o individualnim kvalitetama programera.

Kaubojsko programiranje nije metoda agilnog razvoja. Često ponavljanje analize planova, komunikacija licem u lice uz rijetko korištenje pisanih dokumenata, uzrokuje često miješanje agilnog programiranja s kaubojskim programiranjem. Glavna razlika je u tome što agilni timovi slijede definirane, a često i vrlo striktne procese.

Zanemarivanjem mehanizama kontrole, nekorištenje dobro definiranih procedura, često dovodi do aktivnosti koje su poznate kao kaubojsko programiranje. Ipak, ne može se pretpostaviti kako korištenje metoda koje se mogu opisati kao kaubojsko programiranje treba imati negativne akontacije. Osobe koje rade po principima kaubojskog programiranja, obično su vrlo iskusne te imaju jasnu sliku ciljeva i okvira cijelog projekta.

Primjera projekata koji su zasnovani na kaubojskom programiranju je mnogo i u ovom tekstu neće biti analizirani. Npr. početak Linux projekta, Google web tražilice, Apache web poslužitelja, MySQL baza podataka, Facebook mreža i dr. Svim tim projektima zajedničko je to da su izašli „iz ničega“, bez velikih resursa, na temelju zalaganja pojedinaca. Stoga će neke organizacije dozvoliti ovakav tip razvoja ako imaju iznimne odgovorne individue koji mogu sami iznijeti projekt na svojim „leđima“. Ipak, većina organizacija teško će dozvoliti takav oblik razvoja jer tu ne postoje dobro definirane kontrole, procedure, planovi.

3.5. „Naslonjeni“ razvoj aplikacija

[28][29][30][32]

„Naslonjeni“ (eng. ***lean***) razvoj aplikacija je modifikacija principa i praksi proizašlih iz tzv. „naslonjene“ proizvodnje (eng. *lean manufactoring*) koju je prva počela primjenjivati Toyota u svojim proizvodnim sustavima. Ovaj oblik razvoja spada u agilne metode.

Sam naziv prvi put je proizašao u knjizi naziva Lean Software Development, autora Mary i Tom Poppendieck, 2003. godine. U knjizi su predstavljeni tradicionalni principi u svom modificiranom obliku kao i skup od 22 alata korištenih za potrebe ove metode.

Glavni cilj „naslonjenog“ razvoja aplikacija je isporučivanje što veće vrijednosti klijentima u što kraćem roku na najefikasniji mogući način. Da bi se to postiglo potrebno je da tim prihvaća stalno mijenjanje i poboljšavanje procesa. I jedino stalnim poboljšanjima kvalitete i proizvoda i procesa, moguće je ostvariti velike brzine razvoja i niske troškove.

U pravilu, postoji 7 osnovnih principa proizvodnje na kojima je „naslonjeni“ razvoj zasnovan:

- eliminacija gubitaka,
- povećanje obujma učenja,
- odlučivanje ostaviti za kraj,
- isporučivati čim brže,
- ovlastiti tim,
- ugrađivati integritet u sustav te
- gledanje cjelokupne slike.

Princip **eliminacije gubitaka** posuđen je od Taiichi Ohnoa, oca Toyotinog proizvodnog sustava. Njegovi osnovni oblici gubitaka, a to je

sve ono što ne donosi dodatnu vrijednost korisniku, modificirani su za potrebe razvoja programa i to su sljedeći:

- nepotrebni programski kod i funkcionalnosti,
- kašnjenja u procesu razvoja,
- nejasni zahtjevi,
- birokracija,
- slabe interne komunikacije.

Ali da bi se gubitci mogli eliminirati, potrebno ih je i znati prepoznati te uočiti. Ako se neka akcija može zaobići ili se isti rezultat može postići i bez te akcije, ona se smatra gubitkom. Djelomično programiranje koje se prekida tijekom procesa razvoja, dodatni procesi i funkcionalnosti koje ne trebaju korisnicima, čekanje na druge aktivnosti, pogreške i loša kvaliteta programa pa čak i organizacijski poslovi koji ne donose dodatnu vrijednost smatraju se gubitkom. Da bi se detektirao gubitak, koristi se princip što donosi dodatnu vrijednost. Potom je potrebno identificirati izvore gubitaka te ih eliminirati. Postupak je potrebno iterativno ponavljati sve dok čak i oni procesi i procedure koje izgledaju kao temeljne, ne budu uklonjene.

7 osnovnih gubitaka u razvoju programa
Prekomjerna produkcija (ekstra funkcionalnosti)
Inventar (nezavršeni, netestirani, ... dijelovi)
Višak koraka (nepotrebni procesi)
Motiviranost (mijenjanje poslova-zaduženja)
Defekti (testiranjem nepronađene greške)
Čekanja (na odluke, na korisnike,...)
Transport zadataka (gubitak znanja)

Slika 9. Osnovni gubici

Budući da je za razvoj aplikacija ključno učenje (novih tehnologija, principa rada, arhitektura, …), drugi princip „naslonjenog“ razvoja je **povećavanje obujma učenja**. Proces učenja ubrzava se primjenom kratkih vremenskih ciklusa tijekom kojih se obavlja raščlamba i testiranje integracije. Pomoću kratkih sastanaka s klijentima ubrzava se pristizanje povratnih informacija potrebnih za procjenu trenutnog stanja i prilagođavanje za buduća poboljšanja. Tijekom tih kratkih sastanaka, na temelju postignutih rezultata klijenti bolje poimaju svoje zahtjeve, a s druge strane razvojni tim uči kako bolje zadovoljiti te zahtjeve.

Broj pogrešaka potrebno je smanjiti izvođenjem testiranja čim se programski kod napiše. I umjesto da se generira dodana dokumentacija ili detaljno planiranje, različite ideje se mogu isprobavati te programirati.

Odlučivati što kasnije moguće zbog uvijek prisutne neizvjesnosti u procesu razvoja pa se kasnijim donošenjem odluka mogu postići bolji sveukupni rezultati. Time se potiče donošenje odluka temeljeno na činjenicama, a ne na pretpostavkama i očekivanjima. I što je sustav kompleksniji, to je potrebno ostaviti više mogućnosti za naknadne promjene jer se time omogućava odgađanje razvoja važnih ili ključnih komponenti. Upravo iterativni pristup promovira ovaj princip što kasnijeg odlučivanja koji olakšava prilagodbu na promjene i uklanjanje pogrešaka koje mogu biti vrlo skupe ako ih se otkrije tek nakon distribucije sustava.

Ovaj princip ne znači da u razvoju ne bi smjelo biti planiranja. Baš naprotiv, planiranje bi se trebalo fokusirati na različite opcije te prilagodbu trenutnoj situaciji kao i za rješavanje nejasnih situacija kreiranjem predložaka za brze akcije.

Isporučivati što brže moguće jedan je od principa budući da u današnjem dobu brzog razvoja tehnologija ne preživljavaju najveći već najbrži. Što prije se proizvod isporuči (s podnošljivom količinom i težinom pogrešaka), to će prije pristići i povratne informacije te će ih se moći uključiti u sljedeću iteraciju. Brzina omogućava zadovoljavanje korisnikovih današnjih potreba, a ne onoga što im je trebalo „jučer“.

Ovlastiti tim tj. omogućiti im da sami odlučuju o ključnim elementima u razvoju. U razvoju nije potrebno na menadžeri govore radnicima kako trebaju raditi već trebaju slušati radnike. Time menadžeri mogu lakše objasniti koje akcije je potrebno poduzeti te dati prijedloge za poboljšanja. Razvojni tim mora imati pristup do klijenata, a vođa tima treba osigurati podršku i pomoć u teškim situacijama.

Ugrađivati integritet u sustav i za potrebe korisnika i za potrebe samog razvoja. Za korisnika se integritet odnosi na isporuku, instalaciju, pristup, koliko je intuitivan sustav te koliko dobro mu rješava probleme. Kod razvoja je bitno da sistemske komponente funkcioniraju ispravno kao cjelina balansirajući između fleksibilnosti, lakoće održavanja, efikasnosti i osjetljivosti. Bitno je održavati sustav

jednostavnim, bez ponavljanja istog programskog koda, s minimalnom količinom funkcionalnosti. Na kraju je potrebno testirati integritet testiranjem koje treba potvrditi kako sustav radi ono što klijent očekuje od njega.

Gledanje cjelokupne slike potrebno je kako ne bi nastale pogreške pri implementaciji pojedinih komponenti. Sustav nije samo zbroj komponenti već je produkt interakcija između komponenti. Što je sustav veći, više je organizacija uključenih u njegov razvoj, te je više dijelova izrađivanih od strane različitih timova, a samim time je veća važnost posjedovanja dobro definiranih veza između pojedinih komponenti.

Osnovne **prednosti** *lean* razvoja navedene su u glavnim principima. U prvom redu tu je brzina razvoja, tj. potrebno je isporučivati čim prije moguće. Potom kvaliteta proizvoda, ali i procesa koji je podložan promjena. Kvaliteta se osigurava eliminacijom gubitaka, gledanjem cijele slike, dodatnom edukacijom zaposlenika,... Budući da se radi iterativno s naglaskom na važnijim funkcionalnostima, manji su i rizici da će razvoj otići u krivom smjeru, ili da će naglasak biti na nepotrebnim dijelovima. Sve navedeno, treba rezultirati nižim troškovima. Također, budući da su ovlašteni i obrazovani za promjene, i radnici bi trebali biti zadovoljniji.

Glavni **nedostatak** *lean* razvoja je što zahtjeva veliku podršku menadžmenta koji je velikim dijelom isključen. Stoga se pojavljuju brojne zapreke: nedostatak ovlaštenja zaposlenicima, nema vremena za edukaciju zaposlenika i gradnju intelektualnog kapitala, ne gleda se cijela slika,… Budući da sami zaposlenici imaju mogućnost mijenjanja ne samo finalnog proizvoda već i procesa razvoja, cijeli projekt može ispasti neorganiziran. A budući da je ovo jedna od novijih metodologija agilnog razvoja jedan od glavnih nedostataka je i manjak iskustava.

3.6. Ostale metode agilnog razvoja

[31][33][34][35][36][37][38]

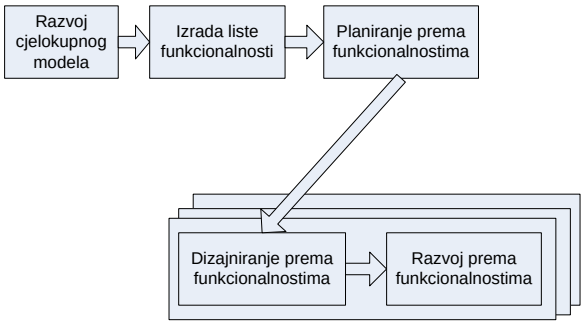
Iako nešto starije od prethodno navedenih, naredne agilne metode razvijane su i nakon njihovog prvotnog nastanka pa su stoga i ovdje ukratko opisane.

Razvoj prema funkcionalnostima (eng. ***Features Driven Development***) primijenio je prvi Jeff De Luca tijekom rada na projektu Singapurske banke 1997. godine, dok je „službeno“ predstavljen u knjizi Java Modeling in Color with UML. Ukratko, FDD se sastoji od 5 osnovnih procesa:

1. Razvoj cjelokupnog modela tijekom kojeg se definira obuhvat projekta i ciljanog sustava pri čemu se pozornost ne posvećuje detaljima već je bitno shvatiti i razumjeti područje u kojem će se razvijani sustav primjenjivati. Tijekom ovog procesa svi moraju dobro surađivati. Nakon što se timu objasni pojedina problematika, projektni tim dijeli se u manje grupe (preporučljivo 3 osobe po grupi) kako bi modelirali model za opisanu problematiku. Na kraju se rješenja uspoređuju te se konsenzusom odabire najbolje. Potom se nastavlja s drugim područjem (problematikom) dok sva ne budu obrađena i sva rješenja objedinjena u sveobuhvatni model.
2. Izrada liste funkcionalnosti drugi je proces tijekom kojeg je potrebno, na temelju spoznaja dobivenih u prvom procesu, identificirati sve funkcionalnosti potrebne da bi se zadovoljili zahtjevi projekta. Sustav se raščlanjuje na manje dijelove koji sadrže određene poslovne aktivnosti – funkcionalnosti sustava. One se opisuju u formi '<akcija> <rezultat> za/prema/od <objekt>', npr. 'izračunaj ukupne uplate za blagajnika'. Svaka funkcionalnost treba se moći realizirati u vremenskom intervalu od 2 sata do 2 tjedna. Ako za realizaciju treba više vremena, onda tu funkcionalnost treba raščlaniti na više manjih.
3. Planiranje prema funkcionalnostima prvi je korak prema planu razvoja. U ovom procesu određuje se kojim redoslijedom će funkcionalnosti biti razvijene, nastojeći pravilno rasporediti opterećenje među članovima te planiranjem ranih isporuka pojedinih komponenti proizvoda kako bi se zadovoljili klijenti.
4. Dizajniranje prema funkcionalnostima odnosi se na definiranje onih funkcionalnosti koje trebaju u vremenskom intervalu od 2 tjedna biti razvijeni. Pri tome se izrađuje detaljni sekvencijalni dijagram za svaku funkcionalnost. Na početku se „ponovi gradivo“, tj. programeri se detaljnije upoznaju s onim što razvijaju nakon čega se definira dizajn, a na kraju se to još dodano i prekontrolira.
5. Razvoj prema funkcionalnostima na temelju definiranog dizajna uključuje programiranje potrebnih funkcionalnosti. Pri tome nakon osnovnog razvoja slijedi inspekcija koda i testiranje te na kraju uključivanje u stabilno izdanje sustava.

Iako FDD ima osnovni nedostatak u tome što iziskuje dosta truda i ulaganja u osnovne procese istraživanja, razvoja, testiranja i evaluacije, njegova glavna prednost je što omogućava česte i

vremenski predvidljive isporuke funkcionalnih komponenata sustava.



Slika 10. Osnovni procesi FDD-a

Metoda razvoja dinamičnih sustava (eng. DSDM – ***Dynamic Systems Development Method***) originalno je predstavljena 1995. godine te je bila zasnovana na RAD metodologiji. Ipak ta metodologija se nastavila razvijati pa je tako 2006. godine izdana inačica 4.2 DSDM-a.

DSDM se fokusira na IT projekte s kratkim rokovima i ograničenim budžetima, definira 3 osnovne sekvencijalne faze: pred-projektna, projekta i post-projektna.

U pred-projektnoj fazi projekt se identificira, te se osigurava budžet i angažiranost za njegovu realizaciju.

Projektna faza sastoji se od 5 podfaza. Prve dvije, studija izvedivosti i poslovna studija, komplementarne su, a odnose se na analizu ostvarivosti s obzirom na moguće rizike, te na karakteristike ciljanog tržišta. Treća, iteriranje funkcionalnog modela, sastoji se od identificiranja funkcionalnosti, kreiranja i revizije funkcionalnog prototipa, zajedno s određivanjem vremenskog rasporeda kad i kako treba razviti pojedine funkcionalnosti. Četvrta, iteriranje dizajna i razvoja, odnosi se na identificiranje, kreiranje i provjeru dizajnerskog prototipa što uključuje i vremenske planove, strategiju implementacije kao i korisničku dokumentaciju i plan testiranja. Zadnja podfaza odnosi se na implementaciju, a uključuje korisničko prihvaćanje dizajna, treniranje krajnih korisnika, implementaciju kod krajnjih korisnika te reviziju utjecaja razvijenog sustava na poslovanje tj. utvrđivanje koliko izrađeni sustav zadovoljava postavljene ciljeve.

Post-projekta faza osigurava da sustav funkcionira efektivno i efikasno, a to se ostvaruje kroz održavanje, poboljšanja i popravke.

4. Sigurnosni aspekti razvoja aplikacija

[39][40]

U novije vrijeme sve veća pažnja se posvećuje sigurnom razvoju aplikacija čijim korištenjem se nastoji minimalizirati vjerojatnost pojave sigurnosnih ranjivosti. Sigurnosne ranjivosti programskih paketa su pogreške u programima koje napadači i zlonamjerni korisnici mogu iskoristiti za narušavanje integriteta računala. Uzroci ranjivosti su razni, a najčešći su oni vezani su uz sigurnost memorije, neodgovarajuću obradu ulaznih podataka, simultano korištenje zajedničkih resursa, neprikladnu dodjelu ovlasti ili uz pogreške grafičkog sučelja. Ranjivosti ili sigurnosni propusti se često javljaju zbog nepažnje programera. Napadač može zlouporabiti ranjivost aplikacije na mnogo načina, a najpoznatiji su pokretanje proizvoljnog programskog koda, neovlašten pristup podacima, napad uskraćivanja usluga (eng. *Denial of Service* - DoS), XSS (eng. *Cross-Site Scripting*) napad i još mnogi drugi.

Standardne tehnike razvoja programskih paketa potpuno su neprikladne za kreiranje sigurnih aplikacija zbog svoje orijentiranosti na ispravne funkcionalnosti programa i istovremenog potpunog ignoriranja ostalih. Pri tome su najčešće greške programera sljedeće:

- **Prepisivanje spremnika** (eng. *buffer overflow*) čiji temeljni uzrok je uporaba statičnih varijabli fiksne veličine za pohranu ulaznih podataka neograničene dužine. Time napadač može podmetnuti vlastiti programski kod koji će se prepisati van alocirane memorije te u idealnom slučaju i izvršiti.
- **Propusti oblikovanja znakovnih nizova** koji služe kao varijabilni argumenti funkcija i potrebno ih je ukloniti validacijom ulaznih podataka i provjerom iznimaka programskog koda.
- **Neprikladna autentikacija** koja se lako zaobilazi ili lako probija različitim *brute force* napadima. I ukoliko je autentikacija korisnika nepravilno izvedena, sve ostale sigurnosne funkcionalnosti, kao što su kriptiranje, provjere ispravnosti i pouzdanosti informacija (eng. *audit*) te autorizacija, postaju beskorisne.
- **Neprikladna autorizacija** koja postoji u slučajevima kad se autenticiranom

korisniku dozvoljava pristup resursima koji mu nisu dodijeljeni.

- **Neprikladna kriptografija** u obliku nepostojanja enkripcije za udaljene konekcije ili u slučajevima zastarjelih kripto-algoritama (npr. DES).
- **Problemi simultanog korištenja resursa** zasnovani su uglavnom na korištenju privremenih resursa što napadač može pažljivim odabirom trenutka napada iskoristiti za modificiranje privremenih podataka.

Sigurnost programskog koda odnosi se na otpornost aplikacije na razne zlouporabe napadača (korisnika) i u tom smislu na nedostatak funkcionalnosti. U nastavku su dani osnovni principi razvoja sigurnog koda kao i Microsoftov model sigurnog razvoja.

4.1. Osnovni principi sigurnog razvoja

Razvoj sigurnih programa je proces koji zahtijeva opreznost i obazrivost pri svakom koraku te na svim razinama organizacije. Razvoj sigurnog programskog koda zahtijeva stroge sigurnosne specifikacije, razvojne programere s mnogo iskustva i znanja na području sigurnosti te skupinu za procjenu kvalitete (eng. *quality assessment* (QA) *team*) za otkrivanje sigurnosnih problema.

Dužnost svakog programera u timu je razviti programski kod bez pogrešaka (eng. *bug-free*). Zbog toga programeri moraju međusobno sudjelovati u revizijama te ponovnim evaluacijama dizajna i programskog koda aplikacije. Kada programer završi pisanje određenog dijela koda, ostali članovi tima moraju potpuno pregledati njegov dizajn i/ili programski kod. Osnovna načela razvoja programa diktiraju pisanje malih, samostojećih jedinica nazvanih modulima. Svaki se modul treba izolirati od štetnog utjecaja ostalih modula, a to se može postići skrivanjem dijelova programa (enkapsulacijom). Programiranje sigurnog koda uvjetuje da svaka akcija koju program (ili potprogram izvodi) mora biti sadržana u njegovim specifikacijama. Pri pisanju sigurnih programa potrebno je pridržavati se tri osnovna principa:

- **Skrivanje informacija** (dijelova programskog koda – enkapsulacija) – čija osnovna ideja je izravna posljedica "principa najmanje ovlasti" (eng. *Principle of Least Privilege*) i ona čini temelj integriteta i sigurnosti programskog koda.
- **Defanzivno** (robusno) **programiranje** temelji se na ideji da dani program nakon pokretanja ne smije ovisiti ni o čemu što je korisnik sam stvorio. Stoga je u svakom modulu potrebno raditi inspekciju ulaznih podataka.
- **Pretpostavljanje nemogućeg** zasnova se na što temeljitijem testiranju i ugradnji provjere pogrešaka na svim mjestima, pa čak i tamo gdje je mogućnost njihove pojave samo teoretska.

4.2. Microsoftov model sigurnog razvoja

Iskustvo o sigurnosti programa iz realnog svijeta rezultiralo je visoko prioritetnim pravilima za izgradnju sigurnijih aplikacija. Tvrtka Microsoft ta pravila naziva SD3+C (eng. *Secure by Design, Secure by Default, Secure in Deployment, and Communications*):

- **Sigurnost po dizajnu** (eng. *Secure by Design*): programski paket treba biti konstruiran, dizajniran i implementiran tako da štiti sebe i informacije koje obrađuje te treba pružati otpor napadačima.
- **Zadana sigurnost** (eng. *Secure by Default*): u stvarnosti, niti jedan programski paket ne pruža potpunu sigurnost. Zato programeri trebaju pretpostaviti da će se u programu otkriti sigurnosni propusti. Početne postavke aplikacije trebaju promovirati sigurnost. Na primjer, tvorničke postavke su pokretanje programa s minimalnim potrebnim ovlastima, servisi i svojstva koja nisu prijeko potrebna su isključeni ili dostupni malom broju korisnika.
- **Sigurnost pri instalaciji** (eng. *Secure in Deployment*): Svaki program treba imati prateće upute i alate koji olakšavaju upotrebu programa na siguran način. Instalacija nadogradnje i ispravljenih inačica treba biti jednostavna.
- **Komunikacija** (eng. *Communications*): Razvojni programeri moraju biti spremni na pojavu sigurnosnih ranjivosti u programskim paketima te otvoreno i odgovorno komunicirati s krajnjim korisnicima i/ili administratorima da im

pomognu otkloniti sigurnosni problem (npr. izdavanje zakrpa, zaobilaženje problema i sl.)

Dok svaki element SD3+C razvojnog procesa definira potrebne značajke programa, prva dva elementa – sigurnost po dizajnu i zadana sigurnost – najviše pridonose sigurnosti programa. Sigurnost po dizajnu diktira pravila za sprečavanje pojave ranjivosti, a zadana sigurnost zahtjeva minimalnu prvotnu izloženost programa napadima.

Kada se promatra tipični životni ciklus nekog Microsoftovog razvojnog procesa, on bi se mogao podijeliti u sljedeće faze:

1. **Analiza zahtjeva** koja definira
   a. tražene funkcionalnosti,
   b. smjernice kvalitete,
   c. dokumentaciju o arhitekturi, i
   d. rokove.
2. **Dizajn** uključuje
   a. specifikaciju dizajna i
   b. funkcionalne specifikacije.
3. **Implementacija** uključuje
   a. razvoj novog koda te
   b. testiranje i verifikaciju.
4. **Verifikacija** uključuje
   a. testiranje i verifikaciju te
   b. ispravljanje pogrešaka.
5. **Izlazak** rezultira
   a. digitalnim potpisom koda i
   b. izdanom inačicom.
6. **Podrška** uključuje
   a. podrška programu
   b. servisna izdanja
   c. nadogradnja

Sigurnosne mjere integriraju SD^3+C paradigmu u postojeći proces razvoja stvarajući sveukupan organizacijski proces kao što je opisano u nastavku.

Tijekom faze **analiza zahtjeva (1)** osnovna grupa za razvoj povezuje se sa središnjim odjelom za sigurnost pri čemu razvojni tim dobiva svog sigurnosnog savjetnika (tzv. *security buddy*). Njegova uloga je povezati razvojnu i sigurnosnu skupinu kako bi pomogao odrediti način integracije sigurnosti u proces razvoja, kako će se sigurnosne komponente i sigurnosne mjere uklopiti i surađivati s nekim drugim programima, identifikaciju ključnih ciljeva te optimizaciju sigurnosti uz minimalne promjene planova i rokova. On sudjeluje u projektu sve do izlaska proizvoda na tržište.

Dizajn (2) definira sveukupnu strukturu programa iz perspektive sigurnosti te se identificiraju komponente čiji je ispravan rad osnova sigurnosti (tzv. "provjerena baza"). Određuju se tehnike dizajna, kao što je uslojavanje (viši slojevi mogu ovisiti o servisima nižih slojeva, ali obratna ovisnost nije dozvoljena), uporaba programskih jezika s dobro definiranim tipovima podataka, aplikacija s minimalnim ovlastima i minimizacija područja potencijalnih napada. U dizajnu se primjenjuje i proces modeliranja prijetnji koji otkriva prijetnje koje mogu naškoditi svakom od sredstava te se procjenjuje rizik. Na temelju modela kreiraju se i protumjere koje umanjuju rizik. Protumjere mogu biti enkripcija ili dodatna zaštite korištenih resursa.

U fazi **implementacije (3)** proizvodna skupina piše programski kod, testira ga i integrira u programski paket. Poduzimaju se koraci za uklanjanje sigurnosnih pogrešaka i sprečavanje njihovog nastanaka. Rezultati modeliranja prijetnji pružaju smjernice za implementacijsku fazu. Sigurnosni elementi ova faze su: primjena standarda sigurnog pisanja programskog koda, primjena alata za testiranje nasumičnim upisom ulaznih podataka (tzv. *fuzzing*), statičke analize kontrole ispravnosti programskog koda, te revizija programskog koda.

U fazi **verifikacije (4)** izdaje se beta inačica koju testiraju i krajnji korisnici, ali i razvojni tim provodi dodatne sigurnosne revizije (tzv. *security push*).

Tijekom **izlaska (5)**, tj. faze puštanja proizvoda na tržište, programski se paket podvrgava Konačnoj sigurnosnoj reviziji ("KSR") (eng. *Final Security Review*) od strane središnje sigurnosne skupine. Cilj KSR-a je odgovoriti na jedno pitanje: "Je li ovaj programski paket spreman za korisnike sa stajališta sigurnosti?". Programski paket mora biti stabilan prije provođenja KSR-a, a predviđene preinake su minimalne i nevezane uz sigurnost.

Tijekom **podrške (6)** razvojne skupine moraju biti spremne odgovoriti na nove metode napada i izdati sigurnosne preporuke i zakrpe kada je to prikladno. Naime, unatoč primjeni ciklusa sigurnog razvoja programskih paketa, na kraju se neizbježno dobivaju programi koji imaju manje ili više sigurnosnih ranjivosti. Čak i kad bi bilo moguće ukloniti sve ranjivosti tijekom razvoja, uvijek će se naći inovativni napadači s novim metodama napada. Cilj faze podrške je učiti iz prethodnih pogrešaka i iskoristiti informacije iz izvještaja o ranjivostima za otkrivanje novih sigurnosnih propusta. Tijekom ove faze razvojna skupina i sigurnosna skupina

imaju priliku prilagoditi razvojni proces tako da se slične pogreške više ne javljaju.

5. Zaključak

U današnje doba, kompetitivne prednosti u razvoju programskih proizvoda proizlaze uglavnom iz brzine i fleksibilnosti. Kako bi bile konkurentne organizacije moraju biti usko povezane sa svojim klijentima te tijekom cijelog razvoja biti otvorene i spremne na promjene korisničkih zahtjeva, u granicama zadanih okvira (budžet, vremenski rokovi, itd.). Iz tog razloga danas, uz tradicionalne modele razvoja, kod organizacija koje su spremne na inovacije dominiraju uglavnom agilne metode.

Svi navedeni „novi“ modeli razvoja aplikacija imaju svoje prednosti i nedostatke pa tako ne postoji najbolja metodologija. Izbor najprikladnije metodologije ovisiti će uvijek o samoj organizaciji, o opsegu i kompleksnosti projekata, kooperantnosti klijenata, iskustvu zaposlenika, i drugim vanjskim i unutarnjim čimbenicima.

Ipak, bez obzira na odabranu metodologiju, kod svih tih modela razvoja važnu ulogu uvijek mora imati i testiranje. U tom procesu sve je veći naglasak i na razvoju sigurnog programskog koda. Zbog brojnih propusta zbog kojih su Microsoftovi proizvodi u prošlosti bili poznati kao vrlo nesigurni, upravo je Microsoft uložio velik trud u razvoj sigurne metodologije razvoja kako bi minimalizirao broj sigurnosnih propusta. Time se ukupni razvoj nažalost usporava, ali rezultira sigurnijim kodom što je u današnje doba postalo jako važno jer su krajnji korisnici premalo svjesni važnosti pravovremenih instalacija sigurnosnih zakrpa.

6. Reference

[1] Project Lifecycle Models: How They Differ and When to Use Them, http://www.business-esolutions.com/islm.htm, siječanj – travanj 2009.

[2] Extreme programming, http://www.extremeprogramming.org, siječanj – travanj 2009.

[3] http://www.xprogramming.com, siječanj – travanj 2009.

[4] P. Emery, The dangers of extreme programming, http://members.cox.net/cobbler/XPDangers.htm, siječanj – travanj 2009.

[5] Benefits of Extreme Programming, http://www.qualitycode.com/html/Essay10.html, siječanj – travanj 2009.

[6] List of Unit Testing Frameworks, http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks, siječanj – travanj 2009.

[7] H. Erdogmus, M. Morisio, M. Torchiano: On the Effectiveness of the Test-First Approach to Programming, Proceedings of the IEEE Transactions, 2005.

[8] Agile Alliance, http://www.agilealliance.com/, siječanj – travanj 2009.

[9] Manifesto for Agile Software Development, http://www.agilemanifesto.org/, siječanj – travanj 2009.

[10] A.V.Sanjay, Overview of Agile Management & Development Methods, 27. lipanj 2005., http://www.projectperfect.com.au/info_agile_programming.php

[11] 10 Good Reasons To Do Agile Development, http://www.agile-software-development.com/2007/06/10-good-reasons-to-do-agile-development.html, siječanj – travanj 2009.

[12] Overview of Agile Management & Development, http://www.projectperfect.com.au/info_agile_programming.php , siječanj – travanj 2009.

[13] J.Proffitt: TDD Proven Effective! Or is it? http://theruntime.com/blogs/jacob/archive/2008/01/22/tdd-proven-effective-or-is-it.aspx, siječanj – travanj 2009.

[14] 10 Key Principles of Agile Software Development, http://www.agile-software-development.com/2007/02/10-things-you-need-to-know-about-agile.html, siječanj – travanj 2009.

[15] Bolshakova E., Programming paradigms in computer science education, International Journal "Information Theories & Applications" Vol.12, str. 285-290, 2005.

[16] The Agile Unified Process (AUP) Home Page, http://www.ambysoft.com/unifiedprocess/agileUP.html, siječanj – travanj 2009.

[17] Agile Modeling and the Rational Unified Process (RUP), http://www.agilemodeling.com/essays/agileModelingRUP.htm, siječanj – travanj 2009.

[18] P.B. Shenulal: The advantages of using Rational Unified Process in Software Development, Raizel Consulting, rujan 2007.

[19] Eclipse Framework Process Project, http://www.eclipse.org/epf/, siječanj – travanj 2009.
[20] Enterprise Unified Process, http://www.enterpriseunifiedprocess.com/, siječanj – travanj 2009.
[21] S.W.Ambler: A Manager's Introduction to The Rational Unified Process (RUP), http://www.ambysoft.com/downloads/managersIntroToRUP.pdf, siječanj – travanj 2009.
[22] N. Shahid: Rational Unified Process, http://ovais.khan.tripod.com/papers/Rational_Unified_Process.pdf, siječanj – travanj 2009.
[23] Unified Process vs Agile Processes, http://ccollins.wordpress.com/2008/06/11/unified-process-vs-agile-processes/, siječanj – travanj 2009.
[24] IBM: Rational Unified Process - Best Practices for Software Development Teams, 2005, http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf, siječanj – travanj 2009.
[25] S. Prabhudas - Agile Unified Process, http://www.test2008.in/Test2008/pdf/Sidd%20Prabhudas%20-%20Agile%20Unified%20Process.pdf, siječanj – travanj 2009.
[26] M. West: Delving into Cowboy Programming, http://cowboyprogramming.com/2007/01/11/delving-into-cowboy-programming/, siječanj – travanj 2009.
[27] Wikipedia: Cowboy coding, http://en.wikipedia.org/wiki/Cowboy_coding, siječanj – travanj 2009.
[28] Lean Software Institute, http://www.leansoftwareinstitute.com, 2009.
[29] T.&M. Poppendieck: The History of Lean Software Development, http://www.informit.com/articles/article.aspx?p=664147&seqNum=6, siječanj – travanj 2009.
[30] D.R.Kumar: Lean Software Development, http://www.projectperfect.com.au/downloads/Info/info_lean_development.pdf, siječanj – travanj 2009.
[31] The Definitive List of Software Development Methodologies, http://www.noop.nl/2008/07/the-definitive-list-of-software-development-methodologies.html, siječanj – travanj 2009.
[32] A. Shalloway: Lean Software Development: Speed – Quality – Low Cost, Net Objectives, prosinac 2006.
[33] Department of the Navy USA: Software Process Improvement Initiative (SPII) 2.0., Software Development Techniques (SWDT) Focus Group, listopad 2007.
[34] Feature Driven Development (FDD) and Agile Modeling, http://www.agilemodeling.com/essays/fdd.htm, siječanj – travanj 2009.
[35] Feature Driven Development, http://www.featuredrivendevelopment.com/, 2009.
[36] M. Bauer: Successful Web Development Methodologies, svibanj 2005, http://www.sitepoint.com/article/successful-development/, siječanj – travanj 2009.
[37] DSDM Consortium, http://www.dsdm.org/, siječanj – travanj 2009.
[38] J.J. Voigt: Dynamic System Development Method, Department of Information Technology, University of Zurich, siječanj, 2004.
[39] CARNet CERT & LSS: Izrada sigurnog koda, http://www.cert.hr/filehandler.php?did=313, prosinac 2007.
[40] The Trustworthy Computing Security Development Lifecycle, http://msdn.microsoft.com/en-us/library/ms995349.aspx, siječanj – travanj 2009.