

**Sveučilište u Zagrebu**  
**Fakultet elektrotehnike i računarstva**  
**Zavod za elektroniku, mikroelektroniku, računalne i inteligentne sustave**

## **Procesi programskog inženjerstva**

Oblikovanje programske potpore: Interna skripta kolegija namijenjena studentima

**Autori:**

**dr. sc. Alan Jović, dipl. ing.**  
**dr. sc. Marko Horvat, dipl. ing.**  
**dr. sc. Danko Ivošević, dipl. ing.**  
**Nikolina Frid, mag. ing. rač.**

**Autori zadržavaju sva prava nad sadržajem skripte.**

**Zagreb, rujan 2014.**

## Sadržaj

1. Uvod .....	3
2. Definicije i temeljna pitanja programskog inženjerstva .....	5
2.1. Programska potpora i programsko inženjerstvo .....	5
2.2. Temeljna pitanja programskog inženjerstva .....	7
3. Životni ciklus programske potpore .....	13
3.1. Specifikacija programske potpore .....	13
3.2. Oblikovanje i implementacija programske potpore .....	13
3.3. Validacija i verifikacija programske potpore .....	15
3.4. Evolucija programske potpore .....	18
4. Inženjerstvo zahtjeva .....	20
4.1. Višedimenzijska klasifikacija zahtjeva .....	20
4.2. Procesi inženjerstva zahtjeva .....	25
5. Procesi i modeli procesa programskog inženjerstva .....	35
5.1. Iteracije u modelima procesa programskog inženjerstva .....	36
5.2. Vodopadni model .....	39
5.3. Evolucijski model .....	41
5.4. Komponentno-usmjereni model .....	42
5.5. Modelno-usmjereni razvoj (RUP) .....	53
5.6. Agilni pristup razvoju programske potpore .....	56
6. Alati i okruženja za potporu razvoja programa .....	61
6.1. Klasifikacija CASE-alata .....	61
6.2. Sustavi za kontrolu inačica programske potpore .....	64
7. Literatura .....	70

## 1. Uvod

Procesi programskog inženjerstva promatraju razvoj programske potpore od svojih začetaka u obliku specifikacije korisničkih zahtjeva sve do krajnje implementacije, isporuke i održavanja. Budući da razvoj programske potpore nije jednoznačan i ovisi o cijelom nizu faktora i okolnosti, inženjeru se daje velika uloga u osmišljavanju i ostvarivanju konačnog proizvoda. Pritom najveću ulogu igra iskustvo i čitavi mentalni sklop inženjera. Razvoj programske potpore nesumnjivo je intelektualna aktivnost koja doprinosi razvoju modernog društva. Međutim, intelektualna aktivnost sama po sebi nije dovoljna da bi se uhvatili ukoštac sa stvarnim problemima i velikim programskim sustavima. S tim u svezi, ideja ove skripte je da pomogne budućim inženjerima koji se prvi put susreću s organiziranim razvojem ozbiljnih programskih proizvoda tako što će im dati temeljna znanja o disciplini programskog inženjerstva.

Tradicionalni razvoj programske potpore poglavito se oslanjao na umješnost i dovitljivost inženjera u implementaciji, pri čemu je dotični trebao razmišljati o svojoj sili hardverskih ograničenja, nespretnosti programskih jezika, nedefiniranih protokola ispitivanja i nerazvijenim alatima za pomoć pri razvoju programa. Moderni razvoj programske potpore je s jedne strane puno pojednostavio čitav postupak razvoja, no i postavio nove, visoke kriterije kvalitete programskog proizvoda. Tako sada više nije dovoljno da neki program donekle dobro radi, već on mora izvrsno raditi, što znači da mora biti točan, učinkovit, bez pogrešaka, nadogradiv, oku promatrača privlačan i što je vjerojatno najbitnije, lagan za naučiti i koristiti.

Pri razvoju suvremene programske potpore, nameću se dva vrlo važna pojma bez kojih danas nije moguće zamisliti kvalitetan programski proizvod: **apstrakcija** i **struktura**. Apstrakcija znači da se razvoju programske potpore pristupa u vidu modela na različitim razinama složenosti, pri čemu svaka viša razina sakriva (apstrahira) detalje niže razine. Tako se složeni sustavi pri modeliranju razlažu na dijelove, a apstrakcija je prisutna i u vidu razlaganja arhitekture i u vidu jezika za prikaz modela sustava, koji ide od najnižeg, strojnog jezika pa sve do grafičkih jezika vrlo visoke razine. Struktura ili strukturiranje znači da se izradi programskog proizvoda pristupa sistematski, pri čemu se svaki korak razvoja, od definiranja zahtjeva, preko specifikacije arhitekture sve do programskog koda i razvoja novih inačica dokumentira na odgovarajući, strukturirani način.

Detaljnije promatrano, moderni se razvoj programske potpore razlikuje od tradicionalnoga u četiri bitna aspekta: 1) uvode se inženjerski propisani postupci u proces oblikovanja programske potpore pri čemu se precizno definiraju i dokumentiraju faze procesa oblikovanja – tko radi, što radi i kada radi, 2) u oblikovanje se uvodi analiza i izbor stila arhitekture programske potpore te pripadni modeli (najčešće u obliku dijagrama) pogodni za formalnu analizu, 3) programski proizvod oblikuje se u manjim cjelinama -

komponentama, pogodnima za ponovnu uporabu, i 4) uz tradicijsko ispitivanje (testiranje) programske potpore uvode se formalne metode provjere, poglavito formalna verifikacija modela. Sve ovo dovodi do povećane pouzdanosti i ponovne iskoristivosti programske potpore kao i povećanja produktivnosti inženjerskog tima.

Ova skripta namijenjena je studentima 3. godine preddiplomskog studija na kolegiju Oblikovanju programske potpore (OPP), pri čemu ona pokriva približno trećinu gradiva kolegija i to prvu nastavnu cjelinu. Neki dijelovi skripte pokrivaju u manjoj mjeri i dijelove ostalih nastavnih cjelina. Nakana je autora da u ovoj skripti predoče programsko inženjerstvo s više teorijskog stajališta, što je bitan preduvjet razumijevanja njegovih praktičnih aspekata. Studenti kolegija OPP upućuju se na Sveučilišni priručnik "UML-dijagrami: Zbirka primjera i riješenih zadataka" za proučavanje modeliranja stvarnih programskih sustava korištenjem jezika UML, a pogotovo sustava temeljenih na objektno orijentiranoj paradigmi.

## 2. Definicije i temeljna pitanja programskog inženjerstva

### 2.1. Programska potpora i programsko inženjerstvo

Programsko inženjerstvo razlikuje se od programiranja odnosno kodiranja. Dok je kodiranje proces pisanja programskog koda u nekom određenom programskom jeziku, a programiranje prije svega mentalni proces predstavljanja i implementacije određenog algoritma najprije u simboličnom obliku (npr. pseudokodu) a potom kodiranjem, programsko inženjerstvo obuhvaća mnogo širi proces razvoja programske potpore.

Prije nego što se objasni što se podrazumijeva pod programskim inženjerstvom i koji je njegov značaj i prije nego što se definira sam pojam, najprije je potrebno pojasniti što je to programska potpora.



**Definicija 2.1.** *Programska potpora (engl. software) je neopipljiva komponenta računala koja uključuje sve podkomponente nužne za uspješno izvođenje računalnih instrukcija. Programska potpora može uključivati izvršne datoteke (programe), skripte, knjižnice, bazu podataka, datotečni sustav i druge podkomponente.*

Programska potpora može se razvijati za ciljanog kupca prema njegovim zahtjevima, za skupinu kupaca (npr. proračunske tablice za mobitele upravljane određenim operacijskim sustavom) ili za opće tržište (engl. COTS - Commercial Off The Shelf).

**Programski proizvod** (engl. *application*) ili **programski sustav** (engl. *software system*) je razvijena programska potpora. Programski proizvod može se izraditi oblikovanjem novih programa, (re)konfiguracijom postojećih programa, ili ponovnom uporabom postojećih komponenata programa.

U okviru programskog inženjerstva, pojam programske potpore u širem smislu uključuje i dokumentaciju (engl. *documentation*) [1].

**Dokumentacija** je pisani tekst u obliku jednog ili više dokumenata koji objašnjava kako radi programska potpora, koji su zahtjevi na rad programske potpore ili kako se ona treba pravilno koristiti. Dokumentacija programske potpore ima za cilj pružiti razne informacije o programskoj potpori raznim ljudima, u ovisnosti o ulozi koju oni imaju u odnosu na programski sustav. Primjeri cjelokupnih projektnih dokumentacija ostvareni na kolegiju Oblikovanje programske potpore prikazani su na slici 2.1.



**Definicija 2.2.** *Programsko inženjerstvo (engl. software engineering) je tehnička disciplina koja se bavi metodama i alatima za profesionalno oblikovanje i proizvodnju programske potpore uzimajući u obzir cjenovnu učinkovitost.*

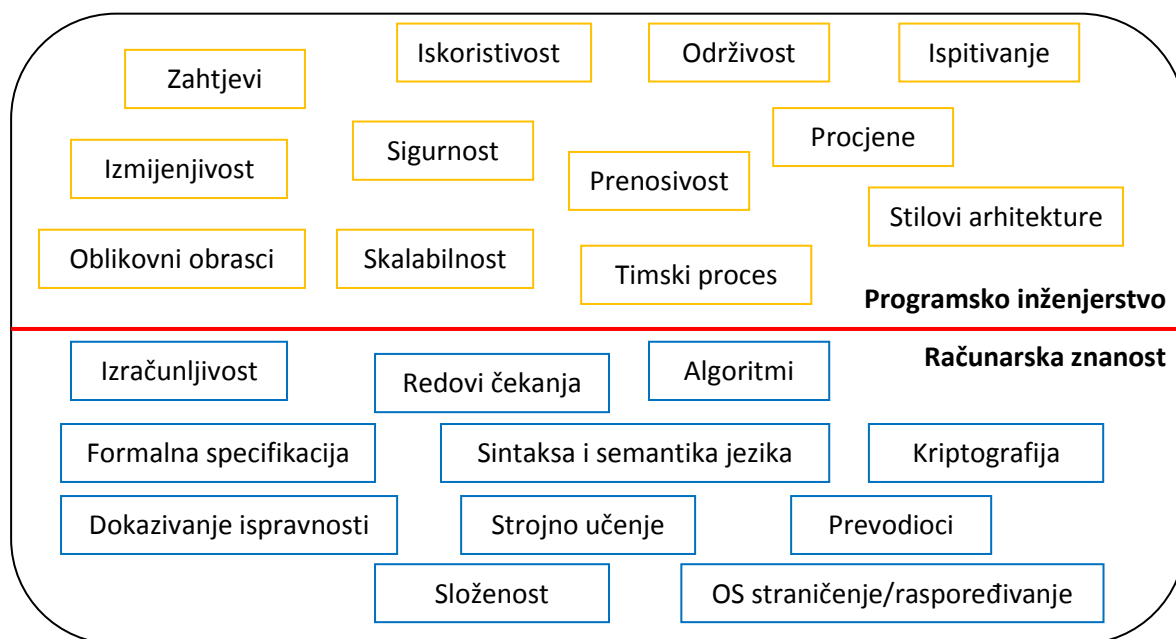


**Slika 2.1.** Dokumentacije projekata iz kolegija Oblikovanja programske potpore.

Zadatak koji pred inženjera postavlja disciplina programskog inženjerstva je taj da on mora prihvatiti sustavni i organizirani pristup procesu izrade programske potpore. Programski inženjer treba upotrebljavati prikladne alate i tehnike ovisno o problemu koji treba riješiti, uz ograničenja u procesu izrade i postojećim resursima. Zašto je inženjerski pristup pri izradi programske potpore vrlo bitan vidjeli smo u uvodu.

U programskom inženjerstvu detaljno se razmatraju teme kao što su: specifikacija zahtjeva, odabir stila arhitekture, ispitivanje programske potpore, sigurnost, prenosivost, održivost, iskoristivost, isplativost i druge.

Potrebno je naglasiti da programsko inženjerstvo nije **računarska znanost** (engl. *computer science*). Računarska znanost bavi se temeljima koji su nužni da bi se suštinski razumjeli problemi s kojima se programski inženjeri susreću u svakodnevnom radu, dok je programsko inženjerstvo orijentirano na praksu i rješavanje problema klijenata. Razlika između ta dva pojma predočena je na slici 2.2. Programsko inženjerstvo predstavlja sponu između cjelokupne teorije pokrivene računarskom znanošću i svakodnevnog svijeta u kojem se traže isplativa i učinkovita programska rješenja s kojima će raditi krajnji korisnici. Oni ne moraju znati niti kako je napravljena programska potpora, a još manje koja je teorija sve bila potrebna kako bi se neki programski proizvod ostvario.



**Slika 2.2.** Područja programskog inženjerstva i računarske znanosti.

Teorije u okviru računarske znanosti još uvijek nisu dostatne za ukupnu podlogu programskom inženjerstvu. To znači da osoba koja dobro zna i razumije računarsku znanost ne može na manje-više automatizirani način izraditi uporabljivi programski proizvod. Između računarske znanosti i programskog proizvoda postoji jaz koji se prekriva raznim tehnikama programskog inženjerstva. Pri tome ne postoji definiran jedinstven način kako doći do uporabljivog programskog proizvoda. Područje blisko području programskog inženjerstva je inženjerstvo računalnih sustava (engl. *computer system engineering*) ili kraće, **računalno inženjerstvo** (engl. *computer engineering*).



**Definicija 2.3.** Računalno inženjerstvo je širi pojam od programskog inženjerstva budući da se to područje bavi svim aspektima sustava zasnovanima na računalima (inženjerstvo sklopovlja, programsko inženjerstvo i inženjerstvo procesa). Programsko inženjerstvo, kao dio inženjerstva računalnih sustava, usredotočeno je na razvoj programske infrastrukture i na upravljanje, primjenu i rukovanjem podacima u računalnom sustavu.

Inženjeri računalnih sustava su često uključeni u specificiranje sustava, oblikovanje arhitekture, integraciju sustava i postavljanje u korisničko okruženje. U okviru kolegija Oblikovanje programske potpore bit ćemo usredotočeni na programsko inženjerstvo, dok se ostala potpodručja računalnog inženjerstva obrađuju tek povremeno.

## 2.2. Temeljna pitanja programskog inženjerstva

S čime se programsko inženjerstvo sve bavi, odnosno koja su to područja koja ono obrađuje? Na sljedećem popisu dan je pregled nekih temeljnih pitanja koja zanimaju

programsko inženjerstvo. Ovaj popis nije iscrpan, ali pokriva većinu pitanja koja će nas dalje zanimati.

- 1) Što je to i kako izgleda proces programskog inženjerstva?
- 2) Što je to i koji su sve modeli procesa programskog inženjerstva?
- 3) Što su to metode programskog inženjerstva?
- 4) Što je to CASE (engl. *computer-aided software engineering*)?
- 5) Koje su značajke dobrog programskog proizvoda?
- 6) Kakva je struktura cijene (troška) u programskom inženjerstvu?
- 7) Koje su osnovne poteškoće i izazovi u programskom inženjerstvu?
- 8) Koje vrste programskih projekata postoje i koje su njihove značajke?
- 9) Što je to profesionalna i etička odgovornost u programskom inženjerstvu?

### 2.2.1. Opis procesa programskog inženjerstva



**Definicija 2.4.** *Proces programskog inženjerstva je skup aktivnosti čiji cilj je razvoj i evolucija programskog proizvoda. U tom procesu bitnu komponentu čini timski rad pomoću kojeg se učinkovito izgrađuje programski proizvod. Proces programskog inženjerstva je stvarni tijek aktivnosti koji se odvija tijekom izrade programskog proizvoda.*

Svaki proces programskog inženjerstva sastoji se od četiri temeljne **generičke aktivnosti**:

- **specifikacije,**
- **oblikovanja i implementacije,**
- **validacije i verifikacije,**
- **evolucije programske potpore.**

Svaka od generičkih aktivnosti procesa programskog inženjerstva razrađena je detaljnije u poglavlju 3.

### 2.2.2. Opis modela procesa programskog inženjerstva



**Definicija 2.5.** *Model procesa programskog inženjerstva je pojednostavljeno predstavljanje procesa programskog inženjerstva iz određene perspektive (pogleda na proces). Postoji više predloženih modela koji nastoje jasnije predstaviti stvarni proces programskog inženjerstva: vodopadni, evolucijski, komponentni, modelno-usmjereni razvoj, agilni razvoj.*



Perspektive pogleda na proces mogu uključivati: uloge i akcije - tko što čini, tijek podataka, tijek aktivnosti, itd. Svi modeli procesa programskog inženjerstva objašnjeni su detaljnije u poglavlju 5.

### 2.2.3. Metode programskog inženjerstva

Metode programskog inženjerstva su organizirani pristupi povezivanju aktivnosti u oblikovanju i implementaciji programske potpore. Metode uključuju:

- izbor modela sustava (npr. objektni model, model toka podataka, model stroja s konačnim brojem stanja),
- notacije (označavanja) modela,
- pravila koje vrijede u cijelom sustavu (npr. svaki entitet u modelu sustava mora imati svoj naziv),
- preporuke dobre inženjerske prakse pri oblikovanju (npr. jedan razred bi idealno trebao imati samo jednu odgovornost za koju je zadužen) i
- smjernice pri opisu procesu (npr. najprije dokumentirati attribute razreda, potom metode).

### 2.2.4. CASE



**Definicija 2.6.** CASE su programski proizvodi ili alati veće ili manje složenosti namijenjeni automatiziranoj podršci generičkim aktivnostima u procesu programskog inženjerstva.

CASE proizvodi često podupiru samo jednu, određenu aktivnost u razvoju programskog proizvoda. Pri tome razlikujemo više CASE proizvode (engl. *upper CASE*), koji podupiru rane aktivnosti kao što su analiza zahtjeva i oblikovanje arhitekture i niže CASE (engl. *lower CASE*) proizvode, koji podupiru kasnije aktivnosti kao što su kodiranje i ispitivanje. Detaljnije se CASE proizvodi obrađuju u 6. poglavlju.

### 2.2.5. Značajke dobrog programskog proizvoda

Temeljna značajka dobrog programskog proizvoda je da proizvod **mora osigurati traženu funkcionalnost i performanse**. Osim toga, ostale značajke su: prihvatljivost za korisnika, pouzdanost i moгуćnost laganog održavanja. Prihvatljivost obuhvaća to da programski proizvod mora biti razumljiv, koristan i kompatibilan s ostalim sustavima na strani korisnika. Pouzdanost govori o tome da korisnik mora vjerovati da sustav ispravno funkcionira. Onaj sustav koji jednom zakaže značajno smanjuje povjerenje korisnika što se treba izbjeći pod

svaku cijenu. Održavanje uključuje naknadno ispravljanje pogrešaka i evoluciju proizvoda sukladno izmijenjenim i proširenim zahtjevima korisnika.

Naravno, svaki **dionik** (engl. *stakeholder*) ima svoja očekivanja i poglede na to što znači dobar proizvod. Tako je za kupca dobar programski proizvod onaj koji rješava problem uz prihvatljivu cijenu. Za krajnjeg korisnika, to je onaj proizvod koji se lagano nauči i prihvaća i zaista pomaže da se posao lakše obavi. Za osobu koja razvija i oblikuje, to je onaj proizvod koji se lagano oblikuje i održava i koji se da ponovno iskoristiti.

#### **2.2.6. Struktura troška programskog proizvoda**

Trošak programskog proizvoda je njegov bitan aspekt koji određuje je li njegov razvoj isplativ ili ne. U pravilu, programski proizvod razvija se tako da njegova korist bude veća od svih troškova vezanih uz razvoj i održavanje. Cijena programske potpore sastoji se od cijene specifikacije, oblikovanja, ispitivanja i održavanja (evolucije). Cijena uvelike ovisi o tipu programskog sustava, zahtjevima, traženim performansama i traženoj razini pouzdanosti. U svemu tome potrebno je biti svjestan da postoji tržišno natjecanje u proizvodnji programske potpore, što znači da proizvod mora biti konkurentan i po pitanju cijene i po pitanju brzine izlaska na tržište, uz zadržavanje performansi i pouzdanosti.

Za programske proizvode koji imaju dugi vijek trajanja (10 i više godina), kao što su razni kontrolni sustavi, **trošak evolucije** (promjene, prilagodbe i održavanja) je često višestruko veći od troška izvornog razvoja. Manji poslovni programski proizvodi imaju često kraći rok trajanja i posljedično manji trošak evolucije.

Prilikom razvoja programskog proizvoda, postoje česta proturječja. Primjerice, povećanje učinkovitosti specijalizacijom čini programski proizvod manje razumljivim i može smanjiti mogućnost održavanja ili ponovnog korištenja. Povećanje lakoće korištenja (npr. uključivanje uputa tijekom rada) može smanjiti učinkovitost. Upravo stoga je nužno unaprijed postaviti razinu kakvoće, što je ključna inženjerska aktivnost. Programski proizvod mora biti dovoljno dobar. Pritom se rade stalni kompromisi i optimizacije ograničenih resursa. Dobra inženjerska praksa je da se izbjegava suvišan posao dodavanja funkcionalnosti koje korisnik nije tražio, jer se time štedi i vrijeme i novac.

#### **2.2.7. Poteškoće i izazovi u programskom inženjerstvu**

Prilikom razvoja programske potpore pojavljuju se razne poteškoće i izazovi. Promatraju se obično oni najčešći, odnosno karakteristični za sve procese programskog inženjerstva. To su poglavito:

- **heterogenost razvoja** (engl. *heterogeneity*),
- **ograničeno vrijeme isporuke** (engl. *delivery*),

- **povećanje povjerenja (engl. *trust*),**
- **česte izmjene zahtjeva (engl. *requirements changes*) i**
- **složenost razvijenog programskog sustava (engl. *complexity*).**

Heterogenost razvoja znači da postoje različite tehnike i metode razvoja programske potpore za različite platforme i okoline izvođenja. Koju je tehniku i metodu najbolje za odabrati za dani problem stvar je iskustva. Ograničeno vrijeme isporuke je često diktirano od strane vodstva projekta na temelju zahtjeva klijenta. Potrebno je osigurati što kraći interval od zamisli do stavljanja gotovog, prihvatljivog programskog proizvoda u proizvodnju, odnosno na tržište. Povećanje pouzdanosti sustava ostvaruje se iscrpnim i automatiziranim načinima ispitivanja kao i uporabom matematičkih, formalnih metoda dokaza u pojedinim slučajevima. Česte izmjene sustava karakteristične su za poslovne projekte u novije vrijeme, gdje je korisnik postao zahtjevniji i ažurniji. Kako reagirati na iznenadne promjene zahtjeva korisnika stvar je često inicijalnog dogovora između vodstva projekta i klijenta, ali inženjeri se čestu nađu u situaciji da moraju napraviti puno izmjena u kratkom vremenskom periodu. Složenost programskog sustava ne mora biti problem ako je arhitektura dobro osmišljena i ako su korištena dobra inženjerska praksa u oblikovanju i implementaciji.

#### **2.2.8. Vrste programskih projekata**

Većina programskih projekata je evolucijska i vezana je uz održavanje starijih programskih proizvoda koji su ostavljeni novim inženjerima u naslijeđe (engl. *legacy software*). Osim toga, postoje korektivni projekti, kojima je zadatak ukloniti uočene kvarove u programskom proizvodu. Adaptivni projekti su takvi projekti kod kojih se programski proizvod treba prilagoditi novonastalim okolnostima kod klijenta, npr. promjeni operacijskog sustava, sustava za upravljanje bazom podataka, kao i novim pravnim odredbama. Unapređujući ili aditivni projekti imaju za cilj dodavanje novih funkcionalnosti. Re-inženjerstvo programskog proizvoda provodi unutarnje izmjene kojima se olakšava održavanje. Sasvim novi projekti (engl. *green field*) su uglavnom u manjini. Integrativni projekti imaju za cilj oblikovanje novog okruženja iz **postojećih programskih komponenata i radnih okvira** (engl. *framework*). Konačno, jedan dio projekata su hibridni, kod kojih se npr. provode i korekcije i adaptacije i unapređivanje programske potpore.

#### **2.2.9. Etička odgovornost**

Vrlo značajna osobina kvalitetnog programskog inženjera je visoka **profesionalna i etička odgovornost**. Programski inženjer mora se ponašati profesionalno korektno i etički odgovorno. Etičko ponašanje je više od pukog pridržavanja zakona. Ono uključuje povjerljivost prema poslodavcu i klijentu, prihvaćanje posla samo u okviru kompetencija, poštivanje prava intelektualnog vlasništva, ne zlouporabu računalnih sustava (širenje

virusa, igranje za vrijeme rada (predavanja!), neovlašteno upadanje u druge sustave i sl.) i slijedenje smjernica etičkog kodeksa prema udrugama IEEE/ACM [2]. Od smjernica koje tamo stoje, spomenut će se samo neke: odbiti mito i sve njegove oblike, izbjegavati stvarne ili uočene sukobe interesa kada god je to moguće i otkriti ih interesnim stranama ukoliko postoje, jednako se ophoditi prema svim osobama bez obzira na rasu, vjeru, spol, invaliditet, godine ili nacionalnost, pomagati kolegama i suradnicima u njihovom profesionalnom razvoju i podržavati ih da se pridržavaju ovog koda etičnosti.

### 3. Životni ciklus programske potpore

Svaka programska potpora ima svoj životni ciklus. Intuitivno, to je put koji određena programska potpora treba proći od svojeg početka, najprije kao ideja u umovima klijenata koji žele poboljšati nešto u svom poslovanju, zatim kao zapis te ideje, od manje formalnog prema formalnijem, što dovodi do oblikovanja, implementacije u programskom kodu i ispitivanja, sve do evolucije kroz više inačica programske potpore. Tijekom svojeg životnog ciklusa, programska potpora prolazi kroz procese nastanka, rada i održavanja. Ukupno promatrano, bez obzira o kojem se modelu tih procesa radi, moguće je razlikovati nekoliko zajedničkih, generičkih aktivnosti tijekom životnog ciklusa. Prema Sommervilleu [1], to su aktivnosti: **1) specifikacije, 2) oblikovanja i implementacije (ili kraće: razvoja), 3) validacije i verifikacije i 4) evolucije programske potpore.** U nastavku se opisuju detaljnije svaka od tih generičkih aktivnosti.

#### 3.1. Specifikacija programske potpore

Zadatak generičke aktivnosti specifikacije je da se temeljem analize zahtjeva klijenata odredi što sustav treba činiti i koja su ograničenja u razvoju.



Specifikacija programske potpore određuje se procesom inženjerstva zahtjeva (engl. *requirements engineering*), koje je zasebno područje programskog inženjerstva.

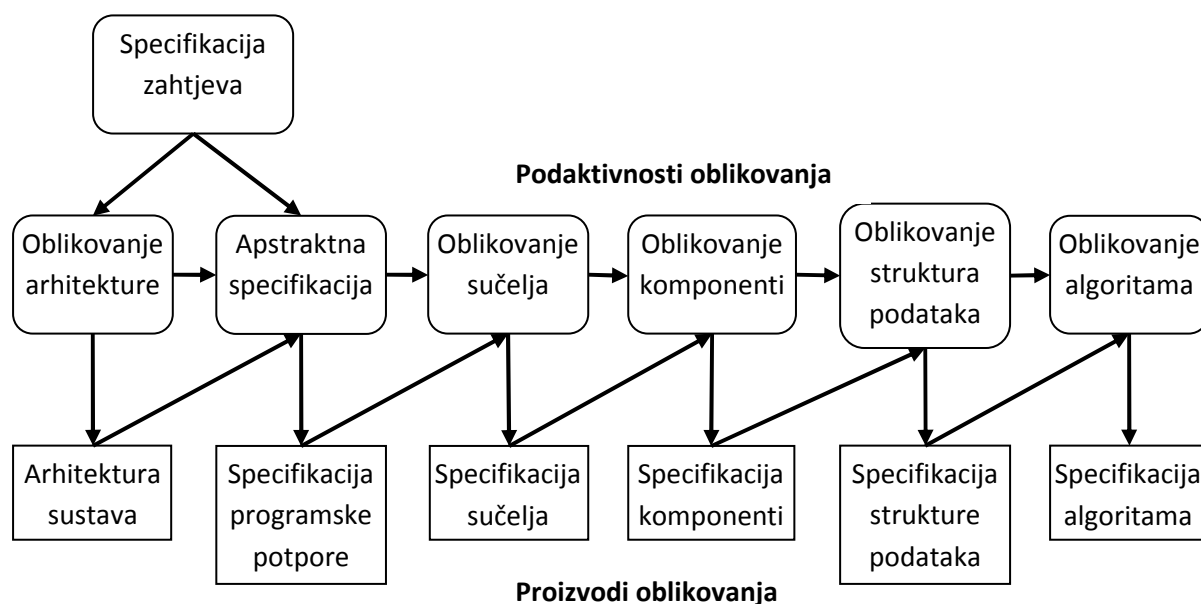
Zbog svoje složenosti i značaja, inženjerstvo zahtjeva se detaljnije razrađuje u zasebnom, četvrtom poglavlju ove skripte.

U kontekstu životnog ciklusa programske potpore, važno je napomenuti da je specifikacija aktivnost koja započinje studijom izvedivosti zamisli ili idejnog projekta klijenta, a završava dokumentom specifikacije programske potpore, u kojem se navode potrebne usluge i ograničenja u radu i razvoju programske potpore koju se tek treba izraditi. Budući da su zahtjevi klijenata često podložni promjenama, posebno se razmatra upravljanje promjenama u zahtjevima nakon što je dokument specifikacije već izrađen.

Pravilna specifikacija programske potpore je posebno kritična aktivnost u njenom životnom ciklusu, budući da pogreške inženjera tijekom ove aktivnosti neizbježno dovode do daljnjih problema tijekom aktivnosti oblikovanja i implementacije.

#### 3.2. Oblikovanje i implementacija programske potpore

Oblikovanje i implementacija predstavljaju generičku aktivnost razvoja programske potpore u užem smislu. Aktivnost oblikovanja i implementacija ima za zadatak uzeti specifikaciju programske potpore i pretvoriti je u stvarni proizvod, često u smislu izvršnog koda. Ako se promatra detaljnije, ova aktivnost sastoji se od dvije međusobno isprepletene faze:



**Slika 3.1.** Opći model procesa oblikovanja programske potpore, prilagođeno iz [1].

**oblikovanje strukture sustava** koji ispravno modelira specifikaciju (izbor i modeliranje arhitekture, oblikovanje programske potpore u užem smislu) i **implementacije**, koja preslikava taj ustroj u izvršni kod.

### 3.2.1. Oblikovanje programske potpore

Prva faza, oblikovanje programske potpore, sastoji se od više podfaza unutar kojih se oblikuje sve od sučelja, komponenata, struktura podataka do algoritama. Opći model procesa oblikovanja programske potpore prikazan je na slici 3.1. Na temelju dokumenta specifikacije sustava kreće se u oblikovanje arhitekture, tijekom kojeg se utvrđuju i dokumentiraju podsustavi i njihove veze, dakle sve ono što će činiti arhitekturu konačnog sustava. Također, u tom koraku odabire se stil arhitekture sustava (npr. objektno orijentirani, cjevovodi i filteri, arhitektura zasnovana na događajima, repozitorij podataka). Izbor arhitekture nažalost nije jednoznačan niti formalno propisan, već se najčešće temelji na neformalnoj analizi i dobroj inženjerskoj praksi. Na temelju odabrane arhitekture slijedi sustavni pristup oblikovanja programskog proizvoda. Arhitektura sustava trebala bi biti dokumentirana skupom modela (najčešće grafičkih dijagrama).

Tijekom oblikovanja arhitekture pristupa se apstraktnoj specifikaciji, gdje se za svaki podsustav navodi ukratko usluge koje će davati i ograničenja pod kojima treba raditi. Oblikovanje sučelja definira za svaki podsustav njegovo sučelje prema ostalim podsustavima. Definicija sučelja mora biti jednoznačna i mora omogućavati funkcioniranje komunikacije bez znanja o unutarnjim dijelovima podsustava. Usluge podsustava raspodjeljuju se među komponentama koje ga čine. Najčešće su komponente programski moduli kod proceduralne paradigme ili razredi kod objektno orijentirane paradigme. U

ovom koraku oblikuju se i sučelja komponenata prema ostalim komponentama istog podsustava. Dobra je inženjerska praksa unaprijed, prije implementacije, definirati sve ili većinu struktura podataka (liste, polja, tablice raspršenog adresiranja i sl.) koje će komponenta koristiti. Konačno, oblikuju se pseudokodovi algoritama koji ostvaruju određenu uslugu. Zadnje dvije podfaze koji put se odgađaju do faze implementacije.

### 3.2.2. Implementacija programske potpore

Implementacija je preslikavanje dokumentiranog oblikovanja (odabrane arhitekture) u konačni program. Programiranje je osobna aktivnost koja podrazumjeva naglašenu mentalnu aktivnost – **ne postoji definirani generički proces programiranja**. Neki programeri počnu s onim komponentama koje bolje razumiju pa nastavljaju s onima koje razumiju slabije. Drugi postupaju obrnuto. Neki put je programiranje orijentirano najprije prema kritičnim komponentama sustava, a one manje značajne se programiraju kasnije. Programeri tijekom implementacije izvode i ispitivanje (testiranje) programskog koda koji su napisali kako bi otkrili moguće kvarove. Nakon što su otkrili da kvar postoji, potrebno ga je locirati i otkloniti (engl. *debugging*). **Lociranje i otklanjanje kvarova** je u novije vrijeme jednako tako dio aktivnosti implementacije kao i aktivnosti validacije i verifikacije.

Važno je napomenuti da se temeljni dijagram oblikovanja programske potpore na različit način ostvaruje ovisno o modelu procesa programskog inženjerstva. Tako, primjerice, pri korištenju agilnih metoda, svi proizvodi nakon dokumentirane arhitekture sustava nisu više dokumenti već su to napisani dijelovi koda. Sustav se inkrementalno poboljšava gotovo isključivo generiranjem novog koda, bez opsežne dokumentacije. Kod strukturnih metoda kao što je to Rational Unified Process (RUP), pojedini proizvodi dokumentacije dani su grafičkim prikazima u obliku UML dijagrama. Također, često je podržano automatsko generiranje rudimentarnog koda na temelju takvih grafičkih prikaza.

### 3.3. Validacija i verifikacija programske potpore

Validacija i verifikacija programske potpore trebaju pokazati da sustav odgovara specifikaciji i da u potpunosti zadovoljava zahtjeve klijenta i krajnjih korisnika.

#### 3.3.1. Opis validacije i verifikacije



Validacija odgovara na pitanje: "Gradimo li pravi sustav?", dok verifikacija odgovara na pitanje: "Gradimo li sustav na ispravan način?".


Drugim riječima, validacijom se utvrđuje odgovara li sustav koji gradimo onoj specifikaciji koju je korisnik zadao. Potrebno je validirati da je sustav koji razvijamo upravo onakav kakav je korisnik zamislio i zatražio od nas. Verifikacija se pak bavi time postoje li kvarovi u našem sustavu koji su uvedeni pogreškom člana razvojnog tima tijekom procesa razvoja.

I validacija i verifikacija provode se ispitivanjem (testiranjem) sustava. U oba slučaja, provodi se inspekcija i pregled napravljenoga tijekom svih faza razvoja programske potpore, počevši od dokumenta specifikacije zahtjeva. Međutim, pravu validaciju zahtjeva moguće je napraviti tek nakon što postoji gotova implementacija nekog sustava ili barem njegovog većeg dijela.

Validaciju, koju se često poistovjećuje s terminom ispitivanja prihvatljivosti sustava (engl. *acceptance testing*), obično provodi tim od strane klijenta zajedno s razvojnim timom. Verifikacija se većinom provodi tijekom razvoja dijelova sustava od strane članova razvojnog tima. Verifikacija uključuje provjeru odsustva pogrešaka, što je zasnovano na više ili manje formalnim metodama. Poželjno je, posebno za kritične dijelove sustave, verifikaciju zasnovati na formalnim matematičkim i logičkim metodama (formalna specifikacija, formalna sinteza i formalna verifikacija) koje garantiraju ispravnost programa uz određenu matematičku izvjesnost.

Ispitivanje programske potpore nije samo pronalaženje i otklanjanje pogrešaka. Cilj ispitivanja je utvrditi da program funkcionira ispravno, odnosno da nema pogrešaka. Međutim, prema Dijkstri (1972.), nepostojanje pogreške je vrlo teško utvrditi, već se samo učinkovito može utvrditi njezino postojanje. Stoga ispitivanje u užem smislu znači da se uspoređuju stvarni rezultati s postavljenim i očekivanim zahtjevima na program. Program treba raditi ispravno, što znači da stvarni rezultati trebaju odgovarati onima unaprijed zadanim. Bez unaprijed zadane specifikacije nema niti ispitivanja!

### 3.3.2. Pojmovi vezani uz ispitivanje programske potpore

 **Definicija 3.1.** *Ispitni ili testni slučaj (engl. test case) u užem smislu je uređeni par  $(I,O)$ , gdje je  $I$  ulazni podatak, a  $O$  očekivani izlazni podatak iz programa, zabilježen prije provođenja ispitivanja.*

Osim ispitnog slučaja, potrebno je unaprijed poznavati i kriterij uspješnog prolaska ispitnog slučaja kroz ispitivanja. Najčešći kriterij je da izlazni podatak potpuno odgovara stvarno dobivenom rezultatu provođenja ispitivanja. Institut inženjera elektrike i elektronike (IEEE) definira test (ispit) kao jedan ili više ispitnih slučajeva, a ispitivanje kao proces analize programskog koda sa svrhom pronalaska razlike između postojećeg i zahtijevanog stanja te vrednovanja svojstava programa.

Uz iznimku vrlo malih programa, programski proizvodi ne mogu se ispitivati kao cjelovite, monolitne jedinice. Stoga se ispitivanje programske potpore u praksi provodi tijekom više faza, koje se mogu ugrubo podijeliti na:

1. ispitivanje komponenti,
2. ispitivanja integracije,



3. ispitivanje sustava,
4. ispitivanje prihvatljivosti,
5. ispitivanje instalacije.

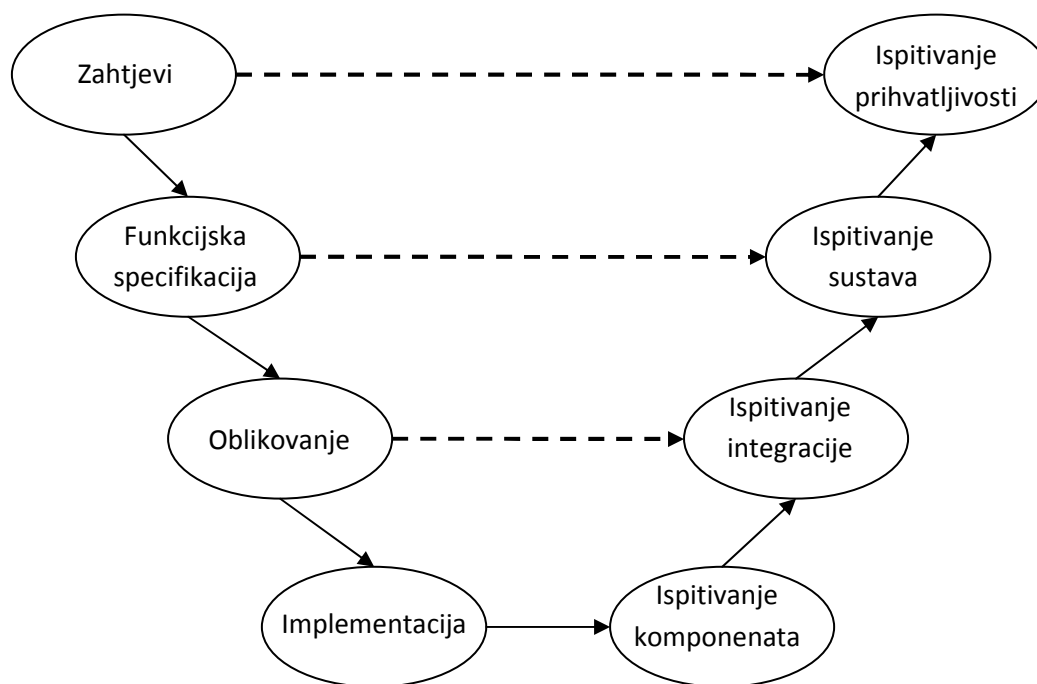
Ispitivanje komponenti (engl. *unit testing*, *component testing*) uključuje ispitivanje pojedinih programskih modula, razreda, funkcija ili drugih pojedinačnih komponenata sustava. Ispitivanje komponenti obično provodi programer koji je komponentu i izradio, uz direktan uvid u programski kod. U tu svrhu koristi se razvijenim ispitnim slučajevima. U mnogim slučajevima, ispitivanje komponenata moguće je i automatizirati korištenjem odgovarajućih CASE alata ili čak i jednostavnih skripti.

Pojedine komponente integriraju se u sve složenije podsustave i ispituju se na različite načine, ovisno o povezanosti komponenata, prioritetu komponenata i načinu izgradnje sustava. Najčešće se ispituju sučelja skupine komponenata kako bi se otkrili mogući kvarovi. Cilj je osigurati zajednički rad grupe komponenti prema specifikaciji zahtjeva. Osnovni problem pri integracijskom ispitivanju predstavlja lokalizacija pogrešaka zbog složenih interakcija komponenata. Ipak, važno je integrirati komponente u podsustave kako bi se ispitala njihova međusobna suradnja prije nego što se razmatra sustav u cjelini.

Tijekom ispitivanja čitavog sustava, provodi se **funkcijsko ispitivanje** i **ispitivanje performansi** na razini cijelog sustava od strane razvojnog tima ili posebno zaduženog tima za ispitivanje sustava. Tim provjerava odgovara li izgrađeni sustav u smislu funkcionalnosti i zadanih ograničenja i kvaliteta onome što je klijent naručio. Najčešće se ispitivanje sustava provodi prateći korisničke scenarije za uporabu sustava, koji bi svi trebali uspješno proći. Ponekad prilikom ispitivanja sustava u cjelini izranjaju iznenađujuća svojstva koja nisu bila vidljiva na razini podsustava. Stoga je ispitivanje cijelog sustava vrlo bitna stavka u ispitivanju prije nego što se sustav da na ispitivanje klijentima i krajnjim korisnicima.

Ispitivanje prihvatljivosti provodi ispitni tim koji se najčešće sastoji od članova od strane klijenta i od razvojne strane. Ispitni tim validira sustav, što znači da utvrđuje je li sustav prihvatljiv za klijenta. Važno je unaprijed, prilikom specifikacije zahtjeva, jednoznačno utvrditi što znači da je sustav prihvatljiv za klijenta, budući da oko toga zna biti prepirke u ovoj konačnoj fazi, što otežava daljnji razvoj i povećava cijenu dorada. Često se ispitivanje prihvatljivosti naziva i **alfa ispitivanje**, pri čemu se podrazumjeva ispitivanje unutar razvojne tvrtke.

Konačno, ispitivanje instalacije je ispitivanje programske potpore koja se tada nalazi na klijentskoj strani odnosno na strani krajnjeg korisnika. Pritom je okruženje upravo ono za koje je programski proizvod i izrađen. Korisnik provodi ispitivanje sustava tako što ga koristi u svakodnevnoj praksi tijekom određenog vremena. Nakon što je utvrđeno da je programski proizvod uspješno prošao ispitivanje instalacije, može ga se bez ograničenja koristiti. Često se ispitivanje instalacije naziva i **beta ispitivanje**.



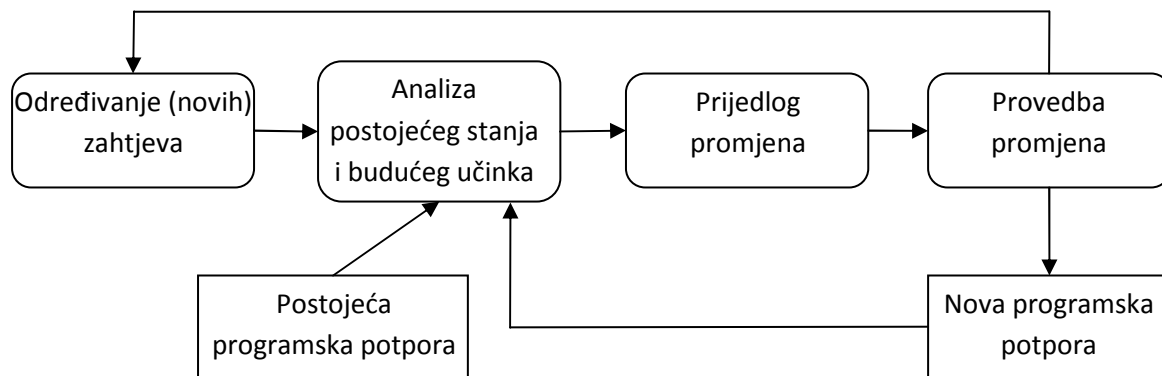
**Slika 3.2.** V-model ispitivanja programske potpore.

Na slici 3.2. prikazan je klasični **V-model** ispitivanja koji podrazumijeva da se sustav najprije izrađuje od specifikacije do krajnjeg koda i potom ispituje od razine komponenta do ispitivanja prihvatljivosti. V-model u novije vrijeme češće se u praksi zamjenjuje s W-modelom (dualnim V modelom, engl. *Dual Vee model*), koji podrazumijeva da se ispitivanje provodi u paraleli sa svakom aktivnosti tijekom životnog ciklusa programske potpore, od specifikacije do izgradnje čitavog sustava. Osnova tog modela je da se sve što napiše ili implementira ujedno i što prije provjeri, koliko god je to moguće iscrpno.

### 3.4. Evolucija programske potpore

Programski proizvod je inherentno fleksibilan i može se mijenjati, za razliku od sklopovlja, kod kojih je promjena teža i skuplja. Kako se tijekom vremena mijenjaju zahtjevi na sustav (zbog promjene poslovnog procesa ili okruženja), programski proizvod koji podupire poslovni proces u pravilu se mora mijenjati. Promjene programskog proizvoda tijekom vremena oslabljuju strukturu njegove izvorne arhitekture što dovodi do smanjene učinkovitosti pri provođenju novih promjena. S vremenom, strukturu je potrebno ponovno oblikovati, uz moguće značajne izmjene dijelova koda, kako bi se zadržala kohezija i spriječilo daljnje rasipanje funkcionalnosti. U protivnom, programski proizvod pomalo gubi utjecaj i na kraju nestaje.

Povijesno promatrano, uvijek je postojala granica između razvoja programske potpore i njezine evolucije, koja je uglavnom podrazumijevala **održavanje sustava**. Inženjeri su



**Slika 3.3.** Evolucija programske potpore.

tradicionalno smatrali da je razvoj programske potpore kreativna aktivnost, dok su održavanje smatrali dosadnim i nezanimljivim. U novije vrijeme, granica između razvoja i evolucije je sve manje značajna, jer je sve manje potpuno novih programskih projekata. U tom pogledu, više ima smisla promatrati razvoj i održavanje programske potpore kao kontinuum, odnosno kao trajnu evoluciju. Zanimljivo je da razvoj programske potpore ima često mnogo manju cijenu od održavanja, koje je kod dugovječnih sustava (10 i više godina) najznačajnija financijska stavka.

Na slici 3.3. prikazan je opći shematski dijagram evolucijske programske potpore. Bitno je naglasiti da se programska potpora stalno mijenja kako dolaze novi zahtjevi od strane korisnika. Zahtjevi korisnika, a time i prijedlozi promjena, mogu ići u smjeru:

- nove funkcionalnosti (engl. *system enhancement*),
- izmjenama postojeće funkcionalnosti radi prilagodbe novonastalim okolnostima (engl. *platform adaptation*) ili
- ispravci zbog uočenih kvarova programske potpore (engl. *fault repair*).

Prije provedbe promjena, nužno je da se sagleda trenutačna funkcionalnost i mogući učinci prilikom provođenja promjena. Ako takva analiza pokaže da bi promjene dovele do većih problema nego što bi se dobilo na funkcionalnosti, od nekih promjena može se i odustati.

Programska potpora može evoluirati u smjeru različitih proizvodnih linija i različitih inačica. **Linije proizvoda** (ili porodica proizvoda) su skupovi svih proizvoda izrađenih na zajedničkoj osnovnoj tehnologiji. Različiti proizvodi u liniji proizvoda imaju različite ostvarene značajke kako bi zadovoljili različite segmente tržišta. Tako, naprimjer, proizvodne linije nekog programskog proizvoda mogu nositi oznaku “demo”, “pro”, “lite”, “enterprise” i sl. Inačice su vezane uz svaku od tih proizvodnih linija pri čemu nije nužno da inačice budu jednake između različitih proizvodnih linija. Evolucija osnovnog programskog proizvoda tako može ići u raznim smjerovima, pri čemu osnovicu za razvoj različitih linija čine najčešće jedan, a koji put i više specijaliziranih radnih okvira.

## 4. Inženjerstvo zahtjeva

Inženjerstvo zahtjeva je proces izrade specifikacije programske potpore. To je prva generička aktivnost tijekom svakog procesa programskog inženjerstva. Zahtjevi opisuju što programski sustav treba raditi kao i ograničenja u njegovom radu. U procesu izrade specifikacije programske potpore postoje postupci pronalaženja, analiziranja, dokumentiranja i provjere funkcija i ograničenja u uporabi. Zahtjevi sami za sebe su kraći ili duži dokumenti koji specificiraju usluge sustava i ograničenja u uporabi.

### 4.1. Višedimenzijska klasifikacija zahtjeva

Pojam zahtjeva ne koristi se konzistentno u industrijskoj primjeni. Negdje su zahtjevi apstraktne izjave o usluzi koju bi programski sustav trebao ponuditi ili o njegovim ograničenjima. Drugdje se tu podrazumijeva detaljna, često formalna definicija funkcije programskog proizvoda.

Zašto dolazi do takvih razlika u shvaćanju zahtjeva možda najbolje ilustrira sljedeći primjer [3].

**Primjer 4.1.** Ako neka tvrtka želi ponuditi natječaj o izradi velikog, složenog programskog projekta na otvoreno tržište, tada ona nudi apstraktnu specifikaciju koja u najopćenitijim crtama opisuje funkciju konačnog programskog proizvoda. Tada se na natječaj javljaju ponuđači koji predlažu načine na koje će riješiti zadani problem. Nakon što pregleda prijedloge ponuđača, tvrtka odabire izvođača projekta. Zatim taj izvođač piše detaljniju specifikaciju sustava koju tvrtka vrednuje prije nego što se potpisuje ugovor i kreće u izradu projekta.

Obje vrste specifikacije, i ona apstraktna, i ona detaljnija, smatraju se specifikacijom zahtjeva.

#### 4.1.1. Podjela zahtjeva prema razini detalja

Mnogi projekti zapadnu u probleme kad se razine zahtjeva međusobno miješaju. Upravo stoga je vrlo bitno klasificirati zahtjeve prema razini detalja i prema sadržaju. Tako se prema razini detalja razlikuju:

- **Korisničke zahtjeve (engl. *user requirements*)**
- **Zahtjeve sustava (engl. *system requirements*)**
- **Specifikaciju programske potpore (engl. *software specification*)**

Korisnički zahtjevi su oni zahtjevi najviše razine apstrakcije. Njih najčešće zadaje korisnik i to uglavnom u neformalnom, nestrukturiranom obliku. Tako se oni pišu u prirodnom jeziku

#### Korisnički zahtjevi

1. Programski sustav za ministarstvo zdravlja generirat će mjesečna izvješća za upravu u kojima će predložiti cijenu lijekova koji se propisuju za svaku kliniku tijekom tog mjeseca.

#### Zahtjevi sustava

- 1.1 Na zadnji radni dan u mjesecu, generirat će se sažetak o propisanim lijekovima, njihovoj cijeni i klinikama koje su ih propisale.
- 1.2 Sustav će automatski generirati izvješće spremno za ispis nakon 17:30 na zadnji radni dan u mjesecu.
- 1.3 Izvješće će biti napravljeno i za svaku kliniku posebno i popisat će pojedinačne nazive lijekova, ukupan broj recepata, broj propisanih doza i ukupnu cijenu propisanih lijekova.
- 1.4 Ako su lijekovi dostupni u različitim dozama (npr. 10 mg, 20 mg), zasebna izvješća će se napraviti za svaku postojeću dozu.
- 1.5 Pristup svim izvješćima o cijenama bit će ograničen na sve ovjerene korisnike koji se nalaze na kontrolnoj listi s razinom pristupa: "uprava".

**Slika 4.1.** Korisnički zahtjevi i zahtjevi sustava, prilagođeno iz [3].

i crtaju jednostavnim grafičkim dijagramima. Često iz njih nije jasno kako će točno sustav funkcionirati niti koji će biti njegovi dijelovi. Korisnički zahtjevi obično dolaze u okviru ponude za izradu programskog produkta, kao u prethodno primjeru 4.1.

Zahtjevi sustava su vrlo detaljna specifikacija o funkcionalnosti i ograničenjima programske potpore. Pišu se strukturiranim prirodnim jezikom, posebnim jezicima za oblikovanje sustava, dijagramima i matematičkom notacijom. Primjer korisničkih zahtjeva i zahtjeva sustava dan je na slici 4.1. Može se primijetiti da su zahtjevi sustava mnogo detaljniji u svojoj razradi funkcionalnosti i ograničenja od korisničkih zahtjeva. Za razliku od korisničkih zahtjeva koji se ne zamaraju načinom izvedbe funkcionalnosti, zahtjevi sustava definiraju što programski proizvod treba raditi na način da se iz tako strukturiranih zahtjeva već nazire njegova buduća arhitektura.

Specifikacija programske potpore je najdetaljniji opis i objedinjuje korisničke zahtjeve i zahtjeve sustava. Ona također može dodatno obuhvaćati tehničku specifikaciju koja opisuje detaljne zahtjeve na arhitekturu i programske dijelove.

Svaku razinu zahtjeva čitaju i dorađuju različiti dionici na programskom projektu. U tablici 4.1. predložen je opis toga koji dionik čita ili dorađuje zahtjeve, ovisno o razini detalja.

#### 4.1.2. Podjela zahtjeva prema sadržaju

Prema sadržaju, zahtjeve se može podijeliti na:

- **funkcionalne zahtjeve (engl. *functional requirements*),**

**Tablica 4.1.** Dionici koji čitaju ili dorađuju zahtjeve u ovisnosti o razini detalja zahtjeva.

Razina zahtjeva	Dionici
Korisnički zahtjevi	Klijentski rukovoditelji i menadžeri Klijentski inženjeri Krajnji korisnici sustava Rukovoditelji za pisanje ugovora Specijalisti za oblikovanje sustava - arhitekti
Zahtjevi sustava	Klijentski inženjeri Krajnji korisnici sustava Specijalisti za oblikovanje sustava - arhitekti Specijalisti za razvoj programske potpore
Specifikacija programske potpore	Klijentski inženjeri (možda) Specijalisti za oblikovanje sustava - arhitekti Specijalisti za razvoj programske potpore

- **Nefunkcionalne ili ostale zahtjeve (engl. *non-functional, other requirements*),**
- **Zahtjeve domene primjene (engl. *domain requirements*).**

Funkcionalni zahtjevi su izjave o uslugama koje programski proizvod mora pružati, kako će sustav reagirati na određeni ulazni poticaj, te kako bi se trebao ponašati u određenim situacijama. U nekim slučajevima, funkcionalni zahtjevi trebaju eksplicitno definirati i što sustav ne treba raditi. Funkcionalni zahtjevi su kompletni ako sadrže opise svih zahtijevanih mogućnosti, dok su konzistentni ako ne sadržavaju konflikte ili proturječne tvrdnje. U praksi je nemoguće postići kompletan i konzistentan dokument o funkcionalnim zahtjevima.

Nefunkcionalni zahtjevi su ograničenja u uslugama i funkcijama programskog proizvoda. Najgrublja podjela je na tri tipa:

- 1) zahtjeve programskog proizvoda, koji specificiraju na koji se osobiti način treba ponašati isporučeni proizvod, npr. da ima odgovarajuće vrijeme odziva, da radi na operacijskom sustavu Windows 7, da koristi HTTPS protokol;
- 2) organizacijske zahtjeve, koji su rezultat organizacijskih pravila i procedura, npr. uporaba propisanog normiranog procesa razvoja, korištenje uvijek točno određenog programskog jezika pri razvoju;
- 3) vanjske zahtjeve, koji proizlaze izvan sustava i razvojnog procesa, npr. postizanje međusobne operabilnosti s drugim sustavima, zakonske zahtjeve i drugo.

Dobra je ideja da se većina nefunkcionalnih zahtjeva navede kvantitativno kako bi se mogli ispravno ispitati.

**Primjer: Hipotetski sustav LIBSYS**

Opis: Knjižničarski sustav koji pruža jedinstveno sučelje prema bazama članaka u različitim knjižnicama. Korisnik može pretraživati, spremati i ispisivati članke za osobne potrebe.

Zahtjevi: Korisnik mora moći pretraživati početni skup baza članaka ili njihov podskup. Sustav mora sadržavati odgovarajuće preglednike koji omogućuju čitanje članaka u knjižnici. Korisničko sučelje LIBSYS sustava treba biti implementirano kao jednostavni HTML bez uporabe okvira ili Java "appleta". Svakoj narudžbi mora se alocirati jedinstveni identifikator (ORDER\_ID) koji korisnik mora moći kopirati u svoj korisnički prostor.

**Slika 4.2.** Zahtjevi hipotetskog programski sustav LIBSYS.

Zahtjevi domene primjene su takvi zahtjevi koji su specifični za domenu primjene sustava. Oni mogu biti novi funkcionalni zahtjevi ili ograničenja na postojeće zahtjeve. Takvi zahtjevi se obično saznavaju kasnije u procesu dobivanja (otkrivanja) zahtjeva buduće da ih je teško izvući od stručnjaka. Naime, prilikom otkrivanja zahtjeva, postoje dva problema: razumljivost i implicitnost. Razvojni tim ne razumije domenu primjene i stoga traži detaljan opis zahtjeva (**razumljivost**), dok specijalisti domene poznaju primjenu tako dobro da podrazumijevaju neke zahtjeve (**implicitnost**) i ne navedu ih. Dok razvojni tim uspije izvući traženu funkcionalnost, sustav već možda bude i dovršen, što poskupljuje izmjene.

#### **4.1.3. Razlikovanje tipa zahtjeva**

U praksi se pokazuje da razlikovanje tipova zahtjeva po sadržaju često nije jednostavno. Korisnički zahtjev koji se tiče sigurnosti, kao što je iskaz da se pristup treba ograničiti samo na ovjerene korisnike, može se činiti nefunkcionalnim zahtjevom. Međutim, kad se detaljnije razradi, ovakav zahtjev može generirati druge zahtjeve koji su jasno funkcionalni, kao što je potreba za ugradnjom programske potpore koja se bavi s ovjerom korisnika.

Iz primjera programskog sustava LIBSYS navedenog na slici 4.2. moguće je pojasniti probleme koji se javljaju prilikom navođenja zahtjeva. Iz primjera je očit nedostatak jasnoće, budući da nije odmah jasno točno što se treba napraviti. Preciznost u zahtjevima je teško postići bez detaljiziranog, ali stoga i naporno čitljivog dokumenta. Drugo što se može primijetiti je da se miješaju funkcionalni i nefunkcionalni zahtjevi. Tipični nefunkcionalni zahtjev je specifikacija točne vrste tehnologije koja se treba koristiti (HTML). Nadalje, postoji nenamjerno objedinjavanje više zahtjeva u jednom, što se vidi u zadnjoj rečenici zahtjeva (alokacija identifikatora i njegovo kopiranje). Konačno, nejasno postavljeni zahtjevi mogu biti različito interpretirani od korisnika i razvojnih timova što dovodi do problema u procesu razvoja i konačno, u kršenju ugovora. Tipičan primjer toga je izjava "odgovarajući

preglednik". Namjera korisnika je da postoji više preglednika posebne namjene za svaki tip dokumenta. Namjera razvojnog tima je da postoji samo jedan preglednik teksta, kao bitnog sadržaja dokumenta. Naravno, razvojni tim često ide za time da smanji količinu potencijalno nepotrebnog posla.

#### 4.1.4. Načini specifikacije zahtjeva

Kao što je već ranije rečeno, zahtjevi sustava mogu se napisati na više načina, pri čemu se uvijek nastoji izbjeći nestrukturirani prirodni jezik u obliku pisanog teksta. U tablici 4.2. prikazane su sve mogućnosti za pisanje specifikacije zahtjeva sustava. U praksi se koriste svi pristupi, a često se i kombinira više od jednog načina specifikacije.

#### 4.1.5. Dokument zahtjeva

Dokument zahtjeva programske potpore je službena izjava o tome što točno trebaju razvojni inženjeri ostvariti. On obično sadržava i neformalne korisničke zahtjeve i detaljniju specifikaciju zahtjeva sustava. Kad je to moguće, ponekad se obje vrste zahtjeva integriraju u jedinstveni opis. U ostalim slučajevima, korisnički zahtjevi su definirani u uvodu, dok su zahtjevi sustava definirani u ostatku dokumenta. U slučaju većeg broja zahtjeva sustava detaljni zahtjevi mogu se prikazati u zasebnom dokumentu. Razina detalja koja je uključena u dokument zahtjeva ovisi o vrsti sustava koji se razvija i o vrsti razvojnog procesa. Kritični sustavi, primjerice, trebaju imati detaljnu razradu zahtjeva budući da su pitanje sigurnosti i zaštite od velike važnosti.

**Tablica 4.2.** Mogućnosti specifikacije zahtjeva sustava.

Notacija	Opis
<b>Rečenice prirodnog jezika</b>	Zahtjevi su pisani korištenjem pobrojanih rečenica prirodnog jezika. Svaka rečenica trebala bi izreći samo jedan zahtjev sustava.
<b>Strukturirani prirodni jezik</b>	Zahtjevi su pisani u prirodnom jeziku u normiranom obliku ili na obrascu. Svako polje daje informaciju o jednom vidu zahtjeva.
<b>Specifični jezik za opis oblikovanja</b>	Ovaj pristup koristi jezik sličan programskom jeziku, ali s apstraktnijim značajkama za definiranje operativnog modela programskog sustava. Primjer jezika je SDL (engl. <i>Specification and Description Language</i> ). Danas se ovaj pristup rijetko koristi, osim za specifikaciju sučelja.
<b>Grafička notacija</b>	Grafički modeli koji su nadopunjeni tekstualnim opisom koriste se za definiciju funkcionalnih zahtjeva sustava. Pritom se najčešće koristi <b>UML dijagrami obrazaca uporabe</b> (engl. <i>use case diagram</i> ) i <b>UML sekvencijski dijagrami</b> (engl. <i>sequence diagram</i> ).
<b>Matematička specifikacija</b>	Notacija zasnovana na matematičkim konceptima kao što su strojevi s konačnim brojem stanja, skupovima, logici. Ovo je najstrože definirana specifikacija koju klijenti ne vole jer ju najčešće ne razumiju.



Prema normi instituta IEEE iz 1998. i prema doradi Sommervillea, dokument zahtjeva trebao bi sadržavati sljedeće stavke [3, 4]:

- 1) Predgovor
- 2) Uvod
- 3) Rječnik pojmova
- 4) Definicije korisničkih zahtjeva
- 5) Specifikacija zahtjeva sustava
- 6) Arhitektura sustava
- 7) Modeli sustava,
- 8) Evolucija sustava,
- 9) Dodaci
- 10) Indeks (pjmova, dijagrama, funkcija)

Potrebno je uočiti da ovako strukturirani dokument zahtjeva već predviđa buduću arhitekturu i model sustav, kao i evoluciju sustava, u vidu inačica i mogućih proširenja.

## 4.2. Procesi inženjerstva zahtjeva

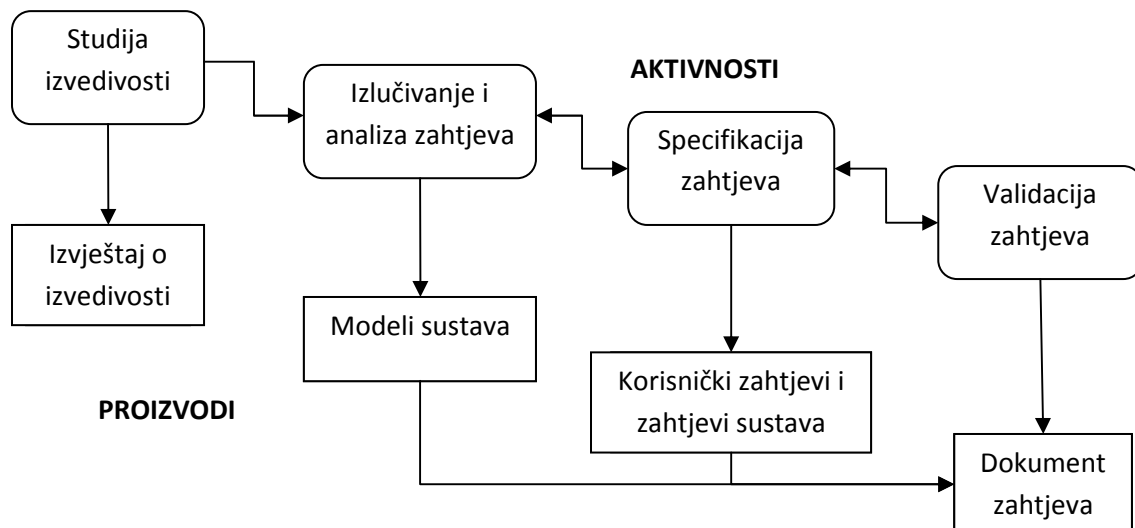
Procesi koji su u uporabi u inženjerstvu zahtjeva programske potpore razlikuju se ovisno o domeni primjene, ljudskim resursima i organizaciji koja oblikuje zahtjeve. Pritom je važno naglasiti da **nema jedinstvenog procesa inženjerstva zahtjeva** koji bi bio primijenjiv neovisno o vrsti projekta. Ipak, u okviru svakog procesa postoje neke generičke aktivnosti inženjerstva zahtjeva koje se redom odvijaju kako bi se u konačnici definirao dokument zahtjeva. To su:

- **studija izvedivosti (engl. *feasibility study*),**
- **izlučivanje (otkrivanje) zahtjeva i analiza zahtjeva (engl. *requirements elicitation and analysis*),**
- **specifikacija (zapisivanje) zahtjeva (engl. *requirements specification*),**
- **validacija zahtjeva (engl. *requirements validation*).**

Osim ove četiri temeljne generičke aktivnosti, u novije vrijeme kao dodatna aktivnost navodi se i upravljanje promjenama u zahtjevima (engl. *requirements management*).

### 4.2.1. Generičke aktivnosti inženjerstva zahtjeva

Generičke aktivnosti inženjerstva zahtjeva ne odvijaju se nužno slijedno, već se mnogo češće isprepliću. Na slici 4.3. prikazan je **klasični model procesa inženjerstva zahtjeva** u kojem se konačni dokument zahtjeva postepeno slaže na temelju nekoliko manjih



**Slika 4.3.** Klasični model procesa inženjerstva zahtjeva.

dokumenata - proizvoda pojedinih generičkih aktivnosti. Iteracije su prisutne između aktivnosti izlučivanja i analize zahtjeva, specifikacije zahtjeva i validacije zahtjeva, sve dok se zahtjevi ne usaglase.

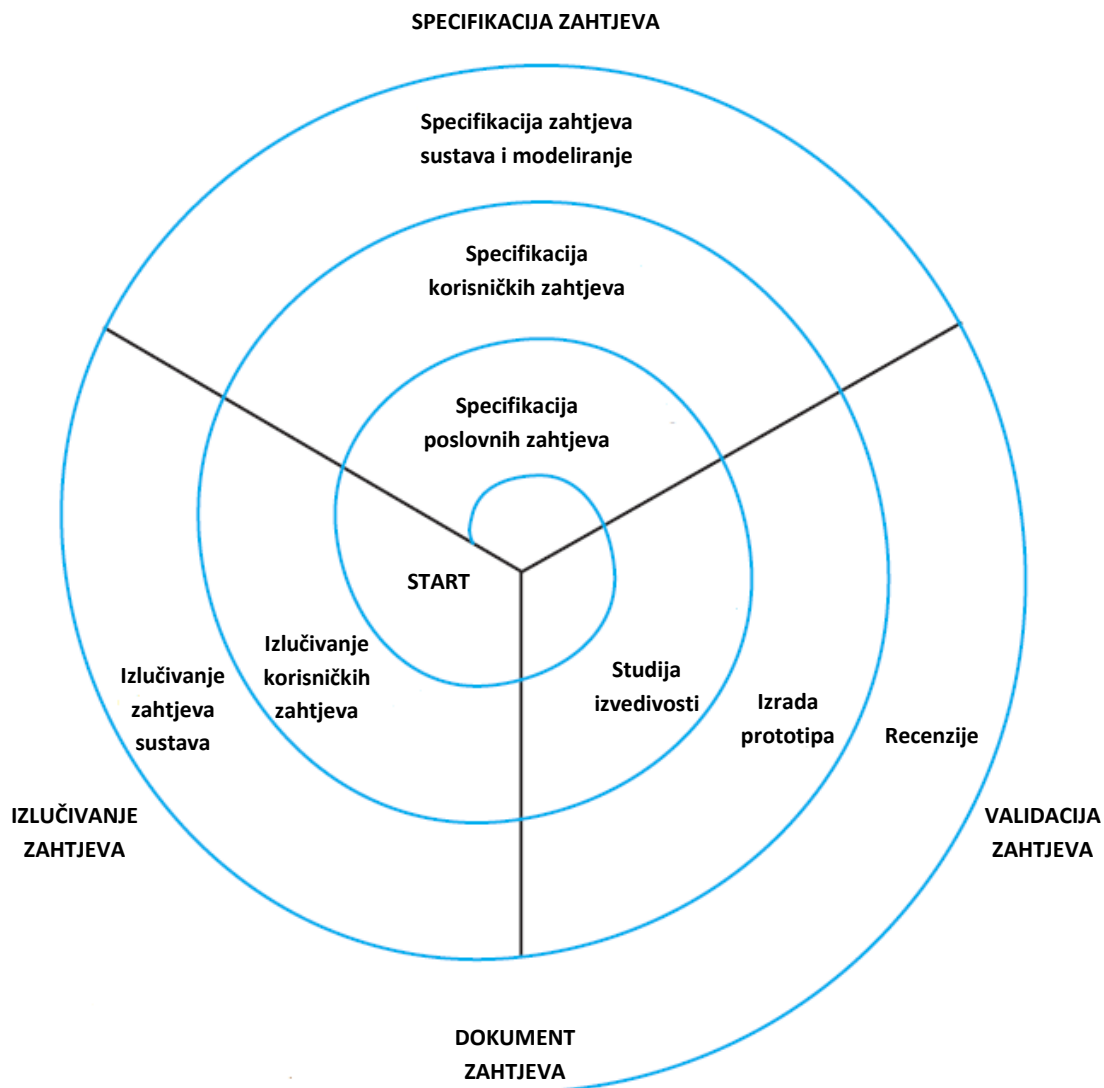
Bolji model razvoja specifikacije programske potpore od klasičnog je **spiralni**, kod kojeg vrijeme i trud uloženi u svaku aktivnost ovise o stupnju razvoja i vrsti programskog proizvoda, slika 4.4. Spiralni model je **trostupanjski ciklus** koji se sastoji od ponavljanja **specifikacije, validacije i izlučivanja** zahtjeva.

U ranim fazama procesa, najveći intenzitet aktivnosti je na razumijevanju poslovnih i nefunkcionalnih zahtjeva visoke razine apstrakcije. Zatim se proučavaju korisničkih zahtjevi i izrađuju prototipovi sustava, da bi se u vanjskim slojevima spirale konačno prešlo na detaljno izlučivanje, analizu i specifikaciju svih zahtjeva sustava. Na taj način, spiralni model izravno podržava razradu zahtjeva prema razini detalja. Broj iteracija po spirali pritom može varirati. Kod agilnih metoda razvoja programske potpore, umjesto razvoja prototipa proizvoda, može se ići na simultani razvoj specifikacije i implementacije.

#### 4.2.2. Studija izvedivosti

Studija izvedivosti je kratka, fokusirana studija na početku procesa inženjerstva zahtjeva kojom se utvrđuje isplati li se predloženi sustav (tj. je li vrijedan uložениh sredstava). Ulazna informacija je preliminaran skup poslovnih zahtjeva procesa. Zadatak studije izvedivosti je odgovori na tri temeljna pitanja:

- Doprinosi li programski sustav ciljevima organizacije u koju se uvodi?
- Može li se programski sustav izvesti postojećom tehnologijom i predviđenim sredstvima?



**Slika 4.4.** Spiralni model procesa inženjerstva zahtjeva, prilagođeno iz [3].

- Može li se predloženi sustav integrirati s postojećim sustavima organizacije u koju se uvodi?

U slučaju da se utvrdi da je odgovor na barem jedno od tih pitanja negativan, ne bi bilo poželjno nastaviti s daljnjim razvojem programskog sustava. Pri provođenju studije izvedivosti, najprije se određuje koje su sve informacije potrebne kako bi se studija završila, zatim se te informacije prikupljaju, i na kraju se piše izvješće. Studija izvedivosti često se provodi na terenu, u organizaciji gdje bi se sustav implementirao. Neka od pitanja za ljude u organizaciji koja se često postavljaju u studiji su:

- Koji su trenutni problemi procesa organizacije?
- Kako bi predloženi sustav pomogao u poboljšanju procesa?
- Što ako se sustav ne implementira?

- Koji se problemi mogu očekivati pri integraciji novoga sustava?
- Da li je potrebna nova tehnologija ili nove vještine?
- Koje dodatne resurse organizacije traži implementacija novoga sustava?

#### 4.2.3. Izlučivanje i analiza zahtjeva

Izlučivanje i analiza zahtjeva najznačajnija je generička aktivnost procesa inženjerstva zahtjeva. Tijekom ove aktivnosti, razvojni inženjeri i drugo tehnički obrazovano osoblje surađuje s klijentima i konačnim korisnicima sustava kako bi saznali što više o domeni primjene i o tome koje usluge trebaju biti podržane s kakvim performansama i ograničenjima. Vrlo je bitno pri toj aktivnosti uključiti sve ili što više dionika sustava - osoba koje će razvijeni sustav direktno ili indirektno pogoditi. Primjerice, za sustav bankomata, mogući dionici programskog sustava uključuju:

- bankovne klijente,
- predstavnike drugih banaka,
- bankovne rukovoditelje,
- šalterske službenice,
- administratore baza podataka,
- rukovoditelje sigurnosti,
- marketinški odjel,
- inženjere održavanja sustava (sklopovlja i programske potpore),
- regulatorna tijela za bankarstvo.

Model aktivnosti izlučivanja i analize zahtjeva prikazan je na slici 4.5. Svaka organizacija imaće vlastiti način provedbe ovog modela, koji će ovisiti o specijalizaciji osoblja, vrsti programskog proizvoda, korištenim normama razvoja i slično. Proces izlučivanja i analize zahtjeva je iterativan. Znanje o zahtjevima sustava se tijekom svakog ponavljanja ciklusa poboljšava dok se ne kompletira. Prvi korak, **izlučivanje (otkrivanje) zahtjeva** je postupak interakcije sa svim dionicima s ciljem otkrivanja njihovih zahtjeva. Zahtjevi domene primjene se također nastoje definirati u ovom koraku. Izvori informacija su dokumenti, dionici, slični razvijeni sustavi. Tehnike koje se koriste za otkrivanje zahtjeva su uglavnom intervjuiranje, izrada scenarija i etnografija, o čemu će nešto kasnije biti riječi. Ponekad se za otkrivanje zahtjeva koriste i UML obrasci uporabe. Tijekom drugog koraka, **klasifikacije i organizacije zahtjeva**, srodni zahtjevi se grupiraju i organiziraju u koherentne grozdove (klastere). Zahtjevi se u praksi najčešće grupiraju prema podsustavu cijelog sustava koji opisuju. Tako je teško u praksi razgraničiti postupak specifikacije zahtjeva od razvoja arhitekture sustava. Tijekom trećeg koraka, **ustanovljavanja prioriteta i pregovaranja**, zahtjevi se razvrstavaju po prioritetima i razrješavaju se konflikti nastali zbog različitih



**Slika 4.5.** Model aktivnosti izlučivanja i analize zahtjeva.

pogleda različitih dionika na sustav. U praksi je cilj postići kompromis zahtjeva među dionicima. Na kraju se u četvrtom koraku zahtjevi **specificiraju (pisano dokumentiraju)** i predstavljaju ulaz u sljedeću iteraciju. Zahtjevi se specificiraju tehnikama prema tablici 4.2. i zapisuju u formalnim ili neformalnim dokumentima, a najčešće grafičkom notacijom u vidu UML obrazaca uporabe i UML sekvencijskih dijagrama.

Pri izlučivanju zahtjeva, različiti dionici ili grupe dionika imaju različite fokus i perspektivu na zahtjeve sustava.



**Definicija 4.1.** *Pogledi (engl. viewpoint) su način strukturiranja zahtjeva tako da oslikavaju perspektivu i fokus različitih grupa dionika. Svaki pogled uključuje skup zahtjeva sustava.*

Razlikuju se:

- **pogledi interakcije** - ljudi i drugi dionici koji izravno komuniciraju sa sustavom,
- **neizravni pogledi** - dionici koji utječu na zahtjeve, ali ne koriste sustav izravno,
- **pogledi domene primjene** - karakteristike domene i ograničenja na sustav.

Perspektive na sustav nisu posve nezavisne nego se koji put preklapaju. Pogledi se često koriste kako bi se strukturirale aktivnosti otkrivanja i specifikacije zahtjeva.

Aktivnost otkrivanja zahtjeva teška je iz više razloga:

- Dionici ne znaju što stvarno žele (teško artikuliraju zahtjeve, ili su zahtjevi nerealni s obzirom na izvedivost ili cijenu).
- Dionici izražavaju zahtjeve na različite, njima specifične načine (posjeduju implicitno znanje o svojem radu - domeni).
- Različiti dionici mogu imati konfliktne zahtjeve i izražene na različite načine.

- Organizacijski i politički faktori mogu utjecati na zahtjeve (npr. rukovoditelj traži da uvedeni sustav ima značajke koje povećavaju njegov utjecaj u organizaciji).
- Zbog dinamike ekonomskog i poslovnog okruženja zahtjevi se mijenjaju za vrijeme procesa analize.
- Uz promjenu poslovnog okruženja pojavljuju se novi dionici s novim, specifičnim zahtjevima.

**Intervjuiranje**, kao metoda izlučivanja zahtjeva, jedan je od najčešće korištenih i često nezaobilaznih pristupa dobivanju korisničkih zahtjeva. Razlikuju se formalni i neformalni intervjui. Kod formalnih intervjuja, klijenta se prethodno obavještava da će biti intervjuiran vezano uz zahtjeve budućeg programskog sustava, dok se kod neformalnog intervjuja takva obavijest ne zahtijeva. I u formalnom i u neformalnom intervjuiranju tim zadužen za inženjerstvo zahtjeva ispituje dionike o sustavu koji trenutno koriste te o novo predloženom sustavu.

Prema tipu intervjuja, razlikuju se zatvoreni i otvoreni intervjui. Kod zatvorenog tipa intervjuja, klijent odgovara na skup već prije definiranih pitanja koja zanimaju razvojni tim. Kod otvorenog tipa intervjuja ne postoje unaprijed definirana pitanja, već se niz pitanja otvara i raspravlja s dionicima. U praksi, intervjui su najčešće kombinacija oba tipa. Obično se krene s nekim definiranim pitanjima pa se diskusija razvija u određenom smjeru. Potrebno je naglasiti da potpuno otvoreni tip intervjuja rijetko kad rezultira u preciznim informacijama, jer diskusije odu u posve nepotrebnim pravcima.

Intervjui su korisni za dobivanje globalne slike o tome što pojedini dionici rade i koja im je uloga te kako će oni interagirati s novo razvijenim sustavom. Također su dobri za otkrivanje poteškoća s trenutnim sustavom, budući da klijenti najčešće vole pričati o svojem poslu, a često se vole i žaliti na trenutno stanje u tvrtki. Međutim, intervjui nisu toliko korisni za razumijevanje zahtjeva u domeni primjene, budući da inženjeri zahtjeva ne razumiju specifičnu terminologiju domene, a eksperti domene toliko poznaju te zahtjeve da ih ni ne artikuliraju (misle da su svima razumljivi sami po sebi). Intervjui također nisu dobri za dobivanje uvida u organizacijske zahtjeve i ograničenja jer unutar tvrtke često postoje suptilni odnosi moći koji sprečavaju inženjera da dobije ispravnu sliku o organizaciji tvrtke. Osobe koje su zadužene za provođenje intervjuja su učinkovite ako su otvorenog uma, bez unaprijed osmišljenih zahtjeva, ako pažljivo slušaju klijente i ako su spremne klijentu predložiti zahtjeve, posebice u vidu prototipa sustava. U pravilu, nije dobra ideja pristupiti klijentima sa stavom: "reci mi točno što trebaš", već se odnos uvijek treba postepeno graditi i biti susretljiv s klijentom.

Klijenti često puno lakše shvate programski sustav na temelju stvarnim primjera nego kroz apstraktne opise. Upravo iz tog razloga, **scenariji** su popularna i uspješna metoda izlučivanja zahtjeva.



Scenariji su pažljivo osmišljeni primjeri o stvarnom načinu korištenja sustava.

Tijekom izlučivanja zahtjeva, dionici diskutiraju i kritiziraju scenarij. Svaki scenarij pokriva jedan ili nekoliko mogućih interakcija korisnika sa sustavom. Scenarij obično počinje s opisom interakcije. Tijekom procesa izlučivanja zahtjeva, dodaju se razni detalji opisu interakcije sve dok scenarij nije završen. Scenariji mogu biti napisani u obliku teksta uz potporu dijagrama, slika ekrana (engl. *screenshot*) i na druge načine. Mogu se koristiti više ili manje strukturirani oblici scenarija, a scenarije je posebno pogodno prikazati UML dijagramima obrazaca uporabe.

Svaki scenarij trebao bi sadržavati sljedeće elemente:

- Opis početne situacije.
- Opis normalnog (standardnog) tijeka događaja.
- Opis što se eventualno može dogoditi krivo.
- Informaciju o paralelnim aktivnostima.
- Opis stanja gdje scenarij završava.

Primjer scenarija za prikupljanje povijesti bolesti u sustavu MHC-PMS prikazan je na slici 4.6.

Jedan od razloga zašto se mnogo programskih sustava isporuči, ali se nikad ne koristi je taj što konačni zahtjevi sustava ne uzimaju dovoljno u obzir to kako društveni i organizacijski kontekst utječu na pojedinu operaciju u sustavu.



Etnografija je tehnika opažanja koja se koristi kako bi se bolje razumjeli procesi kod klijenta i kako bi se pomoglo otkriti što je moguće više ispravnih i korisnih zahtjeva. Etnografija podrazumijeva dolazak jednog ili više ljudi iz razvojnog tima u tvrtku gdje će se sustav primjenjivati i uključivanje tih inženjera (tzv. etnografa) u svakodnevne aktivnosti u tom okruženju.

Svakodnevni rad korisnika se prati i zapisuju se stvarni zadaci koje korisnici obavljaju. Na taj način otkrivaju se implicitni zahtjevi sustava koji pokazuju stvarni način kako ljudi rade, a ne formalne procese koje definira organizacija.

Etnografija se može kombinirati s izradom prototipa sustava u tzv. fokusiranoj etnografiji, slika 4.7. Ideja je da etnograf informira ostale inženjere tijekom razvoja prototipa, čime se smanjuje broj ciklusa poboljšavanja prototipa. Nadalje, izrada prototipa fokusira etnografiju tako što otkriva probleme i pitanja koja se potom mogu prodiskutirati s etnografom i na koje on može potražiti odgovore kod korisnika u sljedećoj fazi razvoja.

**Početna pretpostavka:**

Pacijent se našao s medicinskom sestrom na recepciji koja mu je otvorila zapis u sustavu i prikupila osobne informacije. Druga medicinska sestra se prijavila u sustav i prikuplja povijest bolesti.

**Normalni tijek događaja:**

Sestra pretražuje pacijenta po prezimenu. Ako ima više pacijenata s istim prezimenom, pacijenta se dodatno identificira s imenom i datumom rođenja.

Sestra odabire opciju u izborniku za dodavanje povijesti bolesti.

Sestra slijedi niz upita kako bi unijela informacije o: mentalnom zdravlju (tekst), postojećim fizičkim bolestima (odabir iz izbornika), lijekovima (odabir iz izbornika) i alergijama (tekst).

**Što može poći po krivu:**

Pacijentov zapis ne postoji ili ga se ne može pronaći. Sestra bi trebala napraviti novi zapis.

Pacijentove bolesti ili lijekovi nisu odabrani iz izbornika. Sestra bi trebala izabrati opciju "ostalo" i unijeti tekstualni opis bolesti/lijeka.

Pacijent ne može ili ne želi dati povijest bolesti. Sestra bi trebala unijeti tekstualni opis o tome da pacijent ne može ili ne želi dati informaciju. Sustav treba ispisati uobičajeni, potpisani obrazac na kojem piše da nedostatak informacija znači da će liječenje biti ograničeno ili odgođeno. Sestra treba obrazac predati pacijentu.

**Ostale aktivnosti:**

Pacijentov zapis se može pregledavati ali ne i mijenjati od strane ostalih zaposlenika dok ga sestra mijenja.

**Završno stanje:**

Sestra je prijavljena u sustav. Pacijentov zapis koji sadrži povijest bolesti je unešen u sustav. Dnevnik sustava pokazuje početak i kraj sjednice i ime i prezime sestre koja je provela upis podataka.

**Slika 4.6.** Primjer scenarija za prikupljanje povijesti bolesti, prilagođeno iz [3].

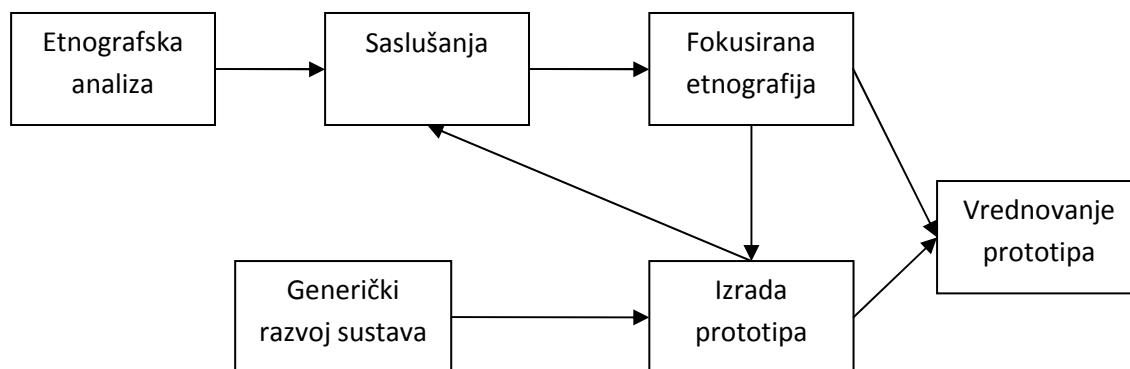
Etnografija nije pogodna za uočavanje novih značajki koje treba razviti, a i teško je pomoću nje uočiti organizacijske i domenske zahtjeve, budući da se fokusira na krajnjeg korisnika. Ipak, vrlo je korisna za uočavanje kritičnih detalja u procesu koji su možda izbjegli ostalim tehnikama izlučivanja i analize zahtjeva. Stoga se uvijek koristi u kombinaciji s ostalim tehnikama.

#### 4.2.4. Validacija zahtjeva

Validacija zahtjeva je proces provjere da li zahtjevi koje su dobiveni od klijenta zaista definiraju sustav koji korisnik želi. Validacija se isprepliće s analizom zahtjeva budući da se bavi pronalaskom pogrešaka u zahtjevima. Naknadno ispravljanje pogreške u zahtjevima može biti mnogo puta skuplje od jednostavnog ispravljanja pogreške u implementaciji. Tehnike validacije uključuju:

- **Recenziju zahtjeva** - detaljna, ručna analiza zahtjeva od strane zajedničkog tima.
- **Izradu prototipa** - provjera zahtjeva na izvedenom sustavu.
- **Generiranje ispitnih slučajeva** - razvoj ispitnih sekvenci za provjeru zahtjeva.





**Slika 4.7.** Fokusirana etnografija, prilagođeno iz [3].

U zahtjevima se provjerava više svojstava. To uključuje:

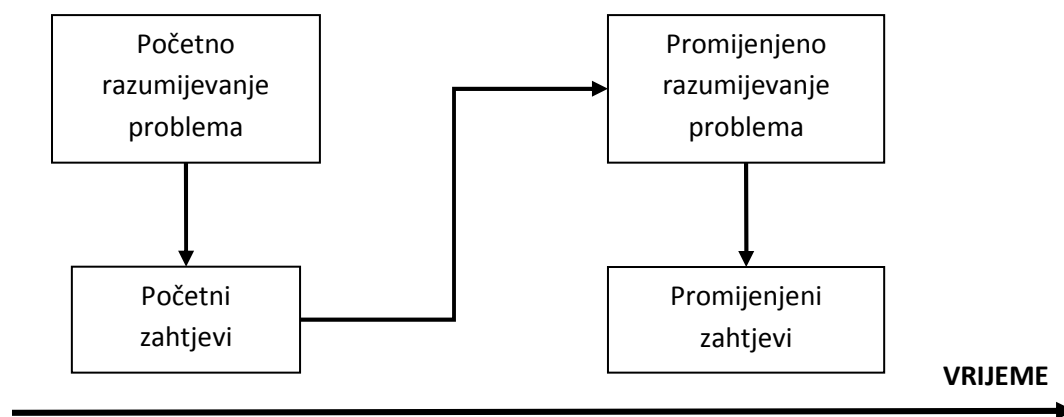
- **Valjanost** - ostvaruje li sustav funkcionalnost koja podupire potrebe većine dionika?
- **Konzistentnost** - postoji li konflikt u zahtjevima?
- **Kompletnost** - uključuje li sustav sve funkcije koje je korisnik tražio?
- **Realnost** - mogu li se sve funkcije implementirati uz danu tehnologiju i proračun?
- **Provjerljivost** - mogu li se svi zahtjevi provjeriti?
- **Razumljivost** - je li dokument zahtjeva jasno napisan?
- **Slijedivost** - je li naveden izvor dokumenta u slučaju više povezanih dokumenata?
- **Prilagodljivost** - mogu li se zahtjevi mijenjati bez utjecaja na druge zahtjeve?

Nije dobro potcijeniti probleme koji se tiču validacije zahtjeva. U konačnici, teško je pokazati da je neki skup zahtjeva točno onaj koji odgovara klijentskim potrebama. Potrebno je imati dobro razvijenu sposobnost zamišljanja programskog sustava u izvođenju, za što treba imati iskustva.

#### 4.2.5. Rukovanje promjenama u zahtjevima

Rukovanje promjenama u zahtjevima je proces razumijevanja i kontroliranja promjena u zahtjevima sustava. Rukovanje promjenama je nužno budući da promjene često nastupaju zbog promijenjenog modela poslovanja, boljeg razumijevanja procesa tijekom razvoja ili konfliktnih zahtjeva u različitim pogledima koji nisu bili na vrijeme uočeni. Evolucija zahtjeva programskog sustava prikazana je na slici 4.8.

Jednom kad je sustav instaliran i u svakodnevnoj uporabi, često se pojavljuju novi zahtjevi. Razlog tome je taj što je korisnicima i klijentima teško unaprijed predvidjeti sve učinke koje će imati novorazvijeni sustav na poslovne ili proizvodne procese. S vremenom bit će uočene nove potrebe i prioritete.



**Slika 4.8.** Evolucija zahtjeva, prilagođeno iz [1].

Moguće je klasificirati vrste promjena zahtjeva u sljedeće četiri kategorije:

- **Okolinom promijenjeni zahtjevi** - promjena zahtjeva zbog promjene okoline u kojoj organizacija posluje (npr. bolnica mijenja financijski model pokrivanja usluga).
- **Novonastali zahtjevi** - zahtjevi koji se pojavljuju kako kupac sve bolje razumije sustav koji se oblikuje.
- **Posljedični zahtjevi** - zahtjevi koji nastaju nakon uvođenja sustava u eksploataciju, a rezultat su promjena procesa rada u organizaciji nastalih upravo uvođenjem novoga sustava.
- **Zahtjevi kompatibilnosti** - zahtjevi koji ovise o procesima drugih sustava u organizaciji; ako se ti sustavi mijenjaju to traži promjenu zahtjeva i novo uvedeni sustav.

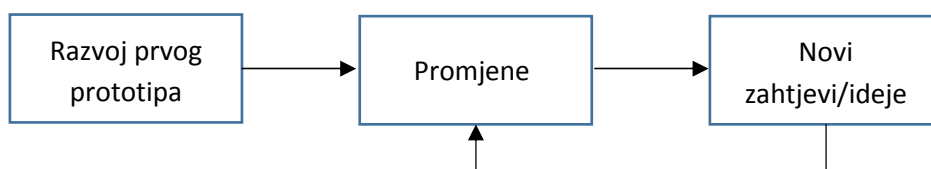
Pri rukovanju promjenama, potrebno je voditi računa o svim pojedinim zahtjevima i održavati veze između ovisnih zahtjeva tako da se može procijeniti značaj promjena. Zato se često na početku ustanovljava formalni proces za prijedlog uvođenja promjena i za povezivanje promjena za zahtjevima sustava. Prvi korak u provođenju rukovanja zahtjeva je planiranje. Tijekom tog koraka, odlučuje se o načinu identifikacije zahtjeva, procesu rukovanja promjenom, načinu sljedivosti promjena i o alatima koji će poduprijeti rukovanje promjenama.

Kod velikih sustava, potrebno je često dobro proanalizirati isplati li se ili ne uvoditi neku promjenu. Pritom se posebnu pozornost treba dati onim promjenama koje utječu na jedan ili više ostalih sustava s kojima trenutni sustav interagira. Najproblematičnije su one promjene koje se moraju "pod hitno" napraviti. U tom slučaju, važno je voditi računa o tome da se promjena provede na svim razinama i da su dokumenti o promjenama u potpunosti slijedivi, kako bi se izbjegla kasnije nemogućnost rekonstrukcije što je točno bilo izmijenjeno. Tu puno pomažu odgovarajući specijalizirani CASE-alati koji nastoje automatizirati proces evidencije promjena.

## 5. Procesi i modeli procesa programskog inženjerstva

Modeli procesa programskog inženjerstva predstavljaju uočene pristupe organizaciji projekta izrade programske potpore. Oni nisu prvenstveno stroži ili manje stroži recepti kako se taj posao mora napraviti, već pomoć pri u razmišljanju kako projekt uspješno privesti kraju. Do njih se došlo na temelju iskustava razvojnih timova kao o skupu pozitivnih zaključaka o važnosti i redoslijedu izvođenja projektnih aktivnosti.

U pristupu koji bi zanemario saznanja na kojima se temelje modeli programskog inženjerstva razvojni tim započinje izradu programske potpore te ju neprestano mijenja sve do zadovoljenja korisničkih zahtjeva. To bi se moglo nazvati kao *ad-hoc* ili oportunistički pristup koji do određene veličine razvijene programske potpore može biti zadovoljavajući. Mogao bi se prikazati kao niz aktivnosti prema slici 5.1.



**Slika 5.1.** *Ad-hoc* ili oportunistički pristup razvoju programske potpore.

Iskustvo takvog razvoja na duže staze prepoznaje određene problemi i nedostatke:

- Nema razrade zahtjeva i dizajna prije same implementacije. Neki korisnički zahtjevi će biti zadovoljeni, a neki neće. Da bi se došlo do zadovoljenja svih korisničkih zahtjeva potrebne su mnoge promjene.
- Događa se (brže) istrošenost ili propadanje (engl. *deterioration*) razvoja programske potpore.
- S obzirom na neplanski razvoj, cilj nikad nije dovoljno jasan pa nema ni pravog uvida radi se ono što se zapravo treba, a nema niti kontrole troškova razvoja.
- Proces ispitivanja ni bilo koji oblik potvrde očekivane kvalitete proizvoda nisu unaprijed ni izričito izdvojeni. To dovodi do postojanja nepokrivenih kvarova čiji popravci zahtijevaju stalne naknadne popravke što prije ili kasnije dovodi do nestabilnosti ili raspada sustava tako da je bolje ponovno početi izgradnju sustava “iz nule”.
- Zbog svih prethodno navedenih razloga trošak razvoja i održavanja je vrlo visok.

S druge strane, tri su jasno definirana i općeprihvaćena generička modela programskog inženjerstva koji nastoje nadići negativne strane ovog previše pojednostavljenog *ad-hoc* pristupa. Moglo bi ih se nazvati procesnim paradigmama gledano sa strane arhitekture programske potpore ili okvirima otvorenima za proširenja i prilagodbe koje će biti potpuno određene samim projektom. To su:

- **Vodopadni model** (engl. *waterfall model*) koji temeljne aktivnosti procesa izrade programske potpore gleda kao nezavisne faze razvoja.
- **Evolucijski model** (engl. *evolutionary model*) kod kojeg se sustav razvija kroz niz inačica ili inkremenata kod kojih svaka sljedeća dodaje neku novu funkcionalnost u onu na koju se nastavlja.
- **Komponentni model** (engl. *component-based model*) čija je motivacija u pretpostavci ponovne iskoristivosti postojećih komponenata.

Osim ta tri generička modela, u novije vrijeme naglašavaju se kao zasebni modeli i **modelno-usmjereni razvoj programske potpore** (engl. *Rational Unified Process, RUP*) kao i **agilni razvoj** (engl. *agile development*). U stvarnosti, oni čine samo podmodele tri osnovna generička modela razvoja.

Tri generička modela programskog inženjerstva, vodopadni, evolucijski i komponentni, nisu potpuno međusobno isključiva. Kod razvoja velikih sustava ima ih smisla kombinirati. Npr., evolucijski pristup u načelu nije tu u principu pogodan, jer krenuti u razvoj velikih sustava bez jasno definiranih temelja sustava koji se temelje na analizi korisničkih zahtjeva i nema baš nekog smisla. Međutim, dijelovi tog podsustava koje često nije potrebno ili je malo teže unaprijed i do kraja jasno specificirati, kao npr. korisničko sučelje, se pritom mogu razvijati evolucijskim pristupom. Ponovno korištenje postojećih komponenti je tako široko rašireno da je skoro nemoguće zamisliti razvojni proces koji se temelji na potpuno nezavisnoj izradi vlastitih komponenti. Komponentno usmjereni razvoj često je i neformalno utkan u velikom broju razvojnih procesa u različitim fazama razvoja.

U potpoglavljima 5.2 - 5.6 opisuju se ipak svih pet modela razvoja programske potpore kao zasebne cjeline.

## 5.1. Iteracije u modelima procesa programskog inženjerstva

Procesi programskog inženjerstva ne odvijaju se odjednom i ne dovode odmah do konačnih rezultata, već se izvode kroz niz koraka koji sukcesivno, jedan za drugim, trebaju voditi bliže konačnoj verziji proizvoda. Pritom je sadržaj konačnog proizvoda određen prije početka tog iterativnog postupka. Proizvodom se u ovom slučaju smatra neki dio projektne dokumentacije, izvršna datoteka, baza podataka, dio programskog rješenja, itd.

Često se može čuti kako određeni dokument ili izvršna datoteka prolazi kroz **iteracije**. Time se želi reći da neki dokument ima više inačica i da mu se u svakom koraku dodaje novi sadržaj, ili otklanjaju pogreške. Pri tome je važno naglasiti da je uvijek tijekom izvođenja procesa programskog inženjerstva nužno imati definiciju ili kriterije završne inačice svakog dokumenta – npr. što završna inačica dokumenta treba sadržavati, koje nefunkcionalne i funkcionalne zahtjeve izvršna datoteka ili neki dio programskog koda mora implementirati, itd.

Iteracije u procesu programskog inženjerstva sastavni su dio velikih projekata jer se pojedini dijelovi moraju ponovno oblikovati. Iteracije se mogu primijeniti na bilo koji generički model procesa programskog inženjerstva i javljaju se u svakoj fazi procesa programskog inženjerstva.

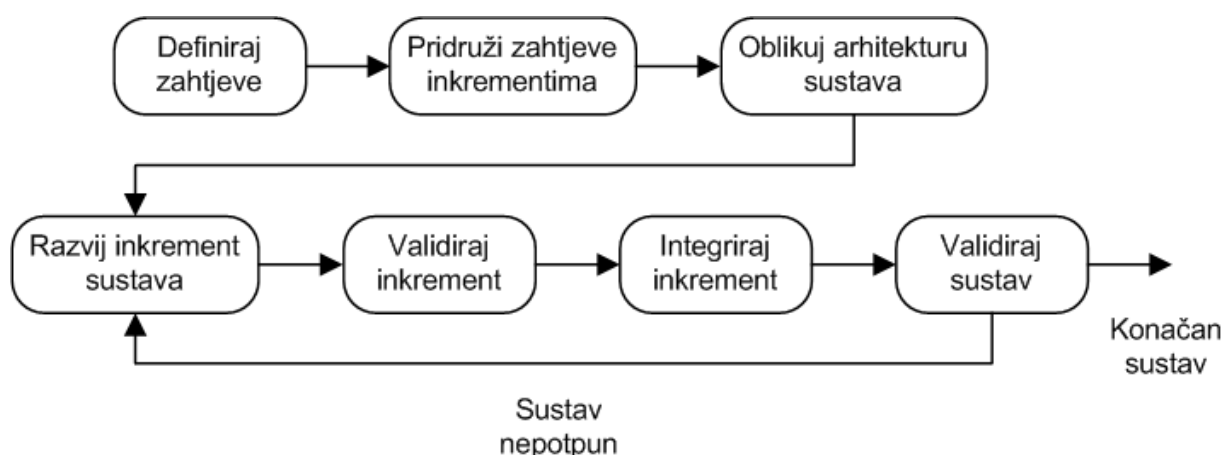
Postoje dva međusobno ovisna pristupa iteracijama:

### 1. Inkrementalni

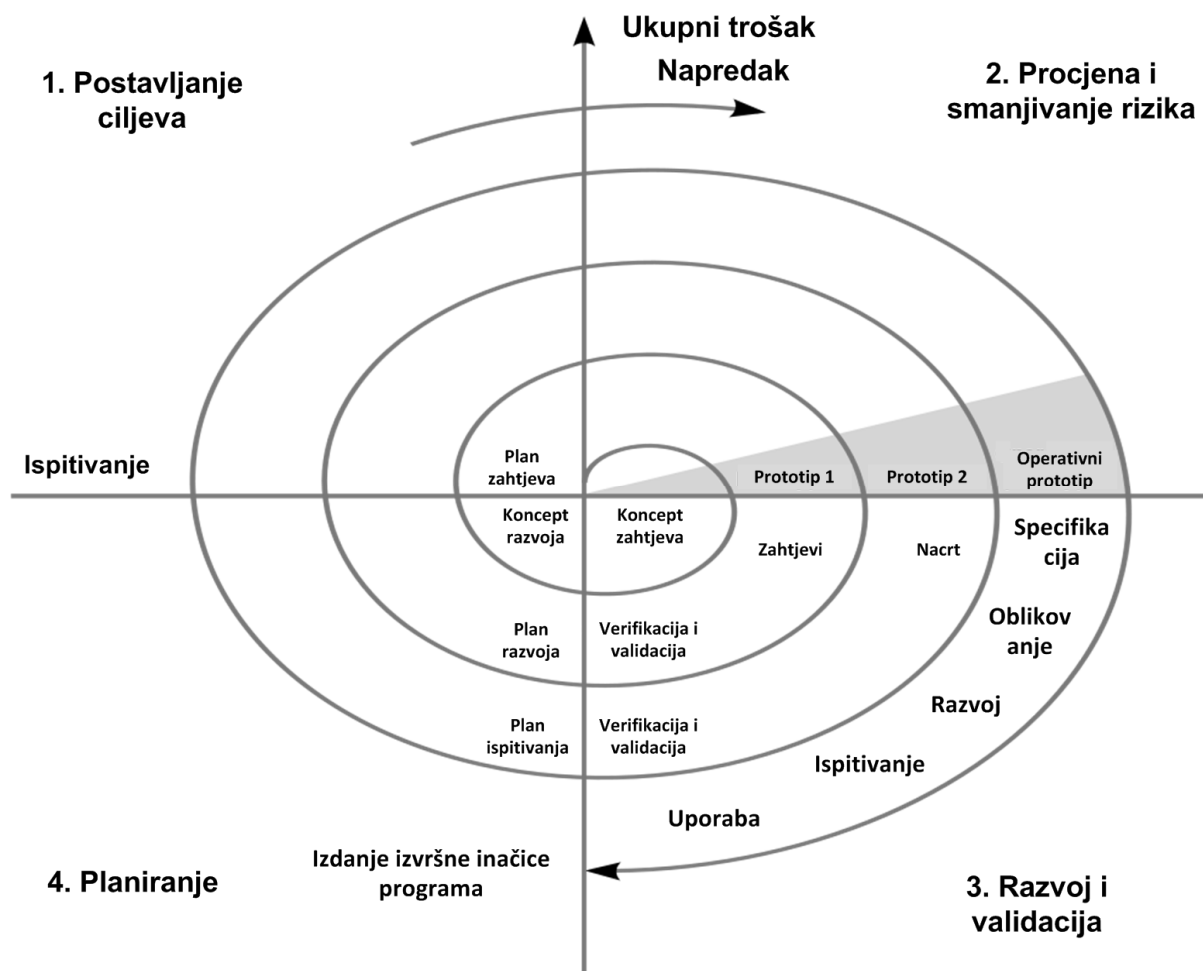
### 2. Spiralni

U inkrementalnom pristupu iteracijama (slika 5.2), sustav se ne isporučuje korisniku u cjelini. Aktivnosti razvoja, oblikovanja i isporuke razlažu se u inkrementalne dijelove koji predstavljaju djelomične funkcionalnosti proizvoda. Zahtjevi korisnika se analiziraju i organiziraju u prioritetne cjeline. Dijelovi višega prioriteta isporučuju se u ranim fazama razvoja sustava (i početnim inkrementima). S početkom razvoja pojedinog inkrementa njegovi zahtjevi se fiksiraju („zamrzavaju“). Zahtjevi na ostale kasnije inkremente nastavljaju evoluirati i prilagođavati se željama korisnika i mogućnostima implementacije.

Inkrementalni pristup ima više prednosti: kupac dobiva novu zahtijevanu vrijednost sa svakim inkrementom, temeljna funkcionalnost sustava se ostvaruje u ranim fazama projekta, rani inkreменти služe kao prototipovi na temelju kojih se izlučuju zahtjevi za kasnije inkremente, smanjen je rizik za neuspjeh projekta, te naposljetku, prioritetne funkcionalne mogućnosti sustava mogu se detaljnije ispitivati jer su implementirane u ranim fazama projekta. Neki od nedostataka inkrementalnog pristupa su otežano preslikavanje korisničkih zahtjeva u inkremente odgovarajuće veličine, a i teško je odrediti koje će zajedničke značajke sustava biti potrebne za razvoj svih daljnjih inkremenata.



**Slika 5.2.** Inkrementalni pristup iteracijama.



**Slika 5.3.** Spiralni pristup iteracijama.

Kod spiralnog pristupa (slika 5.3) iteracijama, proces oblikovanja predstavljan je oblikom spirale umjesto nizom aktivnosti. Svaka petlja u spirali predstavlja jednu fazu procesa. U svakoj fazi razvoja, eksplicitno se određuju i razrješavaju rizici razvoja programskog proizvoda.

Spirala je podijeljena u četiri sektora:

- postavljanje ciljeva,
- procjena i smanjivanje rizika,
- razvoj i validacija,
- planiranje.

U sektoru postavljanja ciljeva obavlja se identifikacija specifičnih ciljeva temeljem postojećih zahtjeva. U sektoru procjene i smanjivanja rizika analiziraju se rizici u opcijama te preslikavaju u aktivnosti koje ih reduciraju. Analiza rizika rezultira u evoluciji prototipova. Primjer jednog takvog postupka analize je SWOT analiza (snage, slabosti, prilike, prijetnje;

engl. *Strengths, Weaknesses, Opportunities, Threats*). U sektoru razvoja i validacije analizira se prototip proizvoda nakon čega slijedi njegov daljnji razvoj u sljedećim koracima spirale. U početnim koracima spirale razvija se **koncept**, a u kasnijim spiralnim iteracijama aktivnosti imaju sve više detalja (specifikacija, oblikovanje, razvoj, ispitivanje, uporaba). U sektoru planiranja obavlja se planiranje faza (tj. ciklusa spirale), od prikupljanja zahtjeva do oblikovanja, razvoja i konačno, integracije („od koncepta do produkta“).

## 5.2. Vodopadni model

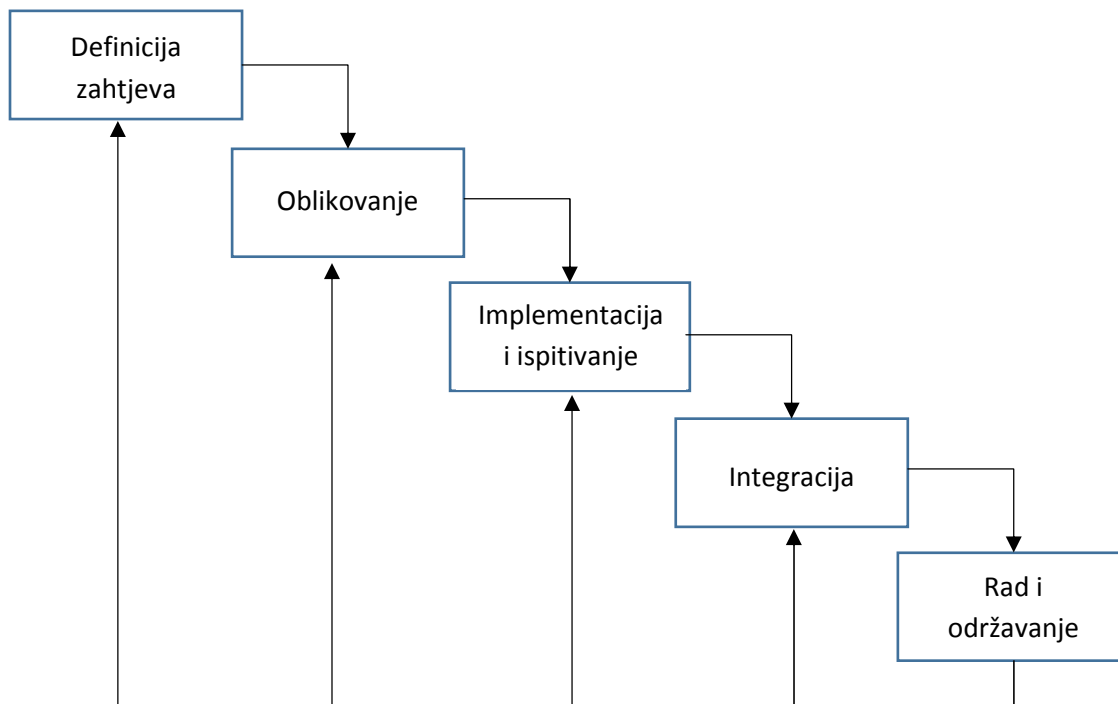
U ovom modelu, procesne faze su jasno odvojene i u načelu nezavisne. Tako se mogu zasebno promatrati faze:

- izlučivanja zahtjeva,
- oblikovanja,
- implementacije,
- integracije i
- održavanja sustava.

Prema tome se pretpostavlja i da prije početka rada na pojedinim fazama postoji dobro definiran plan rada za svaku od njih. Može se reći da ovaj model predstavlja “klasični” pristup projektiranju sustava gdje su koraci razvoja planski definirani i ne preklapaju se, već slijede jedan iza drugoga. Sljedeća faza pritom ne bi smjela započeti dok prethodna nije završena.

Detaljnije, izrada sustava po fazama se sastoji od (slika 5.4):

- Definicija zahtjeva. Konzultacije s korisnicima identificiraju koji su ciljevi sustava, koje usluge treba pružati i koja ograničenja su prisutna, što sve skupa čini konačnu specifikaciju sustava.
- Oblikovanje. Definira se arhitektura sustava kroz identifikaciju programskih i sklopovskih sredstava u odnosu na postavljene zahtjeve te se na strani programske potpore određuju temeljne apstrakcije izvedbe te njihovi odnosi.
- Implementacija i ispitivanje. Ostvarenje sustava kroz skupove programskih jedinica koje se pritom propisno ispituju odgovaraju li svojim specifikacijama.
- Integracija. Programske jedinice se ujedinjuju u cjelinu uz neizostavno ispitivanje cijelog sustava nakon čega bi trebao biti spreman za isporuku korisniku.
- Rad i održavanje. Podrazumijeva instalaciju i pogon sustava te dugotrajan rad na održavanju sustava kroz ispravljanje kvarova i nadogradnje već prema novim zahtjevima na funkcionalnost i uvjete radne okoline.



**Slika 5.4.** Vodopadni model procesa programskog inženjerstva.

U samoj ideji, ovaj pristup je strog na način da se očekuje visoka vremenska nezavisnost pojedinih faza. U stvarnosti, to nije tako jednostavan model, već se te faze uvijek djelomično preklapaju te postoje povratne veze od trenutnih prema prethodnim fazama. Potpuno zaključenje pojedinih faza ipak dolazi tek nakon povratnih informacija od faza koje ih slijede, npr. konačna specifikacija se zaključuje tek nakon ispravljanja nesuvislosti prepoznatih u fazi oblikovanja, arhitektura sustava se zaokružuje tek nakon prepoznavanja problema tijekom faze implementacije, itd.

Strogost vodopadnog modela se posebno očituje u činjenici nezavisnog dokumentiranja svake pojedine faze ne bi li se jasnije pratio stvarni napredak projekta u odnosu na postavljene zahtjeve. Sukladno tome se dokumentacija mijenja kako se događaju promjene sustava na temelju povratnih informacija od susjednih faza, što može biti vrlo skupo.

Zbog tako izražene nefleksibilnosti u ranoj podjeli na faze i skupim promjenama korisničkih zahtjeva, vodopadni model je pogodan za projekte koji imaju visoko razumljive zahtjeve s malom vjerojatnošću njihove promjene. Kao model upravljanja projektom, vodopadni model je najpogodniji te se uglavnom koristi za velike inženjerske projekte gdje se sustav razvija na nekoliko odvojenih mjesta. Također je prikladan ako je moguće uspostaviti i formalnu specifikaciju sustava temeljenu na matematičkom modelu što pogotovo ima smisla za sustave kritične u pogledu sigurnosti.



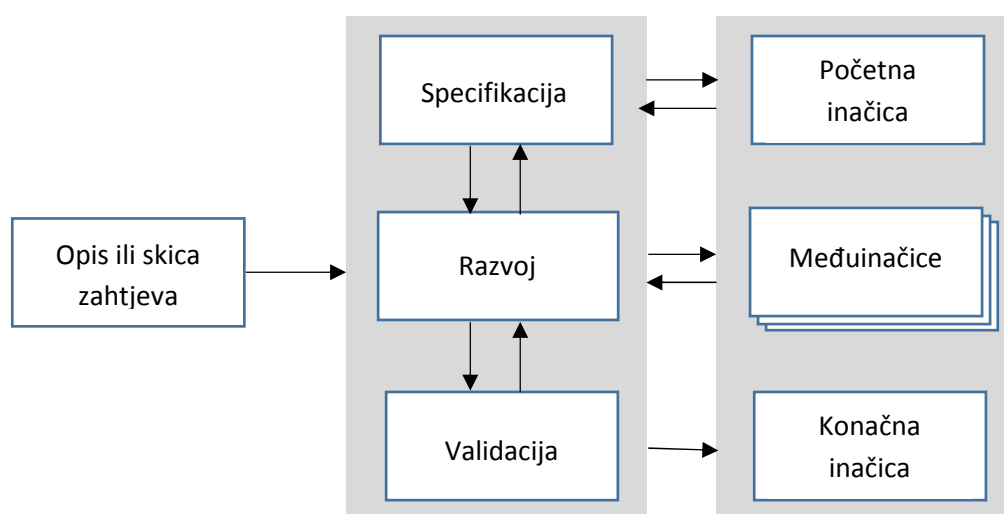
### 5.3. Evolucijski model

Ovaj model se još u literaturi referencira kao inkrementalni (engl. *incremental*) model gdje se sustav razvija kroz niz inačica (ili inkremenata), pri čemu svaka nova inačica pridodaje nove funkcionalnosti u prethodno izgrađenu.

Izdvajaju se elementi **specifikacije**, **razvoja** i **validacije**, koji su više isprepleteni nego razdvojeni te imaju više narav pojedinih aktivnosti nego pojedinačnih faza izrade. Između susjednih aktivnosti, učestalija je povratna komunikacija nego u vodopadnom modelu pa bi se moglo govoriti i o određenoj istodobnosti (engl. *concurrency*) među njima, slika 5.5. Dva su osnovna tipa evolucijskog modela razvoja programske potpore:

- **Istraživački razvoj i oblikovanje** (engl. *exploratory development*). Kod ovog pristupa je stalan rad s naručiteljem radi identifikacije pravih zahtjeva. Razvoj započinje u dijelovima sustava za koje su zahtjevi dovoljno razumljivi, a nakon toga se dodaju nove funkcionalnosti prema prijedlozima naručitelja.
- **Metoda odbacivanja prototipa** (engl. *throwaway prototyping*). U ovom pristupu je cilj bolje razumijevanje zahtjeva naručitelja i na osnovi toga bolja specifikacija zahtjeva. Prototipovi koji se pritom razvijaju na temelju slabo definiranih zahtjeva služe samo kao pomoć pri razjašnjavanju stvarnih potreba korisnika.

Ovaj model programskog inženjerstva odražava način kako vrlo često pristupamo rješavanju nekog programskog zadatka. Svaka inačica uvodi neku od novih funkcionalnosti zadanih u opisu zahtjeva. Najčešće prve inačice pritom sadrže najvažnije ili najžurnije funkcionalnosti pa ih korisnik već u ranim razdobljima projekta može vrednovati. U slučaju neispunjavanja ili promašaja zahtjeva, jeftiniji su ispravci pogrešaka jer je potrebno mijenjati samo jednu ili nekoliko zadnjih inačica koje prethode. U odnosu na vodopadni



**Slika 5.5.** Evolucijski model procesa programskog inženjerstva.

model, evolucijski pristup razvoju je značajno brži, posebice u početnom razdoblju rada te time čini i osnovu agilnog razvoja programske potpore. S time se može kombinirati i planski pristup te postoji više različitih načina pogleda na evolucijski model programskog inženjerstva. Kod više planskog pristupa, inkrementi se određuju unaprijed, a kod agilnog se prvi inkrementi brzo razvijaju dok razvoj budućih inkremenata ovisi o brzini napretka i korisničkim prioritetima. Jedna od karakteristika evolucijskog modela koja može biti i prednost je i da se specifikacija razvija inkrementalno.

Nedostatak evolucijskog pristupa razvoju programske potpore je taj što proces razvoja i oblikovanja nije jasno vidljiv. To ne odgovara voditeljima projekata jer ne mogu kvantificirati stvarni napredak. Ako se sustav razvija brzo, stalno dokumentiranje inačica ne mora biti uvijek isplativo. Iz istog razloga i stalnih izmjena, struktura sustava je često vrlo narušena. Refaktorizacija je tada u pravilu iznimno zahtjevnja, kao i ponovna uporaba dijelova sustava.

Evolucijski model je pogodan za male i srednje sustave ili za dijelove većih sustava kojima korisnički zahtjevi nisu kritični ni unaprijed egzaktno poznati, kao npr. korisničko sučelje. Za velike sustave može se koristiti u pojedinim fazama u kombinaciji s vodopadnim modelom, npr. za raščišćavanje pitanja stvarnih zahtjeva.

## 5.4. Komponentno-usmjereni model

### 5.4.1. Općenito o komponentama

Uz komponentno-usmjereni model veže se jedna čitava grana programskog inženjerstva, odnosno **komponentno-usmjereno programsko inženjerstvo** (engl. *component-based software engineering - CBSE*), koja se kao pristup razvoju programske potpore pojavila krajem 90-ih godina prošlog stoljeća s motivacijom postizanja bolje i šire primjene principa ponovne uporabljivosti (engl. *reuse*) programskih komponentata. Kao najjednostavnija definicija pojma komponente može se reći da je to:

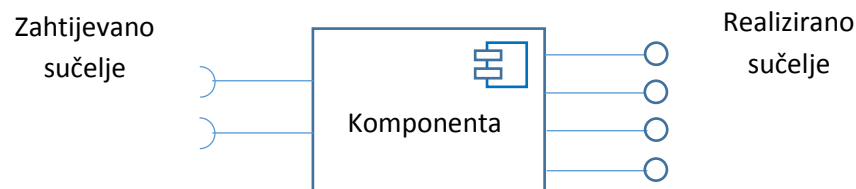


**Definicija 5.1.** *Komponenta je nezavisna jedinica programske potpore koja se može kombinirati s drugim nezavisnim jedinicama u svrhu izrade cjelovitog sustava programske potpore.*

Iako se ta definicija može nadopunjavati s obzirom na svojstva koja se odnose na normiranje, ugradnju i uporabu u kombinaciji s drugim komponentama, praktičan pogled na komponentu je taj da je to nezavisni entitet koji nudi jednu ili više usluga. Pritom ne bi trebalo biti niti bitno niti vidljivo gdje se ta komponenta doslovce nalazi ni kojom je tehnologijom ostvarena. Važno je, međutim, da je komponenta identificirana svojim sučeljem putem kojeg je dostupna i putem kojeg se obavlja sva interakcija s njom.

Različitost funkcionalnosti koje pruža se iskorištava putem parametrizirane komunikacije a da pritom nisu vidljiva unutarnja stanja komponente.

Dva su temeljna tipa sučelja komponente: realizirano ili izvozno (eksportirano, engl. *export*) sučelje, koje definira usluge koje komponenta pruža i zahtijevano ili uvozno (importirano, engl. *import*) sučelje, koje specificira koje usluge moraju biti na raspolaganju komponenti da bi ona bila operabilna. Prvi tip sučelja definira popis metoda koje mogu biti pozvane od strane korisnika komponente. Drugi tip sučelja navodi koje vanjske usluge joj moraju biti osigurane za koristan rad koji izvršava. Pritom se ne definira kako te usluge moraju biti osigurane, već se sadržaji koje daje prenose kao parametri operacija koje čine sučelje. U UML-notaciji se realizirano sučelje prikazuje kružićem, a zahtijevano polukružićem na kraju linije koja ide od simbola koji predstavlja jezgru komponente, slika 5.6.

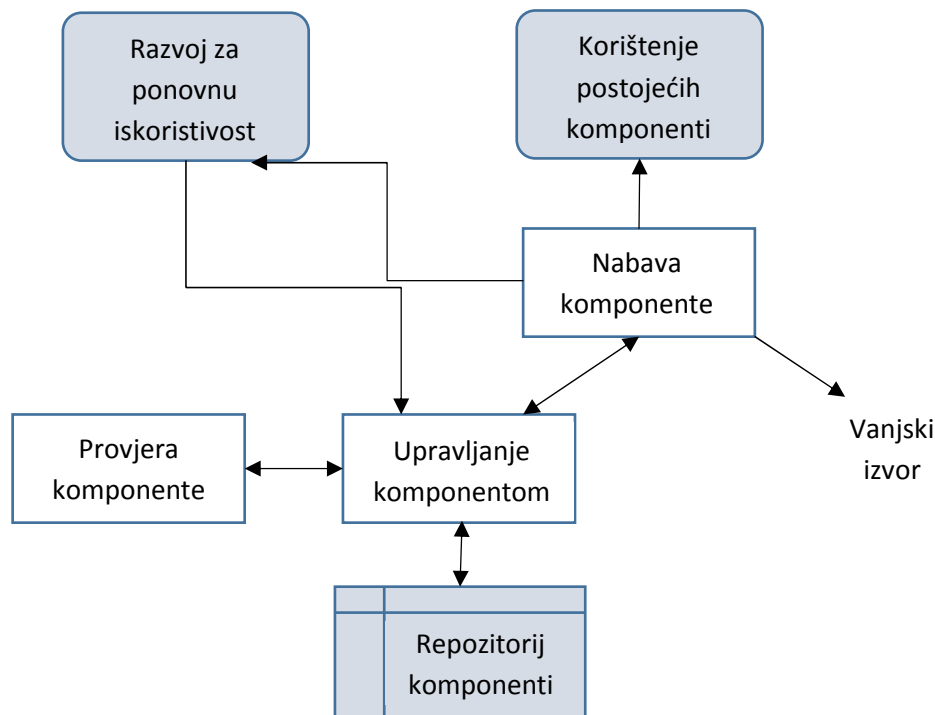


**Slika 5.6.** UML-notacija komponente.

S obzirom na neodvojivost pojma komponentno-usmjerenog programskog inženjerstva od ideje ponovne uporabljivosti komponente, procese za tu granu programskog inženjerstva možemo gledati s dvije strane kao:

- **Razvoj za ponovnu iskoristivost** (engl. *development for reuse*), gdje se grade komponente ili usluge predviđene za iscrpno korištenje u drugim razvojnim procesima. U ovom tipu razvoja, programski kôd komponente je u potpunosti otvoren, a često se proces svodi na generalizaciju postojećih komponenti za koje postoje naznake višestruke uporabljivosti.
- **Razvoj s korištenjem postojećih komponenti** (engl. *development with reuse*) gdje se grade nove aplikacije korištenjem postojećih komponenti. Prikladne komponente se pritom moraju najprije pronaći i identificirati kao one koje zadovoljavaju ili se mogu prilagoditi potrebama aplikacije koja se gradi. Pritom za te komponente ne mora biti dostupan programski kôd.

Slika 5.7 prikazuje pregled temeljnih procesa komponentno-usmjerenog programskog inženjerstva, uključujući oba pogleda na komponentno-usmjereni razvoj te procese nabave, provjere i upravljanja komponentama.



**Slika 5.7.** Pregled procesa komponentno-usmjerenog programskog inženjerstva.

Nabava komponente dohvaća komponente iz lokalnog izvora, tj. razvoja komponenti, ili vanjskog izvora za daljnji razvoj novih komponenti ili korištenje unutar novog sustava. Te komponente daje procesu upravljanja komponentama koji ih šalje na provjeru i pohranu u repozitorij. Provjera komponente sastoji se od nekog oblika certifikacije da ima funkcionalnosti navedenu u specifikaciji.

Prvonavedena vrsta komponentnog-usmjerenog inženjerstva, *razvoj za ponovnu iskoristivost*, stvara komponente koje će se koristiti u drugoj, *razvoju s korištenjem postojećih komponenti* te se pojam komponentnog usmjerenog razvoja u konkretnim situacijama najprije veže uz korištenje postojećih komponenti. Sustav se integrira višestrukom uporabom postojećih komponentata ili uporabom komercijalnih, gotovih komponentata (engl. *COTS - commercial-of-the-shelf*).

Komponentni model predstavlja definiciju norme za implementaciju, dokumentaciju i isporuku komponenti. Normiranje je važno i za razvojno osoblje kao i za osoblje podrške i održavanja infrastrukture projekta da bi se osigurala nesmetana ugradnja i komunikacija komponenti. To znači da bi norma komponentnog modela za svaku komponentu trebala uključivati sljedeće informacije:

- Definiciju sučelja koja uključuje nazive operacija koje komponenta obavlja, parametre koje pri tome prima, opise iznimaka i jezik kojim se sučelje definira. Npr., za komponente web usluge jezik je WSDL, za Enterprise JavaBeans to je Java, za

.NET to je Component Intermediate Language (CIL), već ovisno o specifičnosti infrastrukture komponentnog modela.

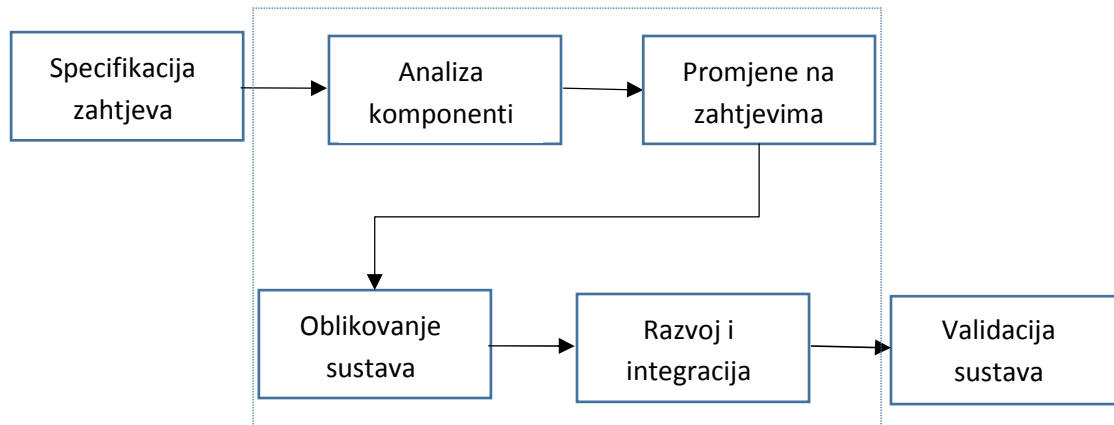
- Uputu za korištenje koja najprije pretpostavlja jedinstveni naziv komponente po kojem joj se može pristupiti, zatim informacije o sučeljima i atributima te način na koji se komponenta može primjenjivati za različite usluge.
- Uputu za isporuku koja specificira način pripreme komponente kao nezavisnog izvršnog entiteta. To znači da sa sobom mora nositi svu prateću podršku koja ne mora nužno biti osigurana u ciljnoj infrastrukturi niti je eksplicitno tražena od strane svojeg zahtijevanog sučelja. Osim toga, dobro bi bilo da uključuje opis uvjeta za daljnju nadogradnju ili zamjenu prema promjenama korisničkih zahtjeva te bi svakako trebala uključivati temeljnu dokumentaciju koja opisuje sadržaj i unutarnju organizaciju komponente kako bi se lakše identificirala kao komponenta koja je prihvatljiva za određenu namjenu.

Skoro svaki projekt izrade programske potpore danas ima elemente razvoja s korištenjem postojećih komponenti. Ako već nije tako unaprijed etiketirano, ponovno iskorištavanje postojećeg kôda se događa često i neformalno, tj. dizajneri koriste ono za što znaju da postoji a da odgovara ili je slično onome za što postoji potreba. Tako se u programerskoj praksi po pitanju granulacije jedinica programske potpore princip ponovne uporabe primjenjuje na različitim razinama:

- gotove aplikacije koja se (do određene mjere) može prilagoditi različitim korisnicima,
- komponenta aplikacije čija se veličina može kretati od njenih većih dijelova ili podsustava do pojedinih modula ili objekata,
- objekata predstavljenih svojim razredima i pojedinačnih funkcija ili procedura kao najniže razine apstrakcije.

S druge strane, za one projekte izrade programske potpore koji se u svojem pristupu fokusiraju upravo na komponentno-usmjereni model, moguće je identificirati nekoliko procesnih faza, slika 5.8. Uvodna faza, specifikacija zahtjeva, i završna faza, validacija sustava, mogu se usporediti sa sličnim fazama i drugih procesnih modela dok su ostale faze potpuno specifične za komponentno-usmjereni model. To su:

- **Analiza postojećih komponenti.** Traži se pogodna komponenta za specifičnu namjenu. Obično se pronađu komponente koje ispunjavaju samo dio traženih funkcionalnosti.
- **Promjene na postavljenim zahtjevima.** Sa saznanjem karakteristika pronađenih komponenti (koje ne moraju nužno ispunjavati sve zahtjeve potrage) ponovno se analiziraju zahtjevi primjene te, ako je to moguće, mijenjaju ne bi li bilo moguće koristiti nađene komponente.



**Slika 5.8.** Proces programskog inženjerstva temeljen na ponovnoj iskoristivosti komponenti.

- Oblikovanje sustava. Uzimajući u obzir dostupne komponente oblikuje se struktura sustava. Za pojedine zahtjeve definiraju se nove komponente kao i elementi integracije sustava.
- Razvoj i integracija. Programska potpora koja nedostaje se razvija te se zajedno s pribavljenim komponentama integrira u novi sustav.

Više je prednosti i dobitaka kod primjene komponentno-usmjerenog modela. Najprije, povećana je pouzdanost komponente koja je već prošla neki oblik ispitivanja ili je već bila dijelom nekog funkcionalnog sustava u odnosu na onu koja to tek treba proći. Kod planiranja troška izgradnje sustava olakšana je, bolja i jasnija procjena troška. Za pretpostaviti je da su komponente rađene i potvrđene za višestruku uporabu ostvarene od strane specijalista razvoja programske potpore s takvim ciljem, a ne u sklopu usputnih potreba drugih projekata. Isto tako, specijaliziranija okruženja pretpostavljaju i bolje poklapanje sa zadanim normama izrade komponente. Na kraju, ono što je vjerojatno najvidljiviji dobitak za voditelje projekata je značajno ubrzanje u razvoju sustava kad se uzme gotova komponenta. S druge strane, postoje i problemi koji se mogu pojaviti kod primjene koncepta ponovne uporabe komponenti. Ako izvorni programski kôd nije dostupan, troškovi njenog održavanja mogu značajno porasti ako postane nekompatibilna s promjenama u sustavu. Ako se koriste i oni alati programske potpore koji ne predviđaju ili ne podržavaju koncept ponovne uporabnosti komponenti, to može biti poseban problem pri integraciji takvih alata zajedno s komponentama izrađenim za ponovnu uporabu. Problem se može očitovati u nemogućnosti integracije takvih alata sa sustavom knjižnice potrebnih komponenti.

Posebni problem je sklonost inženjera da napišu svoj kôd umjesto da koriste tuđi. Često se tu radi o vjeri da se uvijek može izraditi bolja i naprednija komponenta a i o određenom sindromu programerskog intelektualnog izazova. Osim toga, troškovi održavanja knjižnice komponenata mogu biti visoki. Knjižnicu komponenata treba stvoriti i održavati a treba i

osigurati da njeno korištenje bude podržano unutar čitavog razvojnog procesa. Poteškoću može činiti i napor pronalaženja, razumijevanja i prilagodbe komponenata iz vanjskih knjižnica pri čemu treba donositi utemeljene odluke.

Pri odluci hoće li se i u kojoj mjeri pribjeći razvoju koji bi bio srodan ili se temelji na konceptu ponovne uporabe komponenti, nekoliko je vodećih čimbenika. To su:

- Prije svega, vremenski okvir u kojem se očekuje izrada programskog proizvoda utječe na odluku za izbor gotovih komponenti ili čitavog sustava.
- Očekivana dugovječnost programskog proizvoda.
- Razina znanja, vještina i iskustva razvojnog tima.
- Strogost okoline za koju je programska potpora namijenjena i razina performansi koja se očekuje.
- Osobitosti domene primjene programske potpore.
- Izbor platforme za koju je programska potpora predviđena.

Tijekom godina napredaka u razvoju programske potpore razvijene su mnogi koncepti i pristupi koji u nekoj mjeri temelje, uključuju ili podupiru koncept ponovne iskoristivosti programske potpore (osim zasebno definirane grane komponentno-usmjerenog programskog inženjerstva). Kao jedan od najizraženijih proizvoda predstavnika ovog koncepta su **knjižnice programskih komponenata** (engl. *program libraries*) koje za pojedine tehnologije čine norme ili dolaze nezavisno za slobodnu ugradnju u vlastiti programski proizvod. Na višoj razini apstrakcije su se tako prije svega definirali **arhitekturni i oblikovni obrasci** kao apstrakcije pri oblikovanju i ostvarenju sustava. Arhitekturni obrasci se posebno obrađuju unutar kolegija Oblikovanje programske potpore kao opći tipovi i temelji oblikovanja sustava programske potpore, dok oblikovni obrasci (engl. *design patterns*) predstavljaju opće apstrakcije odnosa između konkretnih objekata koje se češće primjenjuju na razini implementacije sustava. Također su to sustavi orijentirani pružanju usluga (engl. *service-oriented systems*), gdje se sustavi ostvaruju povezivanjem privatnih i javno dostupnih dijeljenih usluga. Posebno mjesto zauzimaju i veliki poslovno temeljeni sustavi (engl. *ERP systems*) koji u jednom sadrže opću poslovnu logiku i njoj srodne funkcionalnosti. Kao poseban koncept se razvilo i modelno-usmjereno inženjerstvo (engl. *model-driven engineering*) kao metodologija razvoja programske potpore koja se temelji na modelima kao apstraktnim reprezentacijama znanja iz određene domene primjene. Tu su još i generatori programske potpore (engl. *program generators*) na temelju znanja određene vrste aplikacije i modela sustava specificiranog od strane korisnika generiraju konačni sustav. Kao jedna posebna grana programskog inženjerstva se predstavlja i *aspektno-orijentirani razvoj programske potpore* (engl. *aspect-oriented software development*) koji niz dostupnih dijeljenih komponenti isprepliće na potrebnim mjestima

tijekom prevođenja programa a temeljen je na razdvajanju područja odgovornosti (engl. *separation of concerns*).

U raznolikosti oblika ponovne iskoristivosti elemenata programske potpore nekoliko se njih može izdvojiti kao većem broju razvojnih inženjera bliži pristupi. To su:

1. **(računalni) razvojni radni okviri (engl. *application framework*),**
2. **primjenske linije proizvoda (engl. *software product line*) i**
3. **gotovi proizvodi programske potpore (engl. *Commercial-Off-The-Shelf, COTS*)**

U narednim će se odlomcima još detaljnije analizirati ovi pristupi. Prvi, razvojni radni okviri, se temelje na doprinosima objektno-orijentiranog pristupa. Drugi, primjenske linije proizvoda, se temelje na iskustvima dugotrajnijeg razvoja i izlučivanja komponenti s generičkim svojstvima dok kod trećeg, gotovih programskih proizvoda, najčešće misli na komercijalno dostupne komponente.

#### 5.4.2. Razvojni radni okviri

Objektno orijentirani pristup unosi pretpostavke ponovne iskoristivosti razreda i metoda kao elemenata koje mu čine temelj. Međutim, granulacija takvih elemenata je premala da bi njihova intenzivna uporaba bila isplativa. Najčešće su takvi elementi poprilično specifični da bi ih se lako razumjelo i upotrijebilo na najrazličitijim mjestima pa često ispada da ih je lakše ponovno implementirati nego prilagoditi svojim potrebama. Zato se koncept ponovne iskoristivosti u objektnom orijentiranom pristupu više rabi na razini apstrakcije **razvojnih radnih okvira** kao struktura koje u jednom obuhvaćaju brojnije skupove artefakata (razreda, objekata, komponenti) programske potpore. Takvi radni okviri u sebi pretpostavljaju podršku programima koji nose određene sličnosti da bi im se u jednom trenutku mogla dati podrška u zajedničkim generičkim funkcionalnostima. Zato su radni okviri često i specijalizirani za određene aspekte u funkcionalnostima programa koje se razvijaju. Npr. postoje radni okviri za ostvarenje korisničkih sučelja pa stoga u sebi uključuju podršku za prihvatanje i obradu korisničkih akcija u sučelju (engl. *event handling*) i skupove gotovih prozorčića ili okvira (engl. *widgets*) za prikaz.

Razvojni radni okviri prema tome nude okosnicu (engl. *skeleton*) buduće arhitekture aplikacije što u sebi uključuje i višestruko iskorištenje elemenata s generički istom svrhom. To se konkretno svodi na skupove razreda kao artefakte koji su vidljivi i na raspolaganju za daljnju uporabu i primjenu zasada objektno orijentiranog pristupa. Prema tome, takvi su razvojni radni okviri potpuno vezani uz pojedine programske jezike. Međutim, mogući su i složeniji razvojni radni okvir koji uključuju druge manje radne okvire namijenjene za zasebne izrade pojedinih dijelova aplikacije.

Određene klasifikacije razlikuju nekoliko vrsta razvojnih radnih okvira:



- Radni okviri infrastrukture sustava (engl. *system infrastructure frameworks*). Potpomažu razvoj infrastrukture sustava koja uključuje operacijske sustave, komunikacijske mehanizme, korisnička sučelja i prevoditelje programskih jezika. Prvenstveno se koriste unutar same organizacije i nisu predviđeni za marketinšku prodaju.
- Integrativni radni okviri (engl. *middleware integration frameworks*). Koriste se za integraciju komponenata u raspodijeljenim aplikacijama. Predstavljaju skup normi i pripadajućih im objekata za ostvarivanje komunikacije i razmjene informacija. Oblikovani su s namjerom pomoći programerima pri modularizaciji i upravljanju infrastrukturom u raspodijeljenim okruženjima. Općenito se tu misli na ORB (engl. *object request broker*) temeljene radne okvire, one koji se temelje na razmjeni poruka i za transakcijske baze podataka. Konkretni primjeri takvih radnih okvira kao Microsoft .NET i Enterprise JavaBeans (EJB) daju podršku za svoje normirane komponentne modele.
- Poslovni radni okviri (engl. *enterprise application framework*). U pravilu su fokusirani na domenu primjene i temelj su razvoja korisničkih programa za veće poslovne sustave. U odnosu na radne okvire infrastrukture sustava i integrativne radne okvire, razvoj temeljen na poslovnim radnim okvirima, kao i konačni proizvod, je skuplji jer je izravnije podržan razvoj korisničkih aplikacija kroz prethodno ugrađeno znanje iz domene primjene. Za razliku od ovih prije navedenih, poslovni radni okviri manje su fokusirani na interna pitanja razvoja programske potpore, ali im je dugoročna dobit veća.

U novije vrijeme su, kao zasebna vrsta, sve više prisutni razvojni okviri za razvoj web aplikacija (engl. *web application frameworks - WAFs*). Najčešće su zasnovani na *Model-View-Controller (MVC)* obrascu s idejom razdvajanja prezentacije objekta kroz različite dijelove aplikacije od interne pohrane do njegovog prihvata i prezentacije prema korisniku te uključuju mehanizme interakcije između različitih prikaza.

Radni okviri za razvoj web aplikacija obično u sebi uključuju jedan ili više manjih radnih okvira koji su specijalizirani za određene aspekte programske potpore kao što su: sigurnost, podrška za dinamičke web stranice, rad s bazom podataka, upravljanje korisničkim akcijama i sjednicama (engl. *session management*). Obično se radi o skupovima predefiniranih razreda za pojedine aspekte pa se posao svodi na korisnička proširenja dodavanjem novih razreda koji nasljeđuju razrede ponuđene unutar radnog okvira. Aplikacije razvijene na taj način mogu biti dobra osnova za daljnji razvoj različitih primjenskih proizvodnih linija specifičnije namjene jer se posao svodi na specijalizacije strukture općenitog radnog okvira kroz izmjene postojećih razreda i metoda.

Razvoj temeljen na općenitim radnim okvirima može biti vrlo učinkovit, ali i poprilično složen i vremenski zahtjevan ako se želi ući u dubinu strukture radnog okvira što je obično

potrebno za sveobuhvatnije razumijevanje i za naknadne popravke ako se kod implementacije nešto previdjelo.

### 5.4.3. Primjenske linije proizvoda

U odnosu na razvojne radne okvire kao jednom pristupu koji podržava koncept ponovne uporabe komponenti, **primjenske linije proizvoda** imaju izvor u drugačijem pogledu na potencijalnu iskoristivost programskih artefakata. Dok razvojni okviri “nude” knjižnice generičkih programskih komponenti u uvjerenju da će biti korištene za široku lepezu aplikacija, linije proizvoda se sastoje od artefakata čija je potreba u dobroj mjeri potvrđena i predviđena u već solidno definiranom proizvodnom procesu. Može se relativno lako zamisliti situacija nastanka proizvodnih linija iz postojeće programske potpore. Ako, npr., neka organizacija razvije vlastitu programski proizvod i pojavi se potreba drugih sličnih proizvoda, najbrži zadovoljavajući pristup je ponovna uporaba prethodno razvijenog programskog kôda. Učestalijom primjenom takvog pristupa ona osnovna zamišljena struktura kôda se s vremenom narušava te je smisljena odluka o sačuvanju generičke strukture kôda kroz definiciju nove zasebne proizvodne linije. Uzimaju se opće funkcionalnosti s namjerom olakšavanja njihove ponovne uporabe u budućem razvoju novih programskih proizvoda. Na taj način se, osim izabranih dijelova kôda, u novodefinirani primjenski okvir prenosi i iskustvo prethodnog razvoja te je u odnosu na radne okvire s generičkom jezgrom skraćeno vrijeme uključivanja novih razvojnih inženjera u razvojni proces i bitno potpomognut proces ispitivanja buduće programske potpore.

U odnosu na razvojne radne okvire mogu se istaknuti neke značajne razlike za primjenske linije proizvoda:

- Razvojni radni okviri u većini slučajeva počivaju na objektno orijentiranoj strukturi koja se proširuje da bi se ostvarila specifičnija primjena. Sâm programski kôd te strukture se pritom u pravilu ne mijenja. Primjenske linije proizvode nisu toliko zasnovane na objektno orijentiranoj paradigmi, a kod nadogradnji radi ostvarivanja specifičnijih funkcionalnosti njihov programski kôd se mijenja i nadopunjuje.
- Primjenske linije proizvoda su više orijentirane na domenu primjene nego razvojni radni okviri koji više predstavljaju tehničku potporu razvoju novih programskih proizvoda. Postoje specifičnija podjela razvojnih radnih okvira (kako je već opisano u prethodnom poglavlju) pa tako i npr. razvojni okviri za razvoj web programskih proizvoda dok primjenske linije proizvoda u svom samom temelju uključuju detalje domene primjene i implementacijske platforme. Tako npr. postoje i linije proizvoda za upravljanje radom knjižnice (s knjigama) temeljeno na pristupu webu.
- S obzirom da u sebi već pretpostavljaju detalje radne platforme svog proizvoda, primjenske linije proizvoda isto tako nerijetko uključuju izravnije upravljanje sklopovljem sustava. Tako postoje i primjenske linije proizvoda namijenjene za

upravljanje radom naglašeno sklopovskih sustava (kao npr. nekih porodica pisača). Razvojni radni okviri su u pravilu dosta udaljeni od sklopovlja te su više definirani programerskom tehnologijom.

- Primjenske linije proizvoda predstavljaju skupove povezanih programskih proizvoda unutar posjeda jedne poslovne organizacije. Razvoj novog programskog proizvoda započinje od nekog programskog proizvoda iz te porodice koji su po namjeni ili strukturi najbliži budućem novom proizvodu. Kod razvojnih radnih okvira najčešće se kreće od generičke jezgre programskog kôda izabranog razvojnog radnog okvira što može biti početak razvoja proizvoda koji će biti temelj nove primjenske linije proizvoda. Razvoj specifičnih ogranaka koji će činiti novu primjensku liniju proizvoda dolazi tek nakon izrade tog temeljnog programskog proizvoda.

Iako primjenske linije proizvoda u sebi uključuje više programskih proizvoda s različitim namjenama moguće je izdvojiti određene skupove specijalizacija primjenskih linija proizvoda:

- S obzirom na platformu. Razlikuju se primjenski proizvodi s obzirom na sklopovsku platformu i operacijski sustav. Tako npr. mogu postojati inačice istog proizvoda za operacijske sustave Windows, Linux i Mac OS s nepromijenjenom temeljnom funkcionalnošću.
- S obzirom na radnu okolinu. Uvjeti radne okoline i vrsta uređaja s kojom se komunicira određuju različite inačice programskog proizvoda unutar iste linije proizvoda. Tako npr. sustav za potporu rada hitne pomoći može postojati u različitim inačicama s obzirom na komunikacijski sustav vozila hitne pomoći.
- S obzirom na funkciju. Inačice programskog proizvoda se određuju prema vrsti korisnika. S obzirom na različite zahtjeve korisnika rade se određene funkcijske specijalizacije. Npr. sustav potpore za rad knjižnice može imati inačice ovisno radi li se o javnoj, sveučilišnoj ili internoj knjižnici neke organizacije.
- S obzirom na poslovni proces. Primjenski proizvod se oblikuje prema specifičnostima poslovnog procesa. Npr. mogu se razlikovati inačice koje podržavaju centralizirane i one koje podržavaju raspodijeljene procesne okoline.

S obzirom na mogućnosti razgranatosti inačica, podrazumijevaju se određeni stupnjevi podesivosti linije proizvoda kroz dodavanja i izbacivanja komponenti iz sustava, parametriziranja funkcionalnosti, definiranja radnih ograničenja komponenti i uključivanje znanja o poslovnom procesu. Takva podešavanja se mogu događati u različitim fazama razvoja, ali se mogu svesti na ona koja se događaju tijekom same izrade proizvoda i na ona koja se rade prilikom ili nakon isporuke proizvoda. Kod prvih se radi o izravnim promjenama jezgre linije proizvoda kroz razvoj, izbor i prilagodbu komponenti koje će čini novi proizvod, a kod drugih o razvoju malo više generičkog sustava koji će se moći

podešavati od strane korisnika ili dodijeljenog mu konzultanta. Kod ovih drugih se pretpostavlja da je znanje specifičnosti korisničkih zahtjeva i radne okoline na prikladan način uključeno u obliku konfiguracijskih postavki koje su nadohvat vanjskom korisniku.

#### 5.4.4. Gotovi programski proizvodi

Pod gotovim programskim proizvodima ili sustavima (engl. *commercial-of-the-shelf* - COTS product reuse) se podrazumijevaju oni koje nije potrebno iznutra (a nekad ni moguće) poznavati niti je moguće mijenjati programski kôd da bi ga se koristilo u drugom programskom sustavu. Općenito tu spada sva programska potpora osobnih računala i poslužitelja. Često ta programska potpora ima mnoge funkcionalnosti i načine rada pa stoga i potencijale za često korištenje u različitim razvojnim okolinama i kao sastavnica mnogih programskih proizvoda. Zanimljivo je da se na taj način često shvaća i koristi i programska potpora otvorenog koda (engl. *open-source*) gdje je programski kôd potpuno na raspolaganju za promjene.

Kod gotove programske potpore koja ima specifičniju namjenu podrazumijeva se i postojanje mehanizama podesivosti za upotrebu funkcionalnosti prema korisničkim potrebama ili postojanje dodataka (engl. *plug-in*) koji proširuju izvorno zamišljene funkcionalnosti.

U odnosu na razvoj programske potpore prema zahtjevima korisnika i točno određenoj namjeni, upotreba gotove programske potpore prije svega skraćuje vrijeme razvoja sustava uz isključivanje mnogih rizika dugotrajnijeg razvoja. U tom slučaju je omogućeno lakše se usredotočiti na poslovne procese koji čine jezgru postojanja neke organizacije jer nije potrebno voditi brigu o posebnom razvojnom timu. U dobro dokumentiranim gotovim programskim paketima jasnije se iščitavaju moguće funkcionalnosti i stvarne mogućnosti uporabe takve programske potpore, što uz razmjenu iskustava s prethodnim korisnicima ubrzava procjenu njezine prikladnosti za vlastite potrebe. Isto tako, briga oko nadogradnji programskih proizvoda koje nameće napredak tehnologije pada na isporučitelja proizvoda, a ne na korisnika.

S druge strane, ugrađene funkcionalnosti i predefinirani načini rada gotove programske potpore su nepromjenjivi pa su moguće situacije da se korisnički zahtjevi moraju prilagođavati ne bi li se takva programska potpora koristila. Sama izvorna svrha izrade neke programske potpore u pravilu se ne mora poklapati s temeljnim ciljevima sustava u kojem se želi koristiti pa se s razlikama koje iz toga proizlaze treba pomiriti i naći prihvatljiv presjek zajedničkih funkcionalnosti za njezino smisleno iskorištavanje. Stoga i sâm izbor najprikladnijeg programskog paketa za određenu namjenu može biti zahtjevan posao da bi se minimizirali negativni efekti korištenja takve gotove programske potpore. Činjenica da je održavanje programske potpore odgovornost isporučitelja ima i svoju nepovoljnu stranu u određenoj ovisnosti o isporučitelju i vanjskim konzultantima. Takav nedostatak unutarne

ekspertize za održavanje proizvoda manje vodi prema interesima korisnika u odnosu na interese isporučitelja ili prodavača gotove programske potpore. Posebna nepovoljnost pri korištenju vanjske gotove programske potpore se događa kod većih poslovnih promjena na strani isporučitelja, kao što su promjena poslovnog modela, izlazak iz posla ili preuzimanje od strane druge tvrtke.

Uopće, među različitim modelima korištenja gotove programske potpore razlikuju se dva krajnja principa: korištenje sustava gotovog rješenja (engl. *COTS-solution system*) i korištenje sustava integriranog rješenja (engl. *COTS-integrated system*). Kod prvog se pretpostavlja korištenje programskog proizvoda razvijenog od strane jednog isporučitelja gdje je njihova generička aplikacija podešena prema zahtjevima korisnika. Kod drugog se radi o korištenju dva ili više gotova programska proizvoda (razvijenih od strane različitih isporučitelja) koji su ujedinjeni u sustavu na način da su zahtjevi korisnika zadovoljeni. S obzirom da se prvi temelji na određenom podešavanju nekog više generičkog rješenja i standardiziranom poslovnom procesu, a drugi na integraciji sustava iz više različitih komponenti, kod potonjeg u temelju postoji i veća fleksibilnost uporabe za korisnikove potrebe. Kod prvog su poslovi održavanja sustava i njegove radne platforme u većoj odgovornosti na strani isporučitelja, a kod drugog na strani vlasnika, odnosno korisnika sustava.

## 5.5. Modelno-usmjereni razvoj (RUP)



Modelno-usmjereni razvoj (engl. *Rational Unified Process*, RUP) je metodologija razvoja programske potpore koja se temelji na oblikovanju pomoću modela (engl. *Model Based Design*, MBD), odnosno iterativnom razvoju, obrascima uporabe i usmjerenjem na arhitekturu sustava.

Razvoj RUP-a započeo je ranih 1980-ih u tvrtci Ericsson kao odgovor na potrebu za novom projektnom metodologijom koja će optimalno odgovarati različitim potrebama tvrtke za razvojem novih proizvoda u programskom inženjerstvu, ali i ne samo u njemu. Prema verziji RUP-a koja je završena krajem 90-ih, određeno je da opseg i aktivnosti ove metodologije uključuju inženjerstvo poslovnog procesa, rukovanje zahtjevima, rukovanje oblikovanjem i promjenama, funkcijsko ispitivanje, vrednovanje performansi, inženjerstvo podataka te oblikovanje sučelja. Nakon 2003. godine razvoj RUP-a nastavljen je u tvrtci Rational Software unutar IBM-a.

Važno je naglasiti da ne postoji univerzalno optimalan proces oblikovanja programske potpore. Svaki pristup ili metodologija imaju svoje komparativne prednosti i nedostatke. Stoga se tijekom razvoja RUP-a vodilo računa o fleksibilnosti i budućim proširenjima, jer se time omogućuju razne strategije životnog ciklusa projekta. Naprimjer, moguće je odabrati koje artefakte treba proizvesti, definirati nužne aktivnosti i potrebne resurse (ukupni

trošak, potrebno vrijeme, nužni programeri, vještine i znanja, programska podrška, sklopovlje, itd.), te modelirati koncepte sustava.

Kao i svaka druga projektna metodologija, RUP određuje faze životnog ciklusa (engl. *lifecycle*) procesa i dokumente koji se moraju izraditi završetkom izvođenja svake faze. Faze životnog ciklusa RUP-a su:

- 1) **početak (engl. *inception*)**, u kojemu se definira doseg projekta, razvoj modela poslovnog procesa, specifikacija početnih zahtjeva,
- 2) **razrada (engl. *elaboration*)**, kada se definiraju plan projekta, specifikacija značajki i temelji arhitekture sustava,
- 3) **izgradnja (izrada, engl. *construction*)**, koja se sastoji od oblikovanja, programiranja, i ispitivanja, te
- 4) **prijenos (engl. *transition*)**, pri čemu se završni proizvod prenosi korisnicima i postavlja u radnu okolinu.

Završne, odgovarajuće ključne točke ovih faza su, redom:

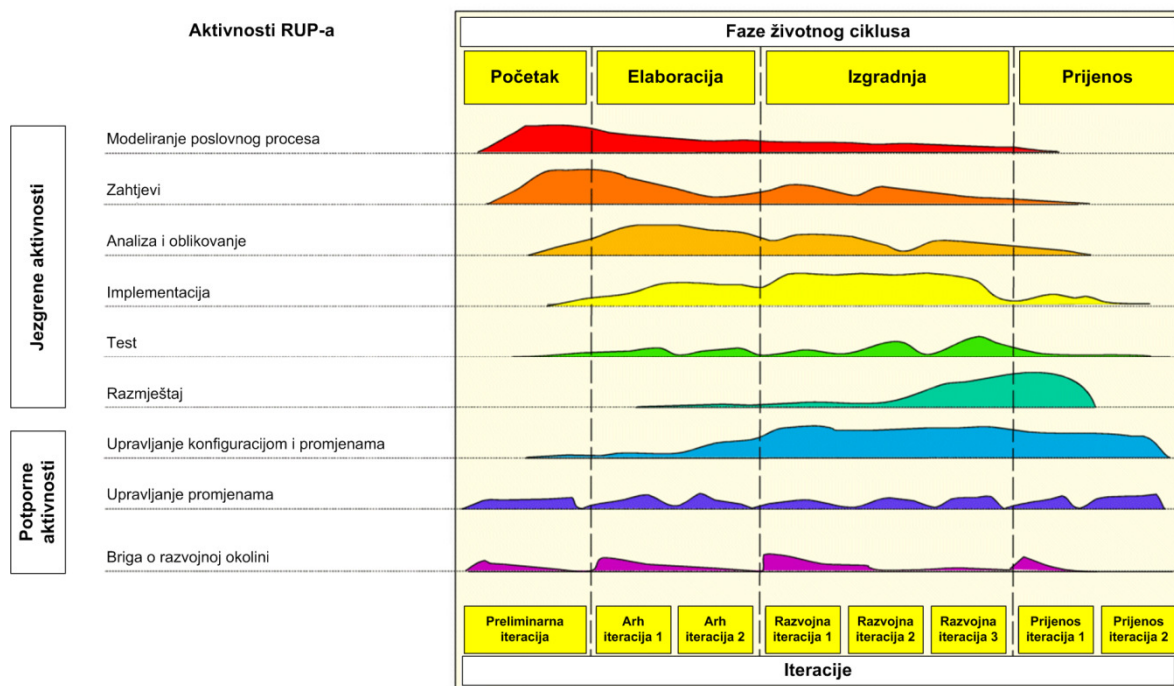
- 1) vizija ili ciljevi životnog ciklusa (engl. *lifecycle objectives*),
- 2) temeljna arhitektura (engl. *lifecycle architecture*),
- 3) početna sposobnost (engl. *initial operational capability*) i
- 4) izdanje izvršne inačice programa ili proizvoda (engl. *release*).

Svaka faza ima barem jednu iteraciju. Stoga, u kontekstu RUP-a, iteracija je niz ili sekvenca aktivnosti u okviru prihvaćenog plana i kriterija evaluacije. Ishod svake iteracije je dokument, a na kraju i jedna izvršna inačica (tj. izdanje) programa ili sustava.

Osim faza životnog ciklusa, RUP definira i niz aktivnosti koje se mogu odvijati u svim fazama, ali obično s različitim intenzitetom. **Jezgrene aktivnosti** (engl. *core workflows*) RUP-a su:

- 1) **modeliranje poslovnog procesa (engl. *business modelling*)**,
- 2) **zahtjevi (engl. *requirements*)**,
- 3) **analiza i oblikovanje (engl. *analysis and design*)**,
- 4) **implementacija (engl. *implementation*)**,
- 5) **ispitivanje (engl. *test*) i**
- 6) **razmještaj (engl. *deployment*)**.

U nekim verzijama RUP-a, skup aktivnosti može biti drugačije definiran. Vrlo često se dodaju dodatne, potporne aktivnosti vezane uz organizaciju i provedbu životnog ciklusa,



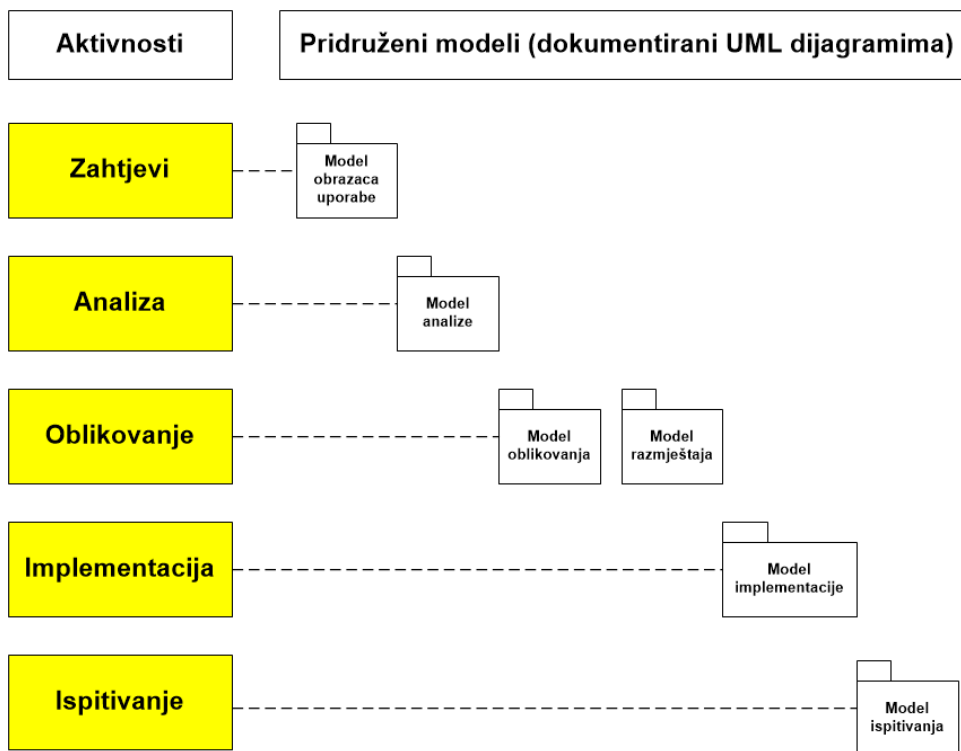
**Slika 5.9.** Faze i aktivnosti, s pretpostavljenim intenzitetom životnog ciklusa modelno-usmjerenog razvoja.

kao što su: upravljanje konfiguracijom i promjenama (engl. *configuration and change management*), upravljanje projektom (engl. *project management*), te briga o razvojnoj okolini (engl. *environment*; svodi se na ispravnu interakciju s dionicima projekta). Ove dodatne aktivnosti se zbog svoje namjene zajedno nazivaju **potporne aktivnosti** (engl. *support workflows*). Ovisno o fazi u kojoj se nalazi razvoj proizvoda, bit će potrebno dodijeliti više ljudi (projektnih resursa) na određene aktivnosti, a manje na druge i obrnuto. Određivanjem prioriteta i raspoređivanjem resursa bavi se voditelj projekta (engl. *project manager*). Faze i aktivnosti s preporučenim optimalnim razinama intenziteta izvršenja prikazani su na slici 5.9. Kao što je već navedeno, osim što se u svojoj strukturi temelji na iteracijama u svakoj pojedinoj fazi, RUP intenzivno koristi obrasce uporabe (engl. *use cases*) i usmjeruje se na arhitekturu sustava, odnosno na strukturu programske podrške koja se izrađuje. Stoga je bitno definirati pojam arhitekture programske potpore.



**Definicija 5.2.** Arhitektura programske potpore je struktura ili strukture sustava koje sadrži elemente, njihova izvana vidljiva obilježja i odnose između njih.

Obrasce uporabe su vrlo važni u modelno-usmjerenom razvoju, jer povezuju i pokreću brojne aktivnosti u životnom ciklusu oblikovanja programske potpore, kao što su: kreiranje i validacija arhitekture sustava, definicija ispitnih slučajeva, scenarija i procedura, planiranje pojedinih iteracija, kreiranje korisničke dokumentacije, razmjesta (engl. *deployment*) sustava, te pomažu u sinkroniziranju sadržaja različitih modela.



**Slika 5.10.** Primjer aktivnosti i pridruženih UML modela u modelno-usmjerenom razvoju.

Za opis pojedinih aktivnosti koriste se odgovarajući modeli. Modeli su dokumentirani s jednim ili, u praksi češće, više dijagrama programske potpore. Dijagrami moraju biti definirani UML standardom. Stoga, prema RUP-u, oblikovana arhitektura sustava sadrži skup UML dijagrama kojima su opisani različiti pogledi (engl. *views*) u modele sustava. Svakoj aktivnosti pridružen je jedan ili više modela za opis, koji su dokumentirani s jednim ili više dijagrama (tj. pogleda), slika 5.10. RUP je najčešće opisan kroz tri perspektive koje se ogledavaju i u korištenim UML dijagramima:

- 1) dinamička perspektiva, koja pokazuje slijed faza procesa kroz vrijeme,
- 2) statička perspektiva, koja pokazuje aktivnosti u pojedinim fazama procesa, i
- 3) praktična perspektiva, koja sugerira aktivnosti kroz iskustvo i dobru praksu.

## 5.6. Agilni pristup razvoju programske potpore

Kako bi odgovorile na dinamičnost suvremenog tržišta, tvrtke moraju moći brzo reagirati i prilagoditi se promjenama. Budući da se programska podrška nalazi u središtu svake poslovne operacije, razvoj novih poslovnih aplikacija mora moći pratiti zahtjeve tržišta. Pri tome se kao najvažniji zahtjev često nameće mogućnost isporuke programskog proizvoda u što kraćem vremenu. Budući da je poslovno okruženje toliko dinamično i promjenjivo, nije moguće potpuno definirati sve zahtjeve na samom početku razvoja programskog proizvoda. Stoga standardni modeli razvoja programske potpore, koji imaju jasno definirane i odvojene faze razvoja, nisu adekvatni i često ne mogu udovoljiti zahtjevima.



Potreba za novim metodama razvoja programske potpore, koje će moći odgovoriti na novonastalu situaciju na tržištu, dovela je do razvoja agilnog pristupa razvoju programske potpore [8].



**Definicija 5.3.** *Agilni pristup razvoju programske potpore podrazumijeva skupinu metoda za razvoj programske potpore kojima je zajednički iterativni razvoj uz male inkremente i brz odziv na korisničke zahtjeve. Ovaj model razvoja programske potpore koristi se za razvoj manjih i srednjih projekata u stalnoj interakciji s klijentima putem stalnog predočavanja novih poboljšanja, uz relativno slabo dokumentiranje.*

Principi agilnog pristupa izneseni su u proglasu (*Agile Manifesto*) [7] koji je sastavilo 17 istaknutih programskih inženjera u SAD-u 2001. Izvorni tekst proglasa nalazi se u nastavku:

### ***Manifesto for Agile Software Development***

*"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*

***Individuals and interactions*** over processes and tools

***Working software*** over comprehensive documentation

***Customer collaboration*** over contract negotiation

***Responding to change*** over following a plan

*That is, while there is value in the items on the right, we value the items on the left more."*

Objašnjenje značenja pojmova na lijevoj strani svake rečenice u manifestu je sljedeće:

- Pojedinci i njihovi međusobni odnosi – u agilnom pristupu važni su samoorganizacija, motivacija i interakcije među pojedincima u timu.
- Upotrebljiva programska potpora – važnije je naručitelju isporučiti programski proizvod koji je odmah upotrebljiv, nego sastaviti iscrpnu dokumentaciju.
- Suradnja s naručiteljem – budući da svi zahtjevi nisu poznati odmah na početku, suradnja s naručiteljem i budućim korisnicima je neizostavna.
- Reagiranje na promjenu – promjene u zahtjevima su stalne i na njih se mora odgovoriti u što kraćem roku.

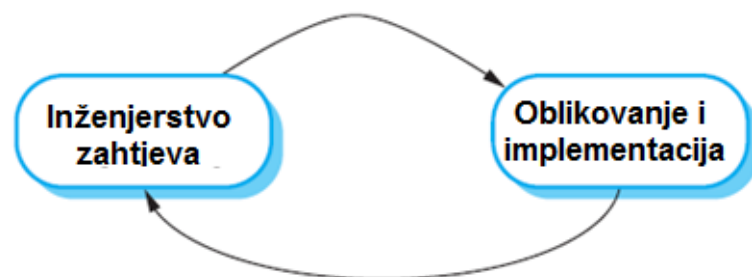
U proglasu je nadalje izneseno dvanaest temeljnih načela agilnog razvoja:

1. Najvažnije nam je zadovoljstvo naručitelja koje postižemo ranom i neprekinutom isporukom programskog proizvoda koji donosi vrijednost.

2. Spremno prihvaćamo promjene zahtjeva, čak i u kasnoj fazi razvoja. Agilni procesi uprežu promjene da naručitelju stvore kompetitivnu prednost.
3. Često isporučujemo upotrebljiv programski proizvod, u razmacima od nekoliko tjedana do nekoliko mjeseci, nastojeći da razmak bude čim kraći.
4. Poslovni ljudi i razvojni inženjeri moraju svakodnevno zajedno raditi, tijekom cjelokupnog trajanja projekta.
5. Projekte ostvarujemo oslanjajući se na motivirane pojedince. Pružamo im okruženje i podršku koja im je potrebna, i prepuštamo im posao s povjerenjem.
6. Razgovor uživo je najučinkovitiji način prijenosa informacija razvojnom timu i unutar tima.
7. Upotrebljiv programski proizvod je osnovno mjerilo napretka.
8. Agilni procesi potiču i podržavaju održivi razvoj. Pokrovitelji, razvojni inženjeri i korisnici trebali bi moći neograničeno dugo zadržati jednak tempo rada.
9. Neprekinuti naglasak na tehničkoj izvrsnosti i dobrom oblikovanju pospješuju agilnost.
10. Jednostavnost – vještina povećanja količine posla koji ne treba raditi – je od suštinske važnosti.
11. Najbolje arhitekture, projektne zahtjeve i oblikovanje programske potpore stvaraju samo–organizirajući timovi.
12. Tim u redovitim razmacima razmatra načine da postane učinkovitiji i zatim usklađuje i prilagođava svoje ponašanje u tom smjeru.

Na slici 5.11 grafički je prikazan model koji predstavlja srž agilnog pristupa razvoju programske potpore. Razvoj se odvija kroz iteracije u kojima se neprekidno smjenjuju faze razrade zahtjeva za sljedeći inkrement u razvoju te implementacije funkcionalnosti tog inkrementa.

### **Agilni razvoj**



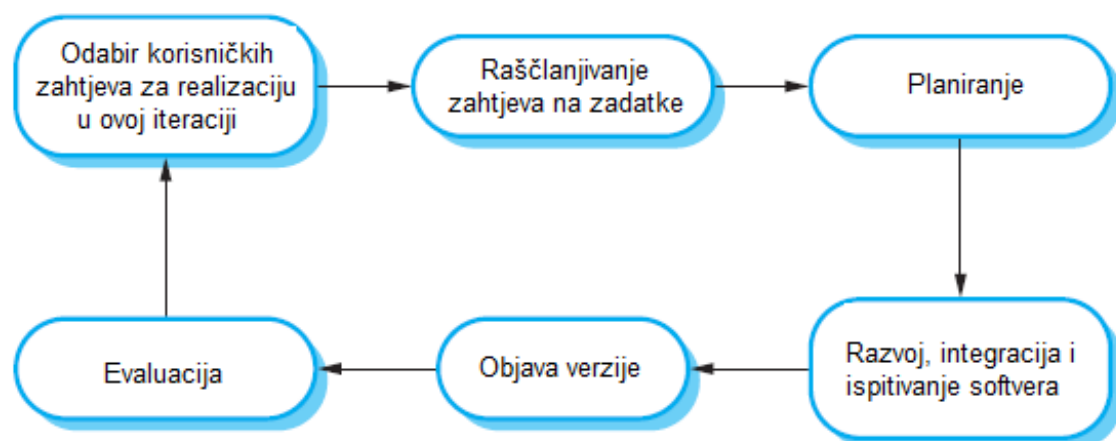
**Slika 5.11.** Model procesa agilnog razvoja programske potpore.

Pojam agilnog razvoja obuhvaća mnoštvo različitih metoda, od kojih su neke nastale i prije samog proglasa. Neke od njih su:

- Ekstremno programiranje (XP)
- Scrum
- Agilno modeliranje
- Adaptivni razvoj programske potpore (ASD)
- *Crystal clear* i ostale metode *crystal*.
- Metoda dinamičnog razvoja sustava (DSDM)
- Razvoj vođen karakteristikama (FDD)
- “Čisti” razvoj (engl. *lean development*)
- Agile Unified Process (AUP).

Ekstremno programiranje (XP) vjerojatno je jedna od najpoznatijih metoda agilnog pristupa. Javlja se sredinom 1990-tih kao otpor strukturiranom (ponajviše vodopadnom) procesu programskog inženjerstva za koji se smatra da unosi previše birokracije u proces oblikovanja. U ekstremnom programiranju jedina mjera napretka je funkcionalni programski proizvod. Ovaj pristup zasniva se na razvoju, oblikovanju i isporuci funkcionalnih dijelova. Sam postupak uključuje kontinuirano poboljšavanje koda i sudjelovanje korisnika/naručitelja u razvojnom timu. U razvoju sustava najčešće se programira u paru (jedno radno mjesto uz međusobno provjeravanje (engl. *pairwise programming*)). Na slici 5.12 prikazan je dijagram procesa razvoja programske potpore metodom ekstremnog programiranja.

Glavni nedostaci ovakvog pristupa su nestabilnost zahtjeva i nerazumljivost sustava zbog nedokumentiranja, što rezultira da postojeće rješenje uglavnom ne podupire ponovnu uporabu (engl. *reuse*) rješenja.



**Slika 5.12.** Model procesa agilnog razvoja programske potpore.

Čisti (*Lean*) razvoj programske potpore odnosi se na metodu koja je izvorno nastala u produkcijskom sustavu Toyote, a čiji je cilj razviti u što kraćem vremenu sustav koji će zadovoljiti korisnike. Ova metoda temelji se na sedam principa oblikovanja:

1. Eliminacija svega suvišnoga (koda, nejasnih zahtjeva, birokracije, spore interne komunikacije).
2. Naglašeno učenje (stalna komunikacija s klijentom, stalna testiranja i brze nadogradnje).
3. Odlaganje odluke (predlaganjem opcija klijentu, skupljanjem činjenica).
4. Dostava proizvoda što je brže moguće (niz metoda, uglavnom mali inkrementi, više timova radi isto).
5. Motivacija i suradnja unutar tima.
6. Ugradnja cjelovitosti u sustav (kupac mora biti zadovoljan sa sustavom u cjelini: funkcionalnošću, intuitivnošću korištenja, cijenom).
7. Razumijevanje modela (svaki član tima mora znati kako i zašto čisti razvoj programske potpore treba funkcionirati).

Agilne metode općenito su pogodne u situaciji kada je naručitelj/korisnik spreman usko surađivati s razvojnim timom te ne postoji kruti vanjski ili kompanijski regulatorni okvir koji se mora zadovoljiti. Zbog manjka formalnosti u cijelom procesu razvoja, izravna komunikacija između članova razvojnog tima je nezaobilazna te stoga timovi moraju biti sastavljeni od članova spremnih na visok stupanj povezanosti.

S druge strane, kada je u pitanju razvoj „velikog“ softvera ili pak razvoj programske potpore s posebnim zahtjevima (sigurnost, pouzdanost, odaziv u stvarnom vremenu), agilne metode ne mogu dati zadovoljavajuće rješenje te je potreban strukturirani razvojni proces s višom razinom formalnosti u komunikaciji. Također, agilne metoda nisu se pokazale dobrima kada je u pitanju razvoj programske potpore koja će se koristiti kroz dugi vremenski period. Zbog manjka formalne i opsežne dokumentacije, gotovo je nemoguće raditi na kasnijem održavanju i nadogradnji jednom kada je gotov proizvod isporučen.

## 6. Alati i okruženja za potporu razvoja programa

U procesu razvoja programske potpore inženjeri koriste posebne alate kako bi automatizirali dio aktivnosti i samim time ubrzali izradu, pribavili korisne informacije o proizvodu koji se razvija te olakšali kasnije održavanje proizvoda. To su tzv. **CASE-alati** (engl. *Computer-Aided Software Engineering Tools*) [3].



**Definicija 6.1.** CASE-alati su programski proizvodi koji podupiru proces programskog inženjerstva, a posebice aktivnosti specifikacije, oblikovanja, implementacije i evolucije.

CASE-alati koriste se u svakoj fazi životnog ciklusa programskog proizvoda. Glavne prednosti korištenja ovih alata su (1) povećanje kvalitete konačnog proizvoda kroz normiranje načina prikaza i dijeljenja informacija te (2) smanjenje vremena i napora potrebnog za izradu programske potpore, što se postiže automatizacijom dijela procesnih aktivnosti (npr. izrada i organizacija dokumentacije) te povećanjem ponovne iskoristivosti (engl. *reusability*) modela i komponenti.

Automatizacija je iznimno važna značajka CASE-alata. CASE podupire automatizaciju oblikovanja raznim alatima kao što su:

- grafički uređivači za razvoj modela sustava,
- rječnici i zbirke za upravljanje entitetima u oblikovanju,
- okruženja za oblikovanje i konstrukciju korisničkih sučelja,
- alati za pronalaženje pogrešaka u programu,
- automatizirani prevoditelji koji generiraju nove inačice programa itd.

S druge strane, napredni CASE-alati mogu biti složeni do te mjere da od korisnika zahtijevaju ulaganje značajnog napora u savladavanje korištenja samog alata, a ni cijena samih alata (većina naprednih alata je komercijalna) nije zanemariva. Također, nastojanja da se kroz alat ostvari što veće normiranje i stupanj automatizacije su često u sukobu s potrebom za kreativnosti i pronalaženjem inovativnih rješenja koje programsko inženjerstvo zahtijeva. Zbog toga, unatoč činjenici da je CASE-tehnologija dovela do značajnog unapređenja procesa oblikovanja programske potpore, postignuta poboljšanja nisu sukladna očekivanjima (tj. poboljšanje učinkovitosti za red veličine). Nadalje, programsko inženjerstvo je timska aktivnost te se u svim većim projektima mnogo vremena utroši na interakcije unutar tima, no CASE-tehnologija slabo podupire timski rad.

### 6.1. Klasifikacija CASE-alata

Kako bi odabrali odgovarajući tip CASE-alata potrebno je razumijevanje različitih tipova CASE-alata i potpore koju pružaju aktivnostima u procesu programskog inženjerstva što

omogućuje njihova klasifikacija. Pri klasifikaciji CASE-alata koriste se tri različite perspektive:

1. Funkcionalna perspektiva – alati se klasificiraju prema specifičnoj funkciji koju obavljaju.
2. Procesna perspektiva – alati se klasificiraju prema aktivnostima koje podupiru u procesu.
3. Integracijska perspektiva – alati se klasificiraju prema njihovoj organizaciji u integrirane cjeline.

#### 6.1.1. Funkcionalna perspektiva

S obzirom na zadaću koju obavljaju CASE-alati, postoje različite vrste/tipovi alata, tablica 6.1.

**Tablica 6.1.** Tipovi CASE-alata prema funkcionalnoj perspektivi.

Tip alata	Primjer
Alati za planiranje (engl. <i>planning tools</i> )	PERT alati, tablični kalkulatori (npr. Excel)
Alati za uređivanje (engl. <i>editing tools</i> )	Uređivači teksta, dijagrama (npr. Notepad, Word, Visio)
Alati za upravljanje promjenama (engl. <i>change management tools</i> )	Sustavi za praćenje promjena u zahtjevima i proizvodima (npr. IBM Rational DOORS, Borland Caliber)
Alati za upravljanje konfiguracijom (engl. <i>configuration management tools</i> )	Sustavi za kontrolu i upravljanje verzijama (npr. Subversion, Git)
Alati za izradu prototipa (engl. <i>prototyping tools</i> )	Jezici vrlo visoke razine apstrakcije (npr. UML)
Alati za potporu metodama (engl. <i>method-support tools</i> )	Različiti podatkovni rječnici, generatori koda
Alati za jezično procesiranje (engl. <i>language-processing tools</i> )	Kompajleri, interpreteri
Alati za programsku analizu (engl. <i>program analysis tools</i> )	Različiti alati za analizu statičkih i dinamičkih performansi programa
Alati za ispitivanje (engl. <i>testing tools</i> )	Generatori ispitnih skupova
Alati za ispravljanje pogrešaka (engl. <i>debugging tools</i> )	Ugrađeni u razvojne okoline (npr. Visual Studio, Eclipse...)
Alati za izradu dokumentacije (engl. <i>documentation tools</i> )	Različiti alati za prijelom i uređivanje slika
Alati za reinženjering (engl. <i>re-engineering tools</i> )	Posebni alati za unakrsnu usporedbu dijelova sustava i restrukturiranje programa

### 6.1.2. Procesna perspektiva

S obzirom na generičke aktivnosti procesa programskog inženjerstva koje podupire određena skupina alata može se napraviti shema prema slici 6.1.

Alati za reinženjering			✓	
Alati za ispitivanje			✓	✓
Alati za ispravljanje pogrešaka			✓	✓
Alati za programsku analizu			✓	✓
Alati za jezično procesiranje		✓	✓	
Alati za potporu metodama	✓	✓		
Alati za izradu prototipa	✓			✓
Alati za upravljanje konfiguracijom		✓	✓	
Alati za upravljanje promjenama	✓	✓	✓	✓
Alati za izradu dokumentacije	✓	✓	✓	✓
Alati za uređivanje	✓	✓	✓	✓
Alati za planiranje	✓	✓	✓	✓
	Specifikacija	Oblikovanje	Implementacija	Validacija i verifikacija

Slika 6.1. Podjela CASE-alata prema procesnoj perspektivi.

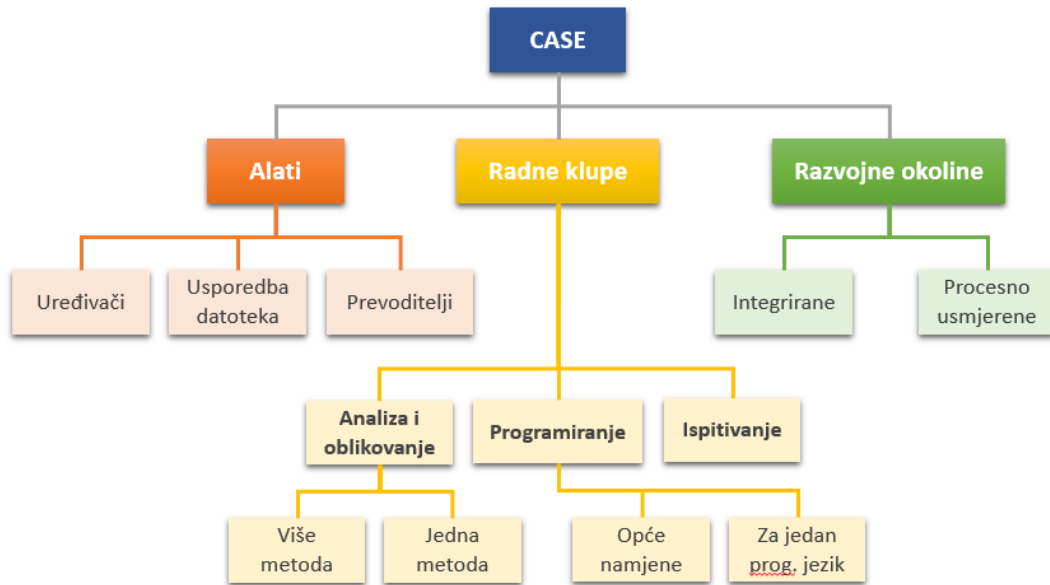
### 6.1.3. Integracijska perspektiva

Ova perspektiva promatra alate s obzirom na stupanj integracije alata u cjelinu.

Alati (u užem smislu) podupiru individualne zadatke u procesu (npr. oblikovanje, provjeru konzistencije zahtjeva, uređivanje teksta, ...). Često se spominju dvije kategorije alata:

- *Upper-CASE* alati (*front-end*), koji se koriste u raniji fazama kao što su izlučivanje zahtjeva, analize i modeliranja
- *Lower-CASE* alati (*back-end*), koji se koriste u kasnijim fazama implementacije, ispitivanja i održavanja.

Radne klupe (engl. *workbenches*) podupiru pojedine aktivnosti (faze) procesa (npr. specifikaciju). One objedinjavaju više različitih alata za potporu u nekoj fazi procesa programskog inženjerstva. Najčešće podržavaju jednu od tri aktivnosti: analizu i oblikovanje, programiranje te ispitivanje.



**Slika 6.2.** Integracijska klasifikacija CASE-alata: alati, radne klupe i razvojne okoline.

Razvojne okoline (engl. *environments*) podupiru cijeli ili značajan dio procesa programskog inženjerstva. Uključuju nekoliko integriranih radnih klupa.

Na slici 6.2 prikazan je dijagram integracijske klasifikacije.

U okviru kolegija Oblikovanje programske potpore koristit će se sljedeći CASE-alati i radne klupe:

- Subversion – upravljanje konfiguracijama programske potpore.
- Astah, Community Edition – oblikovanje zahtjeva, oblikovanje modela objektno usmjerene arhitekture.
- Microsoft Visio - oblikovanje zahtjeva, oblikovanje modela objektno usmjerene arhitekture.
- Microsoft Visual Studio – implementacija programske potpore vezana uz Microsoftove tehnologije
- Eclipse – implementacija programske potpore vezana uz Java tehnologije (i druge).

## 6.2. Sustavi za kontrolu inačica programske potpore

Prilikom razvoja programske potpore nezaobilazna je međusobna suradnja većeg broja osoba/timova koji se često nalaze na geografski razdvojenim lokacijama, no rade na istim datotekama. Stoga je kontrola inačica datoteka i vođenje evidencije o promjenama koje su nastale neizostavno.





**Definicija 6.2.** U programskom inženjerstvu, kontrola inačica programske potpore je svaki postupak koji prati i omogućava upravljanje promjenama nastalima u datotekama s izvornim kodom ili dokumentacijom.

U postupku kontrole inačica programske potpore koristi se nekoliko specifičnih pojmova. Revizija je osnovni pojam kojim se opisuje slijed razvoja. Jedinstveni slijed razvoja u kojem nema grananja naziva se osnovna razvojna linija (engl. *trunk*). Ako postoji potreba za razvojem dodatnih značajki programske potpore mimo osnovne linije, tada dolazi do razdvajanja razvoja u dvije ili više grana pri čemu svaku granu još nazivamo pomoćnom razvojnom linijom. (engl. *branch*).

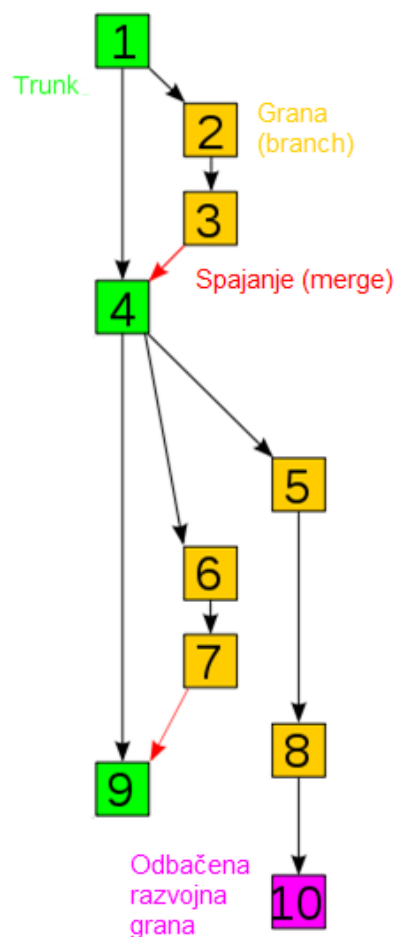
U najjednostavnijem slučaju, kad nema grananja, svaka revizija temelji se isključivo na jednoj jedinici reviziji koja joj neposredno prethodi te sve revizije čine jednu liniju. U takvom nizu postoji jedinstvena zadnja verzija koja se često naziva vršna revizija (engl. *head*). Ukoliko postoji grananje, iz jedne revizije može nastati nekoliko novih, a također je moguće da se nova revizija ne temelji na neposrednoj prethodnici nego na nekoj ranijoj reviziji. U takvom slučaju graf revizija umjesto linijskog poprima stablasti oblik te se vršna revizija mora eksplicitno zadati. Proces koji dovodi do spajanja revizija iz dviju ili više razvojnih linija u jedinstvenu reviziju naziva se spajanje linija (engl. *merge*). U praksi je ovaj proces iznimno teško izvesti i predstavlja jedan od najsloženijih aspekata kontrole inačica. Na slici 6.3 dan je primjer grafa kontrole inačica s osnovnom linijom i grananjima.

U tablici 6.2 sistematizirani su najčešći pojmovi korišteni u sustavima za kontrolu inačica programske potpore.

### 6.2.1. Vrste sustava za kontrolu inačica programske potpore

Danas postoji mnogo različitih sustava za kontrolu inačica programske potpore. Osim što se razlikuju po tipu licence – komercijalni i sustavi otvorenog koda, prije svega se razlikuju po lokalnosti pristupa te centraliziranosti, tj. postojanju središnjeg repozitorija.

Lokalizirani model (engl. *Local data model*) osnovni je tip sustava za kontrolu inačica. Pohrana i pristup podacima ograničeni su na jedno računalo na kojem su također sadržani



**Slika 6.3.** Graf kontrole inačica.

**Tablica 6.2.** Sažetak najčešćih pojmova korištenih u sustavima za kontrolu inačica.

Pojam	Definicija
<i>Branch</i> (pomoćna razvojna linija)	Skup datoteka obuhvaćen sustavom kontrole inačica koji se u određenom trenutku odvaja od osnovne linije razvoja i dalje razvija zasebno i neovisno o ostalim pomoćnim linijama razvoja.
<i>Checkout</i> (dohvaćanje)	Proces stvaranja lokalne radne kopije repozitorija. Korisnik može dohvatiti vršnu reviziju ili može odabrati bilo koju dostupnu reviziju. Ovaj pojam se ponekad koristi i kao imenica koja označava samu radnu kopiju repozitorija.
<i>Commit</i> (provedba)	Proces zapisivanja promjena iz radne verzije natrag u repozitorij. Koristi se još i izraz <i>check-in</i> .
<i>Conflict</i> (sukob)	Sukob koji nastaje kad dvoje ili više korisnika napravi promjene u istom dokumentu te sustav ne može automatski objediniti promjene novu verziju. Tada jedan od korisnika mora riješiti sukob ( <i>resolve</i> ) tako što će sam uklopiti različite promjene ili odbaciti sve osim jedne inačice nove verzije.
<i>Export</i> (izvoz)	Proces sličan <i>checkout</i> -u s razlikom u tome da se u strukturi direktorija ne nalaze i datoteke s meta-podacima potrebnim za kontrolu inačica.
<i>Head</i> vršna inačica	Najnovija inačica u svakoj grani. <i>Head</i> se upotrebljava za najnoviju glavnu inačicu.
<i>Import</i> (uvoz)	Proces uvoza lokalne strukture direktorija u središnji repozitorij po prvi put (nije radna kopija).
<i>Merge</i> (spajanje)	Spajanje dviju ili više inačica skupa datoteka u jedinstvenu inačicu. Događa se kod sljedećih radnji: <ul style="list-style-type: none"> <li>• osvježavanja lokalne radne verzije,</li> <li>• spajanja dviju grana u jednu,</li> <li>• razrješavanja sukoba,</li> <li>• ...</li> </ul>
<i>Repository</i> (središnji repozitorij)	Mjesto (na poslužitelju) na kojem se nalaze sve datoteke i popratni meta-podaci.
<i>Resolve</i> (razrješavanje)	Razrješavanje sukoba koji nastaje kada više korisnika istovremeno pokuša unijeti promjene u isti dokument.
<i>Trunk</i> (osnovna razvojna linija)	Osnovna ili glavna linija razvoja.
Update (osvježavanje)	Osvježavanje lokalne radne kopije s promjenama koje su nastale u središnjem repozitoriju.

podaci o ranijim promjenama. Promjene se prate za svaku datoteku zasebno, što znači da nije moguće odjednom raditi s cijelim skupom datoteka (npr. projektom). Kao dio GNU projekta razvijen je RCS sustav (Revision Control System) koji je kasnije poslužio kao okosnica za razvoj naprednijih sustava poput CVS-a i SVN-a, o čemu će više riječi biti u nastavku.

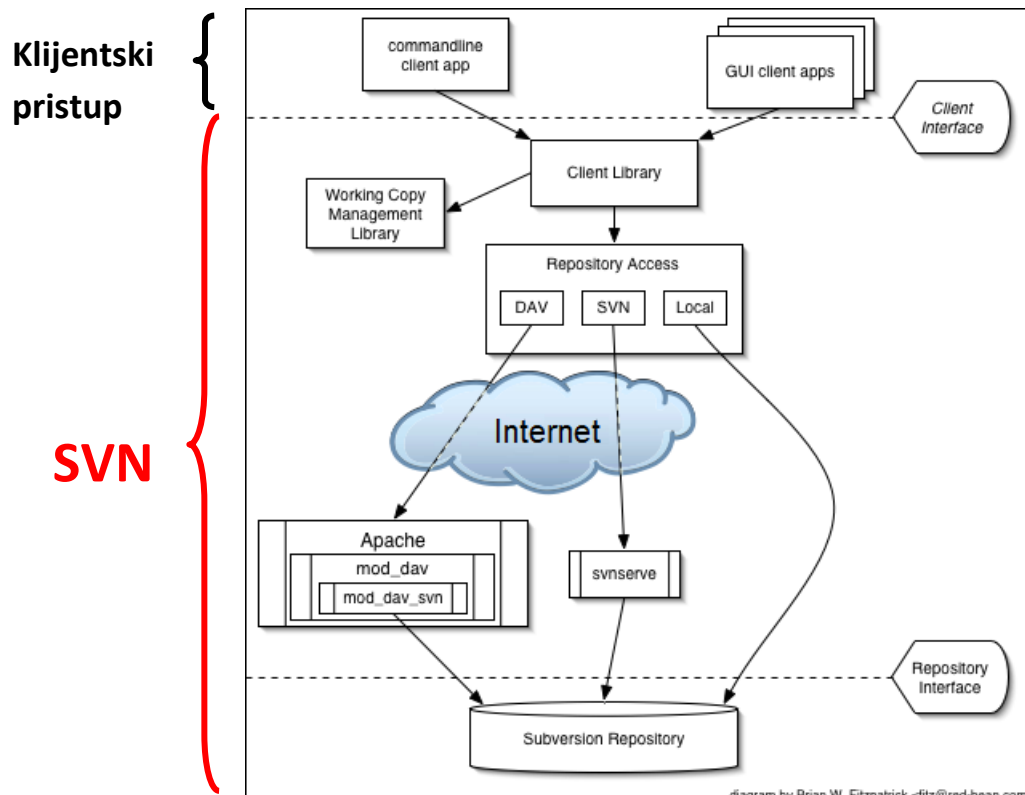
Model klijent-poslužitelj podrazumijeva postojanje središnjeg repozitorija na poslužiteljskom računalu čije radne inačice korisnici pohranjuju na svojim računalima i u

određenom trenutku sinkroniziraju sa središnjim repozitorijem. Jedan od prvih sustava otvorenog koda koji je implementirao ovaj model bio je CVS (Concurrent Versions System). Na temelju njega se kasnije razvio Apache SVN koji se i danas koristi.

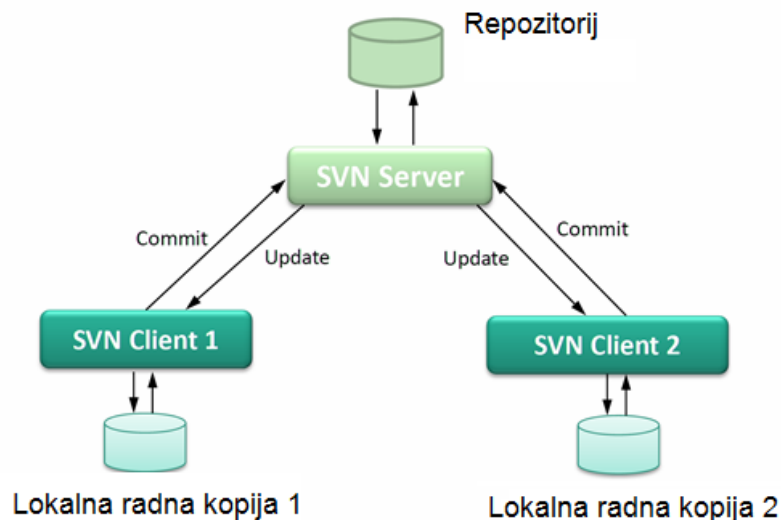
**Apache Subversion (SVN)** [10] je sustav za kontrolu inačica programske potpore distribuiran pod Apache licencom. Ovaj sustav nasljednik je ranijeg CVS sustava, a najznačajnija unaprjeđenja su u vidu potpune atomarnosti operacije *commit*, zaključavanja datoteka u slučaju da više korisnika pokuša istovremeno raditi izmjene, dohvaćanje *read-only* inačice središnjeg repozitorija i dr. Na slici 4 prikazana je arhitektura SVN-sustava, a dijagram na slici 6.4 prikazuje shemu rada sa SVN-sustavom.

Prilikom korištenja SVN-sustava, dvije najčešće korištene operacije su *Update* koja mijenja lokalnu radnu kopiju kako bi bila identična stanju središnjeg repozitorija te operacija *Commit* koja promjene nastale u lokalnoj radnoj kopiji izvozi u središnji repozitorij. Prilikom stvaranja lokalne radne kopije (prvo dohvaćanje sadržaja središnjeg repozitorija) koristi se operacija *Checkout*.

U slučaju sukoba inačica na kojima je istovremeno radilo više korisnika, sukob se može riješiti odbacivanjem vlastitih promjena – *Revert* ili se ide u razrješavanje spora – *Resolve* u kojem se odabire koja će se inačica zadržati (*base*, *mine*, *full*, *working*).



**Slika 6.4.** Arhitektura sustava *subversion* (SVN).



**Slika 6.5.** Rad s SVN-om.

Budući da Apache SVN podržava isključivo komandno-linijski način rada, razvijeni su posebni klijenti poput Tortoise SVN-a koji imaju integrirano grafičko korisničko sučelje.

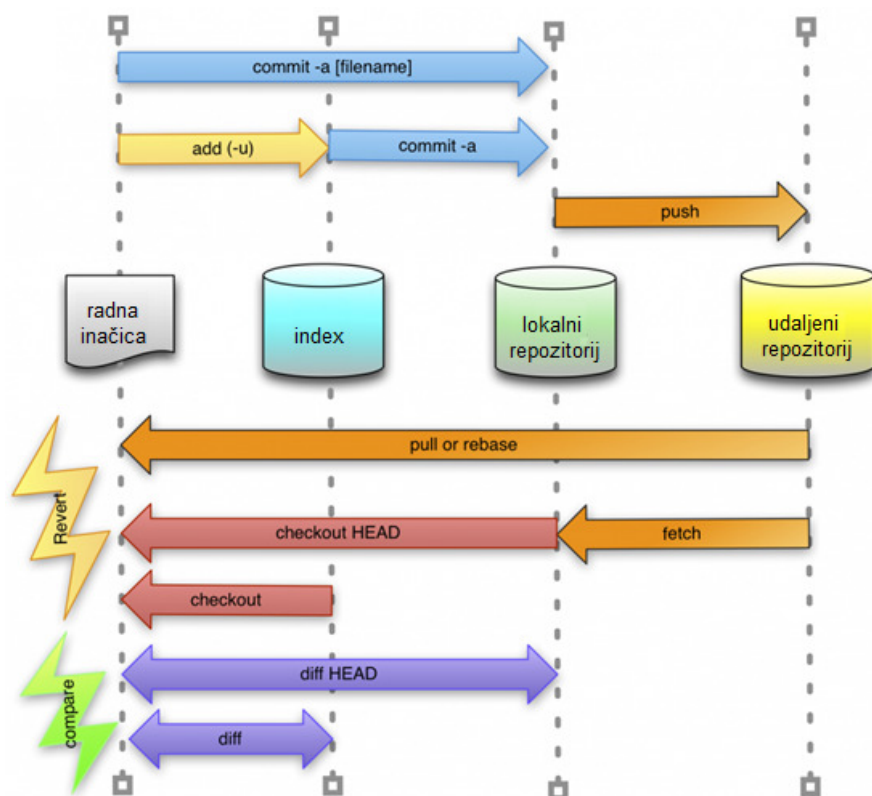
U raspodijeljenom modelu (engl. *distributed model*) ne postoji jedinstveni središnji repozitorij s kojim se klijenti sinkroniziraju, već svaka radna kopija repozitorija predstavlja repozitorij za sebe. U načelu, svi postojeći repozitoriji su ravnopravni, a korisnici mogu međusobno usklađivati repozitorije prema načelu *peer-to-peer*. Glavni predstavnici ovakvog modela su sustavi otvorenog koda Git i Mercurial.

**Git** [11] je danas vjerojatno najrašireniji sustav otvorenog koda za kontrolu inačica programske potpore. U odnosu na SVN znatno je složeniji za savladavanje i korištenje, no zato nudi dodatne funkcionalnosti, a njegova distribuirana arhitektura omogućava istovremeno postojanje više smjerova razvoja programske potpore koji se mogu, ali i ne moraju objediniti.

Arhitektura sustava Git podrazumijeva rad s dva repozitorija – lokalnim i udaljenim. Rad s lokalnim repozitorijem nalikuje na rad s centralnim repozitorijem u SVN-u: operacija *Commit* zapisuje promjene u lokalni repozitorij, a operacija *Checkout* osvježava radnu kopiju lokalnog direktorija (operacija *Update* pod tim imenom ne postoji). Lokalni repozitorij usklađuje se s udaljenim repozitorijem operacijama *Push* i *Fetch* (ukoliko se želi direktno osvježiti lokalna radna kopija umjesto *Fetch + Merge* može se direktno izvršiti operacija *Pull/Rebase*). Prilikom prvog dohvaćanja sadržaja udaljenog direktorija koristi se operacija *Clone*.

Dodatna mogućnost koju nudi Git je *Index* – lokalni kvazi-repozitorij u koji se pohranjuju nove i izmijenjene datoteke prije sinkronizacije s pravim lokalnim repozitorijem (operacije *Add* i *Checkout*).

Sve navedene operacije u sustavu Git shematski su prikazane na slici 6.6.



**Slika 1.6.** Sustav Git – struktura operacija.

## 7. Literatura

- [1] I. Sommerville, Software Engineering, 8th ed., Addison-Wesley, Harlow, England, 2007.
- [2] ACM/IEEE Code of Ethics v5.2, [www.acm.org/about/se-code](http://www.acm.org/about/se-code), pristupljeno 09/2014.
- [3] I. Sommerville, Software Engineering, 9th ed., Addison-Wesley, Harlow, England, 2011.
- [4] IEEE, IEEE Recommended Practice for Software Requirements Specifications. IEEE Software Engineering Standards Collection, Los Alamitos, CA, USA: IEEE Computer Society Press, 1998.
- [5] R. Pressman, Software Engineering: A Practitioner's Approach (7th Edition), McGraw-Hill Science/Engineering/Math, 2009., ISBN-10: 0073375977, ISBN-13: 978-0073375977
- [6] T. C. Lethbridge and R. Laganière, Object-Oriented Software Engineering: Practical Software Development Using UML and Java, 2nd ed., McGraw-Hill Publishing Company, London, 2004.
- [7] Manifesto for Agile Software Development, <http://agilemanifesto.org/>, pristupljeno 09/2014.
- [8] R. C. Martin, Agile Software Development: Principles, Patterns and Practices, Prentice Hall, Upper Saddle River, NJ, 2002.
- [9] GNU Project, RCS, <https://www.gnu.org/software/rcs/rcs.html>, pristupljeno 09/2014.
- [10] Apache, Apache Subversion, <http://subversion.apache.org/>, pristupljeno 09/2014.
- [11] Git, <http://git-scm.com/>, pristupljeno 09/2014.