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

PROCESI PROGRAMSKOG INŽENJERSTVA

E-skripta namijenjena studentima kolegija Oblikovanje programske potpore

Autori:

doc. dr. sc. Alan Jović, dipl. ing.

Nikolina Frid, mag. ing. comp.

dr. sc. Danko Ivošević, dipl. ing.

3. izdanje, Zagreb, rujan 2019.

Sadržaj

Predgovor

Skripta „Procesi programskog inženjerstva“ prvenstveno je namijenjena studentima preddiplomskog studija računarstva na Sveučilištu u Zagrebu Fakultetu elektrotehnike i računarstva, na kolegiju Oblikovanju programske potpore (OPP). Cilj skripte je predočiti programsko inženjerstvo s teorijskog stajališta, što je bitan preduvjet razumijevanja njegovih praktičnih aspekata te je u tom smislu skripta nadopuna gradivu prezentiranom na predavanjima i auditornim vježbama iz kolegija OPP. Treće izdanje skripte doživjelo je velike dorade i nadopune u odnosu na drugo izdanje iz 2015. godine. Prvenstveno, značajno je povećan obujam materijala u smislu toga da skripta sada pokriva gradivo čitavog kolegija, osim dijelova posvećenih modeliranju korištenjem jezika UML te matematičke logike i formalne verifikacije. Studente kolegija OPP upućuje se na sveučilišni priručnik "UML-dijagrami: Zbirka primjera i riješenih zadataka" za proučavanje modeliranja stvarnih programskih sustava korištenjem jezika UML kao i posjećivanje predavanja/auditornih vježbi na kojima se rješavaju zadatci korištenjem UML-a. Matematička logika i formalna verifikacija pojašnjeni su na predavanjima u dovoljnoj mjeri, a studenti će se detaljnije s logikom i formalnom verifikacijom upoznati na drugim kolegijima studija (npr. Umjetna inteligencija i Formalne metode u oblikovanju sustava).

Specifične nadopune trećeg izdanja u odnosu na ranije izdanje sastoje se u:

- novom poglavlju (sada poglavlje 6) posvećenom arhitekturi programske potpore
- novom poglavlju (sada poglavlje 7) posvećenom ispitivanju programske potpore
- nadopuni poglavlja o sustavima za kontrolu inačica programske potpore
- nadopuni poglavlja o modelima programskog inženjerstva vezano uz agilni pristup razvoju programske potpore.

Skripta i u trećem izdanju slijedi dobru praksu naglašavanja pojmova u tekstu pomoću zadebljanja (najbitniji pojmovi) i potcrtavanja (manje bitni pojmovi ili dijelovi rečenica), kao i uokviravanja definicija, bitnih pojašnjenja i danih primjera u zaobljenim pravokutnicima, čime se povećava čitljivost i olakšava učenje.

Konačno, potrebno je istaknuti da u slučaju manjih neslaganja u vidu iznesenog gradiva u ovoj skripti i na predavanjima prednost treba uvijek dati predavanjima, jer je živa riječ nastavnika aktualna i rezultat višegodišnjeg profesionalnog iskustva u radu s programskom potporom. Nastavnicima će biti drago ako im studenti napomenu potencijalna neslaganja te će nastojati uvažiti studentske sugestije.

1. Uvod

Procesi programskog inženjerstva obuhvaćaju razvoj programske potpore od 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 sloboda u osmišljavanju i ostvarivanju konačnog programskog proizvoda. Pritom najveću ulogu igra iskustvo i način razmišljanja inženjera. Razvoj programske potpore nesumnjivo je zahtjevna intelektualna aktivnost koja doprinosi razvoju modernog društva. Međutim, intelektualna aktivnost sama po sebi nije dovoljna da bismo 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 programskih proizvoda srednje težine 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 bilo potrebno razmišljati o sklopovskim ograničenjima, programskom jeziku prikladnom za rješavanje problema, nedefiniranim protokolima ispitivanja i nerazvijenim alatima za pomoć pri razvoju programa. Moderni razvoj programske potpore je značajno pojednostavio čitav postupak razvoja, no postavio je i nove, visoke kriterije kvalitete programskog proizvoda. Od moderne programske potpore se očekuje da radi bez pogrešaka, koristi resurse učinkovito, omogućava nadogradnju te ima intuitivno korisničko sučelje.

Pri razvoju suvremene programske potpore, nameću se dva vrlo važna pojma bez kojih danas nije moguće zamisliti kvalitetan programski proizvod, a to su apstrakcija i struktura. Apstrakcija znači da se razvoju programske potpore pristupa u vidu modela na različitim razinama složenosti, pri čemu svaka viša razina skriva (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 korištenja različitih, najčešće grafičkih jezika za prikaz modela sustava. Također, programski proizvod oblikuje se u manjim cjelinama – komponentama, pogodnima za ponovnu uporabu. Struktura ili strukturiranje znači da se izradi programske potpore pristupa sustavno, pri čemu se svaki korak razvoja, od definiranja zahtjeva, preko specifikacije arhitekture sve do pisanja programskog koda i razvoja novih inačica dokumentira na odgovarajući, inženjerski, strukturirani način. U oblikovanje se uvodi analiza i izbor stila arhitekture programske potpore te pripadni modeli (najčešće u obliku dijagrama) pogodni za formalnu analizu. Svi ovi suvremeni postupci dovode do povećane pouzdanosti i ponovne iskoristivosti programske potpore kao i do povećanja produktivnosti inženjerskog tima.

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.
4. Uz tradicijsko ispitivanje programske potpore uvode se formalne metode provjere, poglavito formalna verifikacija modela.

U poglavlju 2 objašnjeni su temeljni pojmovi vezani uz programsko inženjerstvo i programsku potporu. Poglavlje 3 obrađuje tematiku vezanu uz životni ciklus programske potpore: od specifikacije, preko oblikovanja i implementacije pa sve do validacije, verifikacije i evolucije. Inženjerstvo zahtjeva, usko vezano uz specifikaciju programske potpore, obrađeno je u poglavlju 4. U poglavlju 5 detaljno su obrađeni tradicionalni i suvremeni modeli procesa programskog inženjerstva. Arhitektura programske potpore, arhitekturni stilovi te načela dobrog oblikovanja programske potpore tema su poglavlja 6. Poglavlje 7 bavi se ispitivanjem programske potpore te donosi pregled faza i razina ispitivanja te načina organizacije i provedbe ispitivanja. U poglavlju 8 dan je kratki pregled računalnih alata i okruženja za potporu razvoja programske potpore.

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 programa.

Prije nego što se objasni što se podrazumijeva pod programskim inženjerstvom i koji je njegov značaj i prije nego što se definira sam pojam programskog inženjerstva, 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, skripte, knjižnice i druge podkomponente.

Programska potpore 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. *software product*) ili **programski sustav** (engl. *software system*) sastoji se od jednog ili više računalnih programa. On se može izraditi oblikovanjem novih računalnih programa, (re)konfiguracijom postojećih programa, ili ponovnom uporabom postojećih komponenata programa. Programski proizvod je uvijek vezan uz tržište i označava onu programsku potporu koja se prodaje pod nekom licencom.

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 efikasnost.

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 objašanjava 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.

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. 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 usmjereno na praksu i izradu programske potpore kojom će klijenti unaprijediti poslovni proces ili ostvariti dodatnu vrijednost. Razlika između ta dva pojma zorno je predočena na slici 2.1.

Slika 2.1 Područja programskog inženjerstva i računarske znanosti.

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. 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 (uključuje 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 rukovanje podacima u računalnom sustavu.

Inženjeri računalnih sustava su često uključeni u specificiranje sustava, oblikovanje arhitekture, integraciju sustava i postavljanje u korisničko okruženje. Kolegij Oblikovanje programske potpore prvenstveno je usredotočen na programsko inženjerstvo, dok se ostala potpodručja računalnog inženjerstva dotiču tek mjestimice.

2.2. Temeljna pitanja programskog inženjerstva

Neka od temeljnih pitanja kojma se programsko inženjerstvo bavi su sljedeća:

- Što je to i kako izgleda proces programskog inženjerstva?
- Što je to i koji su sve modeli procesa programskog inženjerstva?
- Što su to metode programskog inženjerstva?
- Što je to CASE (engl. *computer-aided software engineering*)?
- Koje su značajke dobrog programskog proizvoda?
- Kakva je struktura cijene (troška) u programskom inženjerstvu?
- Koje su osnovne poteškoće i izazovi u programskom inženjerstvu?

- Koje vrste programskih projekata postoje i koje su njihove značajke?
- Što je to profesionalna i etička odgovornost u programskom inženjerstvu?

Ovaj popis nije iscrpan, ali pokriva većinu pitanja koja će se obrađivati u nastavku.

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**:

- specifikacija (engl. *specification*)
- oblikovanje (engl. *design*) i implementacija (engl. *implementation*)
- validacija (engl. *valdiation*) i verifikacija (engl. *verification*)
- evolucija programske potpore.

Svaka generička 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)
- notaciju (označavanje) 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)
- smjernice pri opisu procesa (npr. najprije dokumentirati javne metode razreda, potom atribute).

2.2.4. CASE

Definicija 2.6. CASE je područje programskih proizvoda ili alata veće ili manje složenosti namijenjenih 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-proizvode (engl. *lower CASE*), koji podupiru kasnije aktivnosti kao što su kodiranje i ispitivanje. Detaljnije se CASE-proizvodi obrađuju u 8. 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 – programski proizvod mora biti razumljiv, koristan i kompatibilan s ostalim sustavima na strani korisnika.
- pouzdanost – korisnik mora vjerovati da sustav ispravno funkcionira. Onaj sustav koji jednom zakaže značajno smanjuje povjerenje korisnika što se treba izbjeći pod svaku cijenu.
- mogućnost laganog održavanja – uključuje naknadno ispravljanje pogrešaka i evoluciju proizvoda sukladno izmijenjenim i proširenim zahtjevima korisnika.

Naravno, svaki **dionik** (engl. *stakeholder*) programskog proizvoda ima svoja očekivanja i poglede na to što znači dobar proizvod. Dionik programske potpore je fizička ili pravna osoba na koju utječe ili bi mogla utjecati programska potpora koja se razvija te koja je zainteresirana za sudjelovanje u odlučivanju i/ili provođenju odluka vezanih uz proces razvoja; koja je u širem smislu dio razvojnog procesa. Tako je za kupca (klijenta) 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 programske potpore

Trošak programske potpore je aspekt koji određuje je li njezin razvoj isplativ ili ne. U pravilu, programska potpora se razvija tako da korist, tj. dodana vrijednost koju pruža korisniku/naručitelju, 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 konačni proizvod mora biti konkurentan i po pitanju cijene i po pitanju brzine izlaska na tržište, uz zadržavanje performansi i pouzdanosti.

Za programsku potporu koja ima 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 programske potpore, postoje česta **proturječja**. Primjerice, povećanje učinkovitosti specijalizacijom čini programsku potporu manje razumljivom 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, što znači da mora obavljati funkcionalnosti za koje je predviđen pouzdano i unutar zadanih ograničenja. Pritom se rade stalni kompromisi i optimizacije ograničenih resursa. Dobra inženjerska praksa je izbjegavanje 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 oni najčešći, odnosno karakteristični za sve procese programskog inženjerstva. To su poglavito **heterogenost razvoja, ograničeno vrijeme isporuke, povećanje pouzdanosti, česte izmjene zahtjeva** i **složenost** razvijenog programskog sustava.

Heterogenost razvoja znači da postoje različite tehnike i metode razvoja programske potpore za različite platforme i okoline izvođenje. Odabir tehnike i metode prikladne 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 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. Kako bi se primjereno odgovorilo na promjene korisničkih zahtjeva tijekom razvojnog procesa, uvode se nove agilne metodologije razvoja koje omogućavaju jednostavniju prilagodbu razvojnog procesa na promjenu zahtjeva. Složenost programskog sustava ne mora biti problem ako je arhitektura dobro osmišljena i ako je 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, novog 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, neovlašteno upadanje u druge sustave i sl.) i slijeđenje smjernica etičkog kodeksa prema udrugama IEEE/ACM [2]. Od smjernica koje su tamo navedene, 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 začetka, 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 programske potpore. Različiti autori navode različiti broj i vrstu generičkih aktivnosti. Međutim, većina se slaže oko sljedećih aktivnosti, izloženih u [1]:

1. **specifikacije**
2. **oblikovanja i implementacije** (ili kraće: razvoja)
3. **validacije i verifikacije**
4. **evolucije programske potpore**.

U nastavku se opisuju detaljnije svaka od tih generičkih aktivnosti.

3.1. Specifikacija programske potpore

Zadatak generičke aktivnosti specifikacije programske potpore jest određivanje funkcionalnosti te ograničenja na sustav temeljem analize zahtjeva.

Definicija 3.1. Specifikacija programske potpore određuje se procesom inženjerstva zahtjeva (engl. *requirements engineering*), koje je zasebno potpodručje programskog inženjerstva.

Zbog svoje složenosti i značaja, inženjerstvo zahtjeva se detaljnije razrađuje u zasebnom, poglavlju 4 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 izradom dokumenta specifikacije programske potpore, u kojem se navode potrebne usluge i ograničenja u radu i razvoju programske potpore koju 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.

Izrada točne specifikacije 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

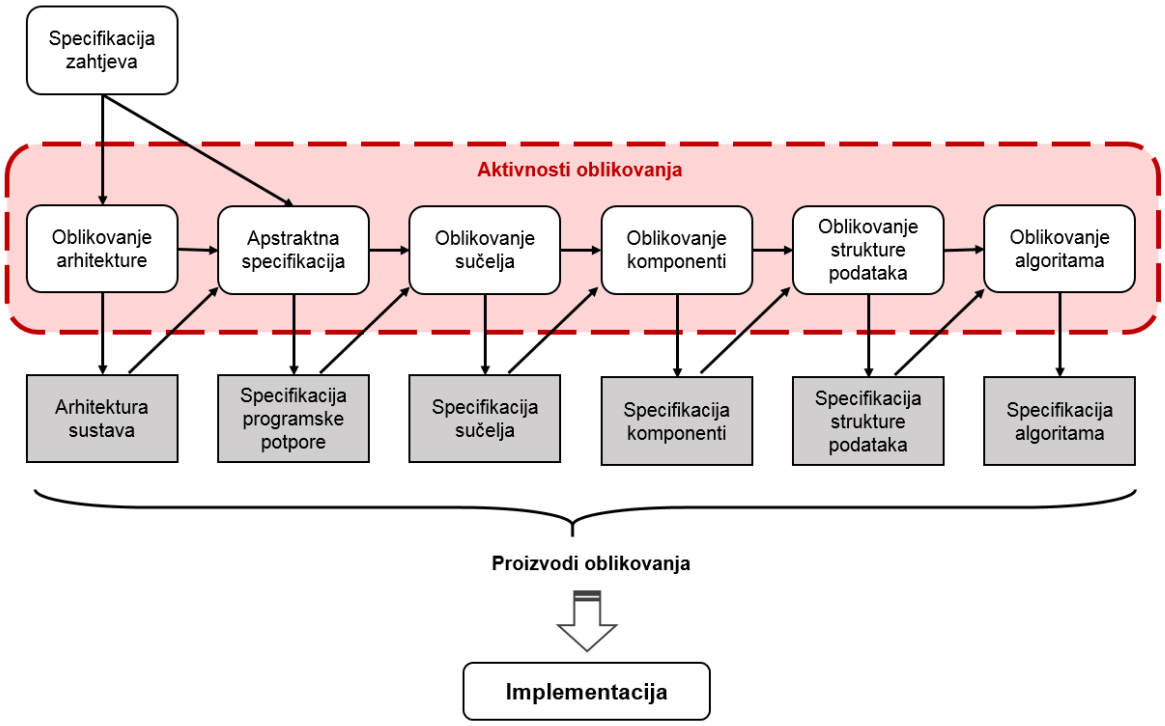
Oblikovanje i implementacija predstavljaju generičku aktivnost razvoja programske potpore u užem smislu. Aktivnost oblikovanja i implementacije na temelju specifikacije programske potpore izrađuje konačni proizvod koji čine izvorni i izvršni kod, podaci potrebni za konfiguraciju i rad te popratna dokumentacija. Ako se promatra detaljnije, ova aktivnost sastoji se od dvije međusobno isprepletene faze:

- **oblikovanje strukture sustava** ispravno modelira specifikaciju – provodi se izbor i modeliranje arhitekture – oblikovanje programske potpore u užem smislu. Rezultat oblikovanja je dokument specifikacije programske potpore (razlikovati od generičke aktivnosti specifikacije!) na temelju koje se kreće u implementaciju.
- **implementacija** – preslikava specifikaciju programske potpore dobivenu oblikovanjem u izvorni i 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 arhitekure, tijekom kojeg se utvrđuju i dokumentiraju podsustavi i njihove veze, dakle sve ono najvažnije što čini arhitekturu konačnog sustava. Također, u tom koraku odabire se stil arhitekure sustava (npr. objektno usmjereni, cjevovodi i filteri, arhitekura zasnovana na događajima, repozitorij podataka). Izbor arhitekture nije jednoznačan niti formalno propisan postupak, već se najčešće temelji na neformalnoj analizi i dobroj inženjerskoj praksi. Na temelju odabrane arhitekture slijedi sustavni pristup oblikovanju programskog proizvoda. Arhitektura sustava trebala bi biti dokumentirana skupom modela (najčešće prikazanih korištenjem dijagrama, npr. UML dijagrama razreda i komponenti).

Tijekom oblikovanja arhitekture pristupa se **apstraktnoj specifikaciji**, gdje se za svaki podsustav navodi ukratko usluge koje će pružati i ograničenja pod kojima on treba raditi. **Oblikovanje sučelja** definira za svaki podsustav njegovo sučelje prema ostalim podsustavima. Definicija sučelja mora biti jednoznačna i mora omogućiti 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 usmjerene 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, oblikovanje strukture podataka i algoritama, koji put se odgađaju do faze implementacije.



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

3.2.2. Implementacija programske potpore

Implementacija je preslikavanje dokumentiranog oblikovanja (odabrane arhitekture) u konačni proizvod. Programiranje je osobna aktivnost koja podrazumijeva naglašenu mentalnu aktivnost – ne postoji definirani generički proces programiranja.

Neki programeri počnu s onim komponentama koje bolje razumiju pa nastavljaju s onima koje slabije razumiju. Drugi postupaju obrnuto. Neki put je programiranje usmjereno 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 opći model procesa oblikovanja programske potpore na različit način ostvaruje ovisno o specifičnom modelu procesa programskog inženjerstva. Tako, primjerice, pri korištenju agilnih metoda, svi proizvodi nakon dokumentirane arhitekure 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 Unified Process (UP), 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 klijenata i krajnjih korisnika.

Definicija 3.2. Validacija provjerava izvršava li sustav sve funkcionalnosti na način kako je zadano u specifikaciji. Verifikacija ispituje ima li u sustavu prisutnih kvarova.

Drugim riječima, validacija odgovara na pitanje: "*Gradimo li pravi sustav?*", dok verifikacija odgovara na pitanje: "*Gradimo li sustav na ispravan način?*". Potrebno je validirati da je sustav koji razvijamo upravo onakav kakav je korisnik zamislio i zatražio od nas. Verifikacija se bavi ustanovljavanjem postoje li kvarovi u našem sustavu koji su uvedeni tijekom procesa razvoja.

I validacija i verifikacija provode se ispitivanjem 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 pronalaženje pogrešaka, a za provjeru odsustva pogrešaka mogu se koristiti formalne metode. 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.

Glavni cilj ispitivanja programske potpore je pronalaženje i otklanjanje pogrešaka. 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 zadanima. Bez unaprijed zadane specifikacije nema niti ispitivanja!

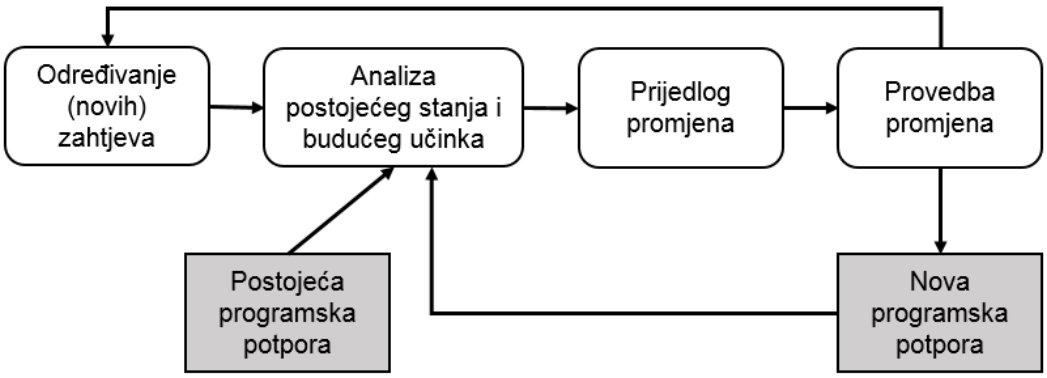
3.4. Evolucija programske potpore

Programska potpora je inherentno fleksibilna 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), programska potpora koja podupire poslovni proces u pravilu se mora mijenjati. Promjene programske potpore tijekom vremena oslabljuju strukturu njezine izvorne arhitekture što dovodi do smanjene učinkovitosti pri provođenju novih promjena. S vremenom, strukturu je potrebno ponovno oblikovati (engl. *refactoring*), uz moguće značajne izmjene dijelova koda, kako bi se zadržala kohezija i spriječilo daljnje rasipanje funkcionalnosti.

Povijesno promatrano, uvijek je postojala granica između razvoja programske potpore i njezine evolucije, koja je uglavnom podrazumijevala **održavanje sustava** (engl. *maintainance*). Inženjeri su 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 3.2 prikazan je opći shematski dijagram evolucije 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:

- dodavanja nove funkcionalnosti (engl. *system enhancement*)
- izmjena postojeće funkcionalnosti radi prilagodbe novonastalim okolnostima (engl. *platform adaptation*)
- ispravaka zbog uočenih kvarova programske potpore (engl. *fault repair*).



Slika 3.2 Evolucija programske potpore.

Prije provedbe promjena, nužno je sagledati trenutačnu funkcionalnost i moguće učinke 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. **Linija proizvoda** (engl. *product line*) je skup 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*", "*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 čini najčešće jedan, a koji put i više specijaliziranih radnih okvira, o čemu će više riječi biti u poglavlju 6.

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 se oblikuju u vidu kraćih ili dužih dokumenata koji specificiraju usluge sustava i ograničenja u uporabi.

4.1. Višedimenzijska klasifikacija zahtjeva

Pojam zahtjeva na programsku potporu 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 primjer 4.1 [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 kada se razine zahtjeva međusobno miješaju. Upravo stoga je vrlo bitno klasificirati zahtjeve prema razini detalja i prema sadržaju. Tako se prema **razini detalja** razlikuju:

- **korisnički zahtjevi** (engl. *user requirements*)

- **zahtjevi sustava** (engl. *system requirements*)
- **specifikacija programske potpore** (engl. *software specification*).

Korisnički zahtjevi su zahtjevi koji imaju najvišu razinu apstrakcije i najmanju količinu detalja. Njih najčešće zadaje korisnik i to uglavnom u neformalnom, nestrukturiranom obliku. Tako se oni pišu u prirodnom jeziku i crtaju jednostavnim dijagramima, najčešće UML dijagramima obrazaca uporabe. 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 proizvoda, kao u prethodnom 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 u primjeru 4.2. 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

Primjer 4.2.

Korisnički zahtjevi:

1. Programski sustav za ministarstvo zdravlja generirat će mjesečna izvješća za upravu u kojima će predočiti 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".

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.

Svaku razinu zahtjeva čitaju i dorađuju različiti dionici na programskom projektu. U tablici 4.1 prikazan je popis dionika koji utječu na zahtjeve, ovisno o razini detalja.

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 Stručnjaci iz domene primjene sustava 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

4.1.2. Podjela zahtjeva prema sadržaju

Prema **sadržaju**, zahtjeve se može podijeliti na:

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

- **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 zahjevi 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 programske potpore. Najgrublja podjela je na tri tipa:

- **zahtjevi programske potpore** – specificiraju na koji se osobit način treba ponašati isporučeni proizvod kao npr. vrijeme odziva, podržani sustav, komunikacijski protokol i sl.
- **organizacijski zahtjevi** – rezultat organizacijskih pravila i procedura kao npr. uporaba propisanog normiranog procesa razvoja, korištenje uvijek točno određenog programskog jezika pri razvoju.
- **vanjski zahtjevi** – izvan sustava i razvojnog procesa, npr. postizanje međusobne operabilnosti s drugim sustavima, zakonski zahtjevi i drugo.

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

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 pojavljuju kasnije u procesu otkrivanja zahtjeva buduće da ih je teško dobiti od stručnjaka. Naime, prilikom otkrivanja zahtjeva, postoje dvije proturječnosti: **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 i ne navode ih – implicitnost. Otkrivanje zahtjeva domene primjene u kasnim fazama razvoja, kada je sustav već gotovo dovršen, može značajno poskupiti 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 4.3 u kojem je opisan hipotetski programski sustav LIBSYS moguće je pojasniti probleme koji se javljaju prilikom navođenja zahtjeva.

Primjer 4.3. 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.

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 teško čitljivog dokumenta. Drugo što se može primijetiti jest 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 tumačeni 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 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.

Tablica 4.2 Mogućnosti specifikacije zahtjeva sustava.

Način zapisa	**Opis**
Rečenice prirodnog jezika	Zahtjevi su pisani korištenjem pobrojanih rečenica prirodnog jezika. Svaka rečenica trebala bi izreći samo jedan zahtjev sustava.
Strukturirani prirodni jezik	Zahtjevi su pisani u prirodnom jeziku u normiranom obliku ili na obrascu. Svako polje daje informaciju o jednom vidu zahtjeva.
Specifični jezik za opis oblikovanja	Ovaj pristup koristi jezik sličan programskom jeziku, ali s apstraktnijim značajkama za definiranje operativnog modela programskog sustava. Primjer jezika je SDL (engl. S*pecification and Description Language*). Danas se ovaj pristup rijetko koristi, osim za specifikaciju sučelja.
Grafička notacija	Grafički modeli koji su nadopunjeni tekstualnim opisom koriste se za definiciju funkcionalnih zahtjeva sustava. Pritom se najčešće koristi UML dijagrami obrazaca uporabe (engl. *use case diagram*) i UML sekvencijski dijagrami (engl. *sequence diagram*).
Matematička specifikacija	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.

4.1.5. Dokument zahtjeva

Dokument zahtjeva programske potpore je službena izjava o tome kakvu programsku potporu trebaju razvojni inženjeri ostvariti. On najčešće sadržava i neformalne

korisničke zahtjeve i detaljniju specifikaciju zahtjeva sustava. Kada je to moguće, ponekad se obje vrste zahtjeva integriraju u jedinstveni opis. U ostalim slučajevima, korisnički zahjtevi 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. Prema normi instituta IEEE iz 1998. [4] te doradi prijedloga iz [3], dokument zahtjeva trebao bi sadržavati sljedeće stavke:

1. Predgovor
2. Uvod
3. Rječnik pojmova
4. Definicija korisničkih zahtjeva
5. Specifikacija zahtjeva sustava
6. Arhitektura sustava
7. Modeli sustava
8. Evolucija sustava
9. Dodaci
10. Indeks (pojmova, 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

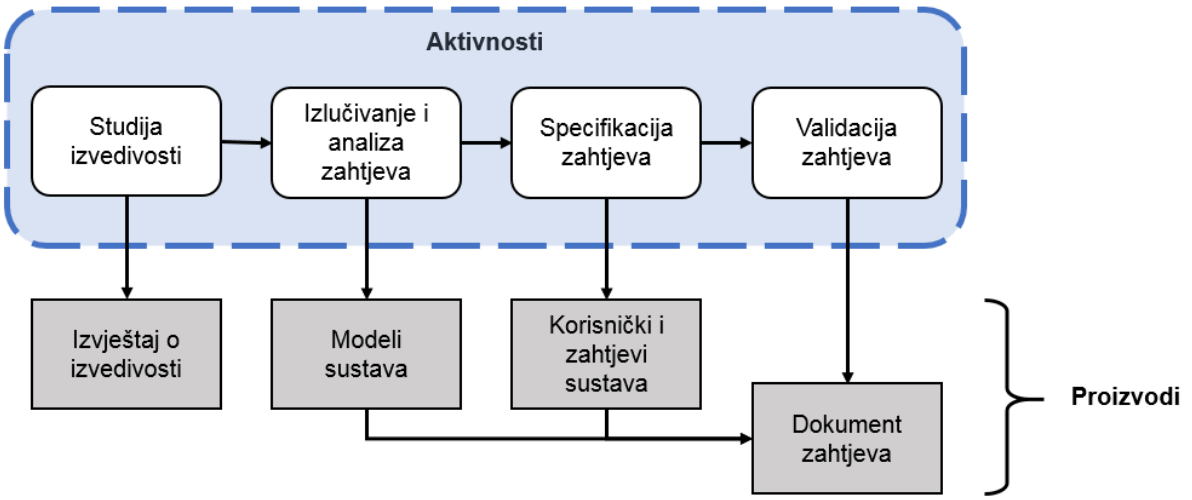
Procesi koji čine inženjerstvo zahtjeva programske potpore razlikuju se ovisno o domeni primjene, ljudskim resursima i organizaciji koja oblikuje zahtjeve. Pritom je važno naglasiti da nema jedinstvenog procesa inženjerstva zahtjeva koji bi bio primijenjiv neovisno o vrsti projekta. Ipak, u okviru svakog 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*)
- **upravljanje promjenama u zahtjevima** (engl. *requirements management*)

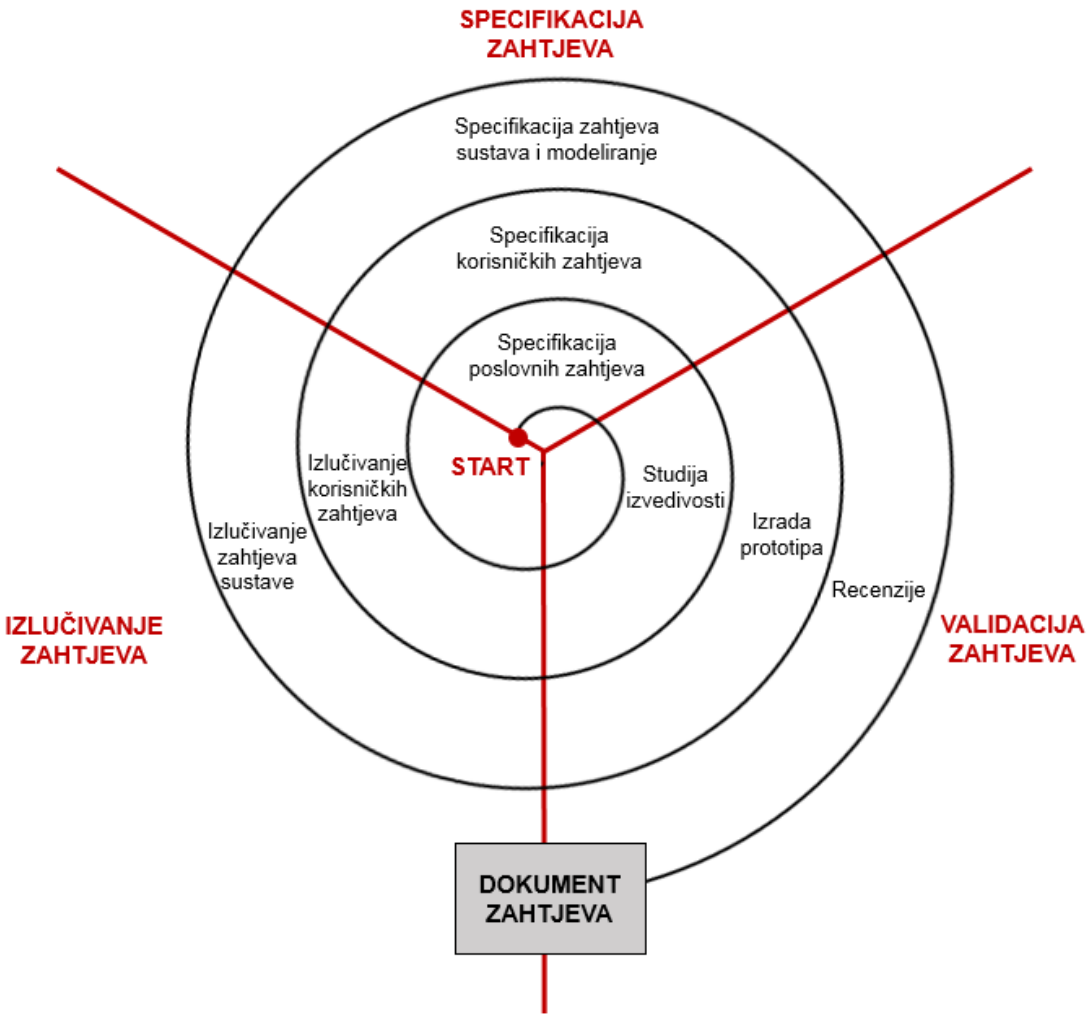
4.2.1. Generičke aktivnosti inženjerstva zahtjeva

Generičke aktivnosti inženjerstva zahtjeva ne odvijaju se nužno slijedno, već se mnogo češće isprepliću. Na slici 4.1 prikazan je **klasični model procesa inženjerstva zahtjeva** u kojem se konačni dokument zahtjeva postepeno slaže na temelju nekoliko manjih dokumenata - proizvoda pojedinih generičkih aktivnosti. Iteracije su prisutne između aktivnosti izlučivanja i analize zahtjeva, specifikacije zahtjeva i validacije zahtjeva, sve dok se zahtjevi ne usuglase.



Slika 4.1 Klasični model procesa inženjerstva zahtjeva.

Bolji model razvoja specifikacije programske potpore od klasičnog je **spiralni model,** prikazan na slici 4.2 , kod kojeg vrijeme i trud uloženi u svaku aktivnost ovise o stupnju razvoja i vrsti programskog proizvoda. 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 korisnički zahtjevi i izrađuju prototipovi sustava, da bi se u vanjskim slojevima spirale konačno prešlo na detaljno izlučivanje, analizu i specifikaciju svih zahtjeva sustava.



Slika 4.2 Spiralni model procesa inženjerstva zahtjeva, prilagođeno iz [3].

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 prelazi se na simultani razvoj specifikacije i implementacije, osim u prvoj iteraciji kada se radi samo specifikacija i modeliranje.

4.2.2. Studija izvedivosti

Studija izvedivosti je kratka, fokusirana studija na početku procesa inženjerstva zahtjeva kojom se utvrđuje isplati li se predloženi sustav financijski (tj. je li vrijedan uloženih sredstava). Ulazna informacija je preliminaran skup poslovnih zahtjeva procesa. Zadatak studije izvedivosti je odgovoriti 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?
- Može li se predloženi sustav integrirati s postojećim sustavima organizacije u koju se uvodi?

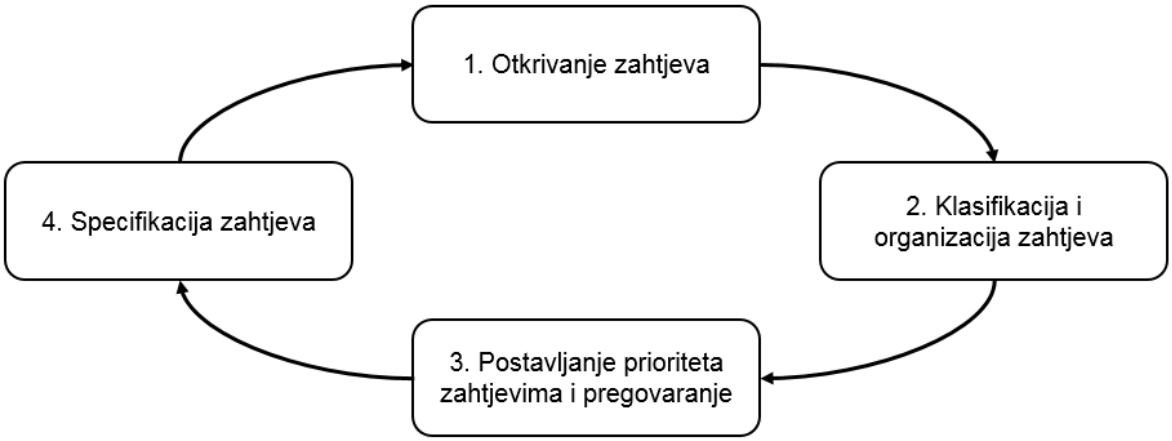
U slučaju da se utvrdi da je odgovor na barem jedno od tih pitanja negativan, nije poželjno nastaviti s daljnjim razvojem programskog sustava. Pri provođenju studije izvedivosti, najprije se određuje koje su sve informacije potrebne kako bi se studija završila, zatim se te informacije prikupljaju, i 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 trenutačni 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?
- Postoji li potreba za novom tehnologijom ili novom vještinom?
- 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 krajnjim 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, službenike na šalterima, administratore baza podataka, rukovoditelje sigurnosti, marketinški odjel, inženjere održavanja sustava (sklopovlja i programske potpore) i regulatorna tijela za bankarstvo.

Model aktivnosti izlučivanja i analize zahtjeva prikazan je na slici 4.3. Svaka organizacija imat će vlastiti način provedbe ovog modela, koji će ovisiti o



Slika 4.3 Model aktivnosti izlučivanja i analize zahtjeva.

specijalizaciji osoblja, vrsti programske potpore, 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. Koraci u iteracijama su:

1. **izlučivanje (otkrivanje) zahtjeva** – 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.
2. **klasifikacija i organizacija zahtjeva** – srodni zahtjevi se grupiraju i organiziraju u koherentne grozdove (engl. *clusters*). Zahtjevi se u praksi najčešće grupiraju prema podsustavu cijelog sustava koji opisuju. Iz tog razloga je teško u praksi razgraničiti postupak specifikacije zahtjeva od razvoja arhitekture sustava.
3. **postavljanje prioriteta i pregovaranje** – zahtjevi se razvrstavaju po prioritetima i razrješavaju se konflikti nastali zbog različitih pogleda različitih dionika na sustav. U praksi je cilj postići kompromis zahtjeva među dionicima.
4. **specifikacija zahtjeva** – zahtjevi se pisano dokumentiraju tehnikama navedenim u tablici 4.2 i zapisuju u formalnim ili neformalnim dokumentima, a najčešće grafičkom notacijom u vidu UML obrazaca uporabe i UML sekvencijskih dijagrama. Dokumentirani zahtjevi predstavljaju ulaz u sljedeću iteraciju.

Pri izlučivanju zahtjeva, različiti dionici ili grupe dionika imaju različite **poglede** (engl. *viewpoint*) tj. 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, tj. teško artikuliraju zahtjeve ili su zahtjevi nerealni s obzirom na izvedivost ili cijenu.
- Dionici izražavaju zahtjeve na različite, njima specifične načine vezane uz implicitno znanje o svojem radu – domeni.
- Različiti dionici mogu imati konfliktne zahtjeve 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 intervjua, klijenta se prethodno obavještava da će biti intervjuiran vezano uz zahtjeve budućeg programskog sustava, dok se kod

neformalnog intervjua takva obavijest ne zahtijeva. I u formalnom i u neformalnom intervjuiranju tim zadužen za inženjerstvo zahtjeva ispituje dionike o sustavu koji trenutačno koriste te o novopredloženom sustavu.

Prema tipu intervjua, razlikuju se **zatvoreni** i **otvoreni** intervjui. Kod zatvorenog tipa intervjua, klijent odgovara na skup već prije definiranih pitanja koja zanimaju razvojni tim. Kod otvorenog tipa intervjua 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 intervjua rijetko kad rezultira u preciznim informacijama, jer diskusije odu u 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 trenutačnim sustavom, budući da klijenti najčešće vole pričati o svojem poslu, a često se vole i žaliti na trenutačno 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 intervjua 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 stvarnih primjera nego kroz apstraktne opise. Upravo zato, **scenariji** su popularna i uspješna metoda izlučivanja zahtjeva.

Definicija 4.2. Scenariji su pažljivo osmišljeni primjeri o stvarnom načinu korištenja sustava.

Tijekom izlučivanja zahtjeva, dionici diskutiraju i kritiziraju scenarije. Svaki scenarij pokriva jedan ili nekoliko mogućih interakcija korisnika sa sustavom. Scenarij uglavnom 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 mogućih odstupanja od normalnog tijeka događaja
- informaciju o paralelnim aktivnostima
- opis stanja gdje scenarij završava.

Primjer 4.4. prikazuje jedan mogući scenarij za prikupljanje povijesti bolesti u sustavu MHC-PMS .

Primjer 4.4. Primjer scenarija za prikupljanje povijesti bolesti za sustav MHC-PMS, prilagođeno iz [3].

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).

Odstupanja:

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:

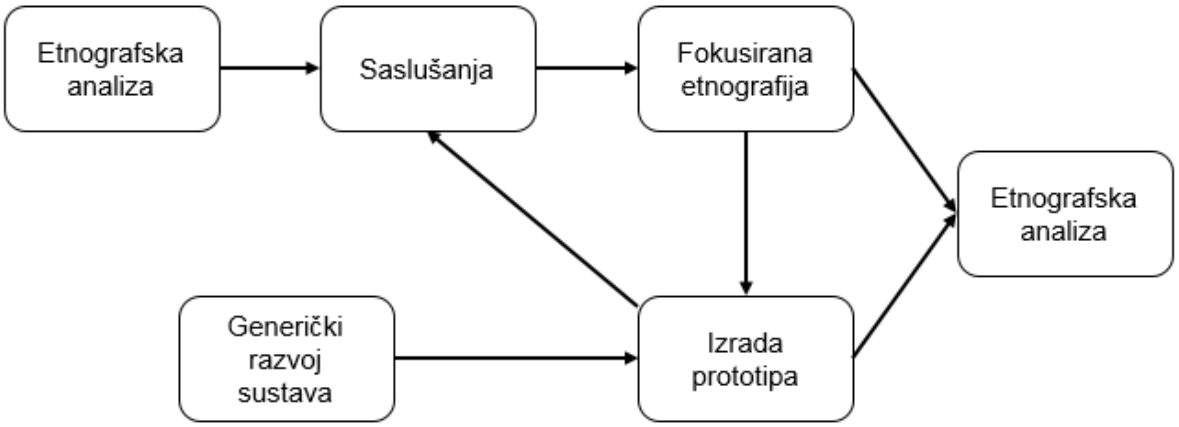
Sestra je prijavljena u sustav. Pacijentov zapis koji sadrži povijest bolesti je unešen u sustav. Dnevnik sustava pokazuje početak i kraj sjednice i ime i prezime sestre koja je provela upis podataka.

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. Stoga je, posebice u složenim sustavima, potrebno ostvariti bolji uvid u stvarne procese u tvrtkama za koje se razvija programska potpora, što se postiže tzv. **etnografijom**.

Definicija 4.3. 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. Tako se otkrivaju 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 čiji je tijek prikazan na slici 4.4. 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.



Slika 4.4 Fokusirana etnografija, 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 definiraju li zahtjevi koji su dobiveni od klijenta zaista sustav koji korisnik želi. Validacija se ispreplićе s analizom zahtjeva budući da se bavi pronalaskom pogrešaka u zahtjevima. Naknadno ispravljanje pogreške u zahtjevima može biti više 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.

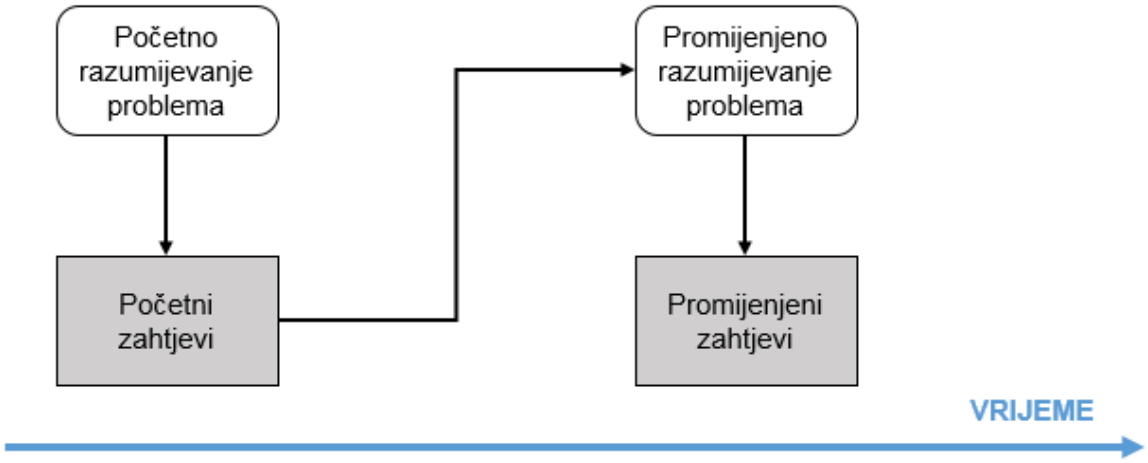
U zahtjevima se provjerava više svojstava. To uključuje:

- **valjanost** – sustav ostvaruje funkcionalnost koja podupire potrebe većine dionika
- **konzistentnost** – ne postoji konflikt u zahtjevima
- **kompletnost** – sustav uključuje sve funkcije koje je korisnik tražio
- **realnost** – sve funkcije se mogu implementirati uz danu tehnologiju i proračun
- **provjerljivost** – svi zahtjevi se mogu provjeriti
- **razumljivost** – dokument zahtjeva je jasno napisan
- **slijedivost** – naveden je izvor dokumenta u slučaju više povezanih dokumenata
- **prilagodljivost** – zahtjevi se mogu mijenjati bez utjecaja na druge zahtjeve.

Nije dobro potcijeniti probleme koji se tiču validacije zahtjeva, što se posebno odnosi na pokazivanje da neki skup zahtjeva točno odgovara klijentskim potrebama.

4.2.5. Rukovanje promjenama u zahtjevima

Rukovanje promjenama u zahtjevima je proces razumijevanja i upravljanja promjenama 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 4.5.



Slika 4.5 Evolucija zahtjeva, prilagođeno iz [1].

Jednom kada je sustav instaliran i u svakodnevnoj uporabi, često se pojavljuju novi zahtjevi. Razlog pojavi novih zahtjeva 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 određeni novi prioriteti.

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 na novo uvedenom sustavu.

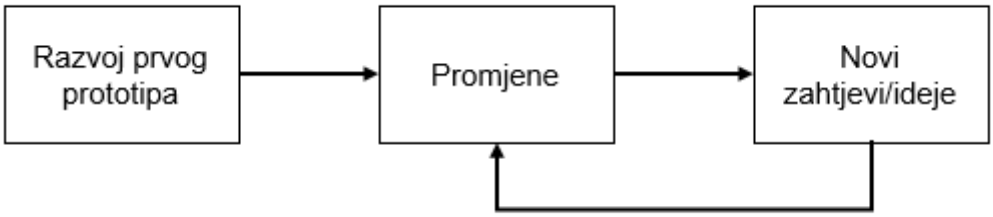
Pri rukovanju promjenama, potrebno je voditi računa o svim pojedinim zahtjevima i održavati veze između ovisnih zahtjeva tako da se može procijeniti značaj promjena. Zato se često na početku ustanovljava formalni proces za prijedlog uvođenja promjena i za povezivanjem promjena za zahtjevima sustava. 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 trenutačni 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 kasnija nemogućnost rekonstrukcije što je točno bilo izmijenjeno. Tu puno pomažu odgovarajući specijalizirani CASE-alati koji nastoje automatizirati proces praćenja promjena.

5. Procesi i modeli procesa programskog inženjerstva

Modeli procesa programskog inženjerstva predstavljaju dobre prakse pristupa izradi programske potpore. Oni su izvedeni na temelju iskustava razvojnih timova kao skup pozitivnih zaključaka o važnosti i redoslijedu izvođenja projektnih aktivnosti. Stoga ih ne treba shvaćati kao stroge ili manje stroge recepte kako se posao mora napraviti, već prije svega kao pomoć pri razmišljanju kako projekt iz programske potpore uspješno privesti kraju.

U pristupu koji bi zanemario saznanja na kojima se temelje modeli programskog inženjerstva, razvojni tim bi započeo izradu programske potpore te ju neprestano mijenjao sve do zadovoljenja korisničkih zahtjeva. Takav pristup naziva se *ad-hoc* ili **oportunistički pristup** koji do određene veličine razvijene programske potpore može biti zadovoljavajući. On se može prikazati kao niz aktivnosti ilustriran na slici 5.1.



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

Iskustvo oportunističkog razvoja na duže staze prepoznaje određene probleme i nedostatke:

- Nema razrade zahtjeva i oblikovanja 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.
- Dolazi do brže istrošenosti ili propadanja (engl. *deterioration*) programske potpore.
- S obzirom na neplanski razvoj, cilj nikad nije dovoljno jasan pa nema ni pravog uvida, radi se samo ono što je trenutačno potrebno, 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.

Zbog nedostataka *ad-hoc* pristupa, inženjeri su tijekom vremena razmatrali različite uređene postupke razvoju programske potpore. Tako su uočena tri jasno definirana i općeprihvaćena generička modela programskog inženjerstva koji nastoje nadići negativne strane *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*) – temeljne aktivnosti procesa izrade programske potpore smatraju se nezavisnim fazama razvoja
- **evolucijski model** (engl. *evolutionary model*) – sustav se razvija kroz niz inačica ili inkremenata od kod kojih svaki sljedeći inkrement dodaje neku novu funkcionalnost u onu na koju se nastavlja
- **komponentni model** (engl. *component-based model*) – motivacija je u pretpostavci ponovne iskoristivosti postojećih komponenata.

Osim ta tri generička modela, u novije vrijeme kao zasebni modeli procesa programskog inženjerstva naglašavaju se:

- **unificirani proces** (engl. *Unified Process, UP*) – temeljen na oblikovanju uporabom modela (engl. *model-based design*, *model-based software engineering*)
- **agilni razvoj** (engl. *agile development*).

U stvarnosti, unificirani proces i agilni razvoj čine samo podmodele tri osnovna generička modela razvoja. Naime, tri generička modela programskog inženjerstva, vodopadni, evolucijski i komponentni nisu potpuno međusobno isključivi te ih kod razvoja nekih sustava ima smisla kombinirati. Na primjer, evolucijski pristup u načelu nije pogodan za razvoj velikih sustava budući da kretanje u razvoj bez jasno definiranih temelja sustava, koji se dobivaju iz analize korisničkih zahtjeva, i nema velikog smisla. Međutim, dijelovi velikog sustava 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 toliko rašireno da je gotovo nemoguće zamisliti razvojni proces koji se temelji na potpuno nezavisnoj izradi vlastitih komponenti. Komponentno usmjereni razvoj često je i neformalno utkan u velik broj razvojnih procesa u različitim fazama razvoja.

Tako i unificirani proces i agilni razvoj koriste i evolucijski model i komponentno usmjereni razvoj, samo to rade na donekle različite načine.

Zbog velikog značaja unificiranog procesa i agilnog razvoja, u potpoglavljima 5.2 – 5.6 svih pet modela razvoja programske potpore se opisuje 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 slijedno, jedan za drugim, vode prema 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 proizvod prolazi kroz **iteracije**. Time se želi reći da neki proizvod 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 proizvoda** – npr. što završna inačica proizvoda treba sadržavati, koje nefunkcionalne i funkcionalne zahtjeve proizvod 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, čak i vodopadni, a javljaju se u svakoj fazi procesa programskog inženjerstva.

Postoje dva međusobno ovisna pristupa iteracijama: **inkrementalni** i **spiralni**.

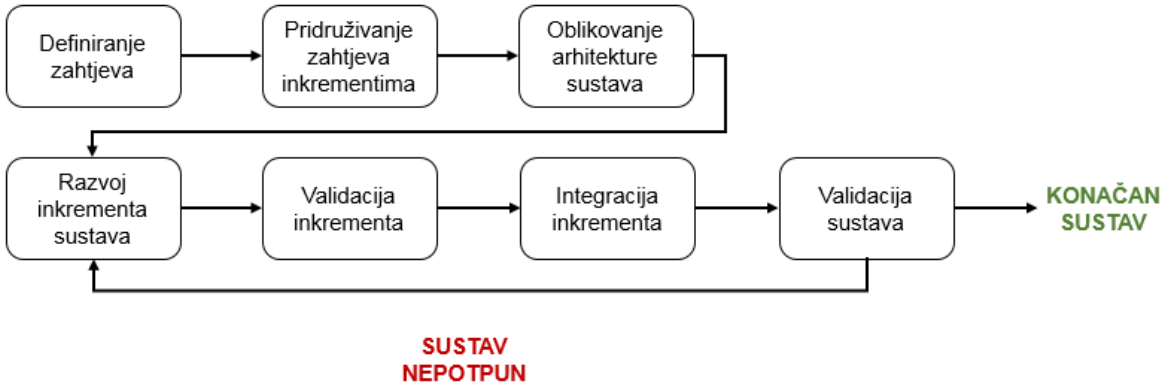
U inkrementalnom pristupu iteracijama, prikazanom na slici 5.2, sustav se ne isporučuje korisniku u cjelini. Aktivnosti razvoja, oblikovanja i isporuke razlažu se u inkrementalne dijelove koji predstavljaju djelomične funkcionalnosti proizvoda. Zahtjevi korisnika se analiziraju i organiziraju u prioritetne 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 mnogo prednosti u odnosu na neiterativni pristup:

- Kupac dobiva novu zahtijevanu vrijednost sa svakim inkrementom.

- Temeljna funkcionalnost sustava se ostvaruje u ranim fazama projekta te rani inkrementi služe kao prototipovi na temelju kojih se izlučuju zahtjevi za kasnije inkremente.
- Smanjen je rizik za neuspjeh projekta.
- 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.

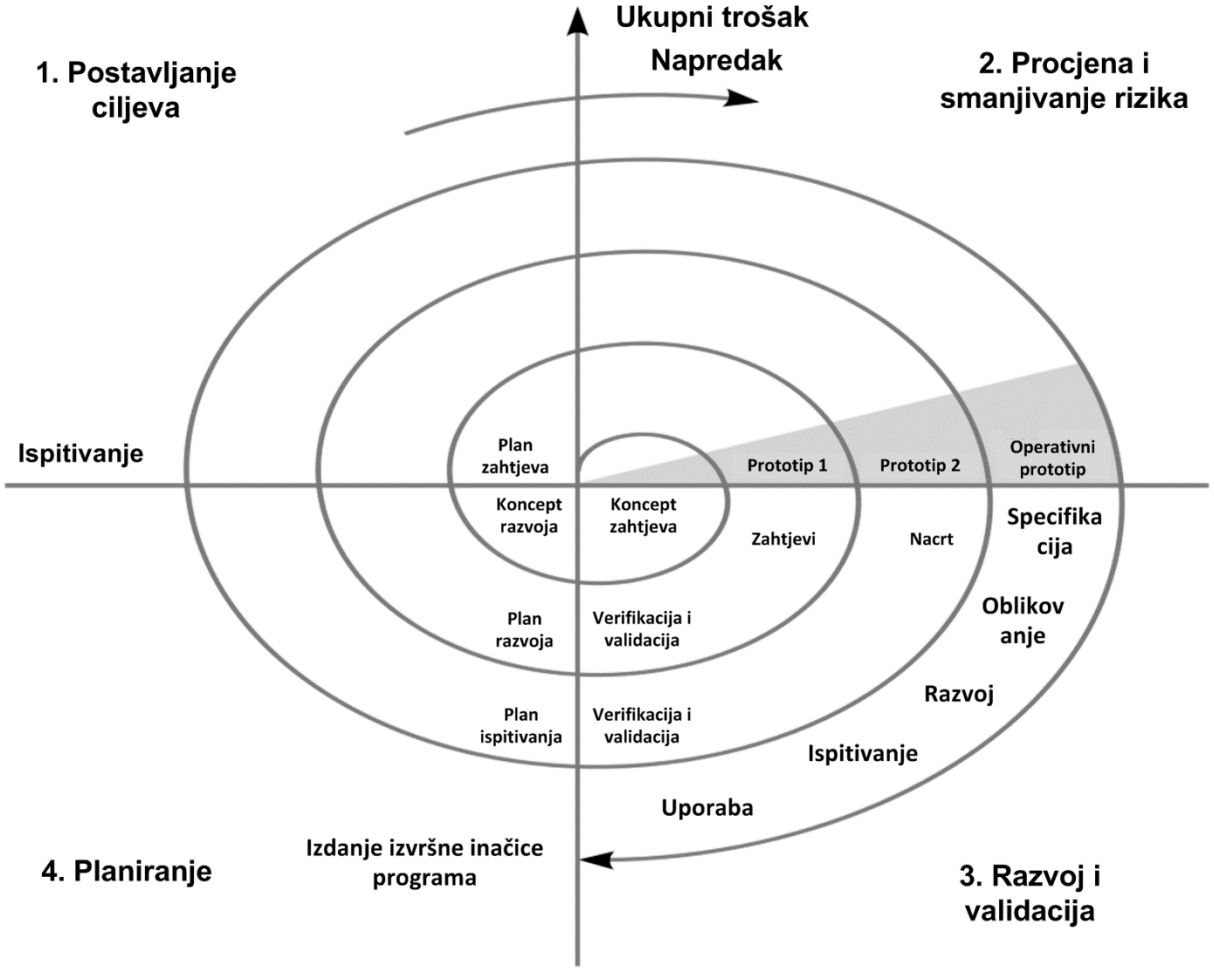


Slika 5.2 Inkrementalni pristup iteracijama, prilagođeno iz [3].

Kod spiralnog pristupa iteracijama, prikazanom na slici 5.3, proces oblikovanja predstavljen je spiralom umjesto nizom aktivnosti. Svaka petlja u spirali predstavlja jednu iteraciju procesa. U svakoj iteraciji 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.



Slika 5.3 Spiralni pristup iteracijama, prilagođeno iz [3].

U sektoru **postavljanja ciljeva** obavlja se identifikacija specifičnih ciljeva iteracije temeljem postojećih zahtjeva. U sektoru **procjene i smanjivanja rizika** analiziraju se rizici u opcijama te se preslikavaju u aktivnosti koje ih reduciraju. Analiza rizika rezultira u evoluciji prototipova. Primjer jednog takvog postupka analize rizika 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 iteracijama aktivnosti imaju sve više detalja (specifikacija, oblikovanje, razvoj, ispitivanje, uporaba). U sektoru **planiranja** obavlja se planiranje iteracija (tj. ciklusa spirale), od prikupljanja zahtjeva do oblikovanja, razvoja i konačno, integracije („*od koncepta do proizvoda*“).

5.2. Vodopadni model

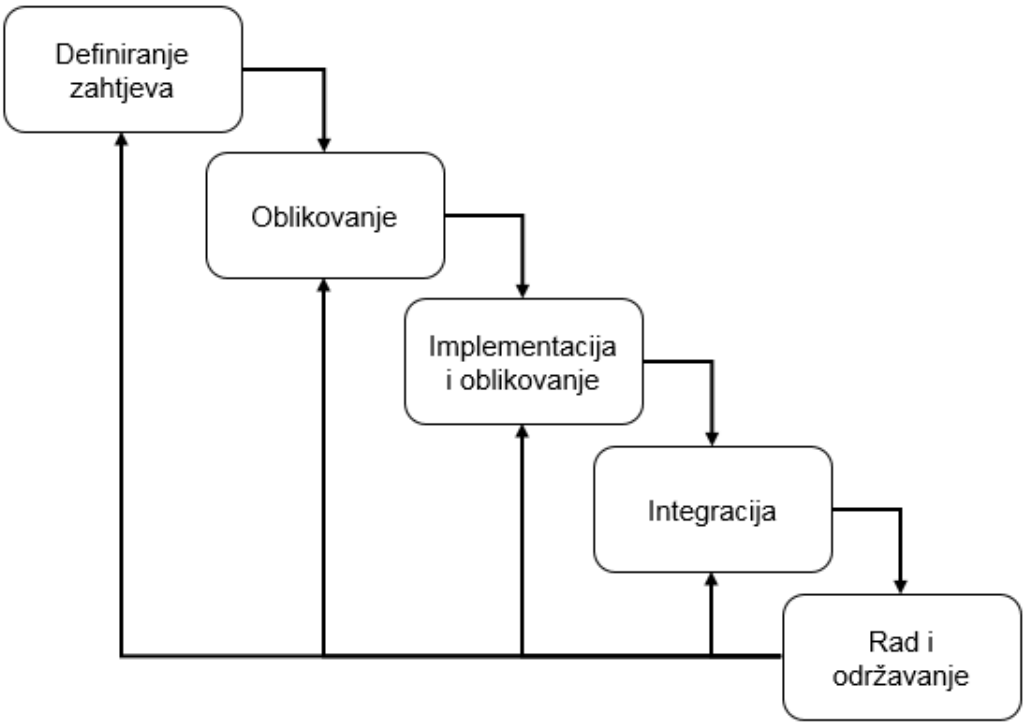
U ovom modelu, procesne faze su jasno odvojene i u načelu nezavisne. Tako se mogu zasebno promatrati faze:

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

Prema tome se pretpostavlja i da prije početka rada na pojedinim fazama postoji dobro definiran plan rada za svaku od njih. Ovaj model predstavlja "klasični" pristup projektiranju sustava gdje su koraci razvoja planski definirani i ne preklapaju se, već slijede jedan iza drugoga. Sljedeća faza započinje tek kada je prethodna faza završena.

Detaljnije, izrada sustava po fazama se sastoji od sljedećih koraka, kako je prikazano na slici 5.4:

- **definiranje** zahtjeva – konzultacije s korisnicima identificiraju koji su ciljevi sustava, koje usluge sustav treba pružati i koja su ograničenja prisutna, što sve skupa čini konačnu specifikaciju sustava.
- **oblikovanje** – definira se arhitektura sustava kroz identifikaciju programskih i sklopovskih potrebnih resursa u odnosu na postavljene zahtjeve te se na strani programske potpore određuju temeljne apstrakcije izvedbe i njihovi odnosi.
- **implementacija i ispitivanje** – ostvarenje sustava kroz skupove programskih jedinica za koje se pritom propisno ispituje odgovaraju li svojim specifikacijama.
- **integracija** – programske jedinice se ujedinjuju u cjelinu uz neizostavno ispitivanje cijelog sustava, nakon čega bi sustav 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 prema novim funkcionalnim zahtjevima i uvjetima 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 faze uvijek djelomično preklapaju te postoje povratne veze od trenutačnih 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, dokumentacija se 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 lagano razumljive zahtjeve s malom vjerojatnošću njihove promjene. Kao model upravljanja projektom, vodopadni model je najpogodniji te se uglavnom koristi za velike inženjerske projekte, često ne samo programske već i sklopovske i procesne, gdje se sustav razvija na nekoliko odvojenih mjesta. Također je prikladan ako je moguće

uspostaviti i formalnu specifikaciju sustava temeljenu na matematičkom modelu što osobito ima smisla za sustave koji su kritični u pogledu sigurnosti.

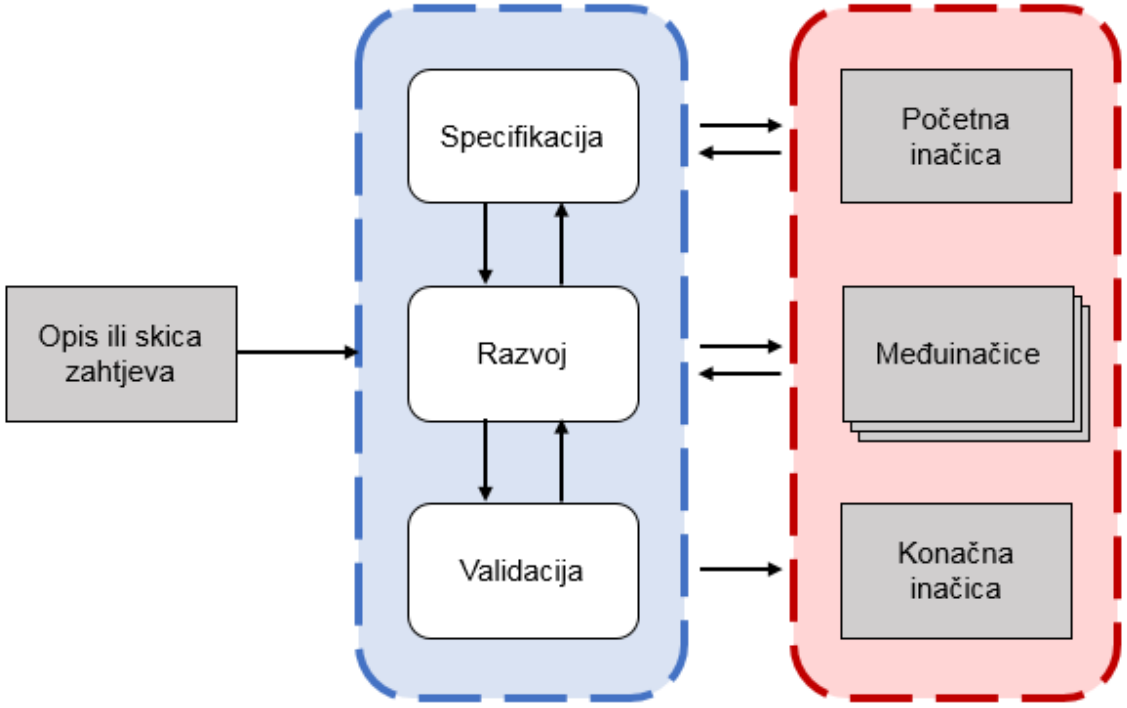
5.3. Evolucijski model

Ovaj model u literaturi spominje se i 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.

Karakteristika evolucijskog modela je ispreplitanje faza **specifikacije**, **razvoja** i **validacije**, koji imaju više narav pojedinih projektnih aktivnosti nego pojedinih faza izrade čitavog projekta. Između susjednih aktivnosti, učestalija je povratna komunikacija nego u vodopadnom modelu pa bi se moglo govoriti i o određenoj istodobnosti (engl. *concurrency*) među njima, kao što je prikazano na slici 5.5. Dva su osnovna tipa evolucijskog modela razvoja programske potpore:

- **istraživački razvoj i oblikovanje** (engl. *exploratory development*) – prisutan je stalan rad s naručiteljem radi identifikacije pravih zahtjeva. Razvoj započinje u dijelovima sustava za koje su zahtjevi dobro definirani, a nakon toga se dodaju nove funkcionalnosti prema prijedlozima naručitelja.
- **metoda odbacivanja prototipa** (engl. *throw-away prototyping*) – cilj je bolje razumijevanje zahtjeva naručitelja i na osnovi toga bolja specifikacija zahtjeva. Ovaj pristup predviđa izradu prototipova sustava koji se ne koriste u konačnici. Prototipovi se razvijaju na temelju slabo definiranih zahtjeva i služe samo kao pomoć pri razjašnjavanju stvarnih potreba korisnika.

Evolucijski model programskog inženjerstva odražava način na koji 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 najhitnije funkcionalnosti pa ih korisnik već u ranim razdobljima projekta može vrednovati. U slučaju neispunjavanja ili promašaja zahtjeva ispravci pogrešaka su jefitiniji jer je potrebno mijenjati samo jednu ili svega nekoliko zadnjih inačica koje prethode. U odnosu na vodopadni model, evolucijski pristup razvoju je značajno brži, posebice u početnom razdoblju rada te time čini osnovu agilnog razvoja programske potpore. U evolucijskom pristupu može se kombinirati i planski pristup te stoga postoji više različitih načina pogleda na evolucijski model programskog inženjerstva. Kod više planskog pristupa, inkrementi se određuju unaprijed i prema točno utvrđenom planu, a kod agilnog se prvi inkrementi brzo razvijaju dok razvoj budućih inkremenata ovisi o brzini napretka i korisničkim prioritetima. Jedna od



Slika 5.5 Evolucijski model procesa programskog inženjerstva.

karakteristika evolucijskog modela koja može biti i prednost je i ta 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 im nije lako kvantificirani stvarni napredak. Ako se sustav razvija brzo, stalno dokumentiranje inačica ne mora biti uvijek isplativo. Iz istog razloga i zbog stalnih izmjena, struktura sustava je često narušena. Izmjena strukture programskog koda (engl. *refactoring*) je tada u pravilu iznimno zahtjevna, kao i ponovna uporaba dijelova sustava.

Evolucijski model je pogodan za male i srednje sustave ili za dijelove većih sustava kojima korisnički zahtjevi nisu 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

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 20. stoljeća s motivacijom postizanja bolje i šire primjene principa ponovne uporabljivosti (engl. *reuse*) programskih komponenata. Kao najjednostavniju definiciju pojma komponente može se dati:

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

Iako se ova 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 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:

- **ostvareno** ili eksportirano sučelje (engl. *exported interface*) – definira usluge koje komponenta pruža. Ovaj tip sučelja definira popis metoda koje mogu biti pozvane od strane korisnika komponente.
- **zahtijevano** ili importirano sučelje (engl. *imported interface*) – specificira koje usluge moraju biti na raspolaganju komponenti da bi ona bila operabilna. Ovaj tip sučelja navodi koje vanjske usluge moraju biti osigurane komponenti kako bi ona uspješno funkcionirala. 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, realizirano sučelje prikazuje se kružićem, a zahtijevano polukružićem na kraju linije koja ide od simbola koji predstavlja jezgru komponente, kao što je prikazano na slici 5.6 .

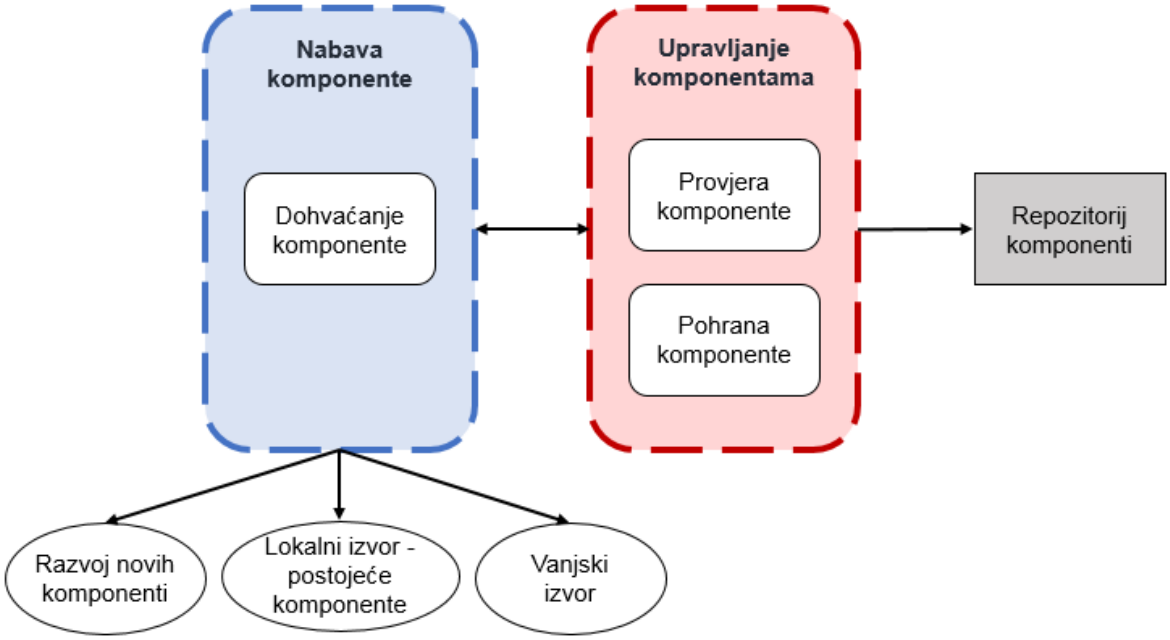
Slika 5.6 UML-notacija komponente.

S obzirom na neodvojivost pojma komponentno-usmjerenog programskog inženjerstva od ideje ponovne uporabljivosti komponente, na procese te grane programskog inženjerstva može se gledati s dva različita stajališta, a to su:

- **razvoj za ponovnu iskoristivost** (engl. *development for reuse*) – komponente ili usluge koje se grade su predviđene za iscrpno korištenje u drugim razvojnim procesima. U ovom tipu razvoja, programski kôd komponente je u potpunosti otvoren, a često se proces izrade programske potpore svodi na generalizaciju postojećih komponenti za koje postoje naznake višestruke uporabljivosti.
- **razvoj s korištenjem postojećih komponenti** (engl. *development with reuse*) – nove aplikacije se grade 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.

Pojam komponentno-usmjerenog razvoja u konkretnim situacijama ponajprije se veže uz korištenje postojećih komponenti. Sustav se integrira višestrukom uporabom postojećih komponenata ili uporabom komercijalnih, gotovih komponenata (engl. *COTS - commercial-of-the-shelf*).

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. Nabava komponente dohvaća komponente iz lokalnog izvora, tj. razvoja komponenti, ili iz vanjskog izvora radi daljnjeg razvoja novih komponenti ili korištenje istih unutar novog sustava. Te komponente predaju se procesu upravljanja komponentama koji ih šalje na provjeru i pohranu u repozitorij komponenti. Provjera komponente sastoji se od nekog oblika certifikacije da komponenta ima funkcionalnosti navedene u specifikaciji.



Slika 5.7 Pregled procesa komponentno-usmjerenog programskog inženjerstva.

Komponentni model definira 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. Norma komponentnog modela za svaku komponentu treba uključivati sljedeće informacije:

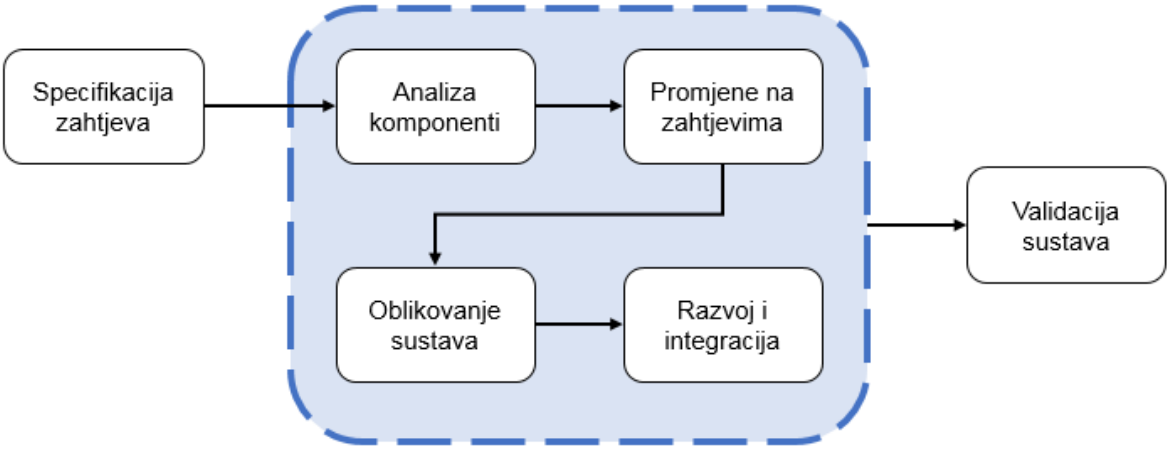
- definicija sučelja – uključuje nazive operacija koje komponenta obavlja, parametre koje pri tome prima, opise iznimaka i jezik kojim se sučelje definira.
- uputa za korištenje – sadrži jedinstveni naziv komponente po kojem joj se može pristupiti, zatim daje informacije o sučeljima i atributima te konačno određuje način na koji se komponenta može primjenjivati za različite usluge.
- uputa za isporuku – 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, uputa svakako treba uključivati temeljnu dokumentaciju koja opisuje sadržaj i unutarnju organizaciju komponente kako bi se lakše identificirala kao komponenta koja je prihvatljiva za određenu namjenu. Konačno, dobro bi bilo da uputa uključuje opis uvjeta za daljnju nadogradnju ili zamjenu prema promjenama korisničkih zahtjeva.

Gotovo svaki projekt izrade programske potpore danas ima elemente razvoja s korištenjem postojećih komponenti. Ako ta činjenica već nije tako unaprijed formalno istaknuta, ponovno iskorištavanje postojećeg kôda se događa često i neformalno, tj. dizajneri koriste ono za što znaju da postoji, a 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:

- objekti – predstavljeni svojim atributima i pojedinačnim funkcijama ili procedurama kao najniže razine apstrakcije.
- komponente aplikacije – veličina se može kretati od njenih većih dijelova ili podsustava do pojedinih modula ili objekata.
- gotova aplikacija – može se do određene mjere prilagoditi različitim korisnicima.

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 prikazanih na slici 5.8. Uvodna faza, specifikacija zahtjeva, i završna faza, validacija sustava, mogu se usporediti sa sličnim fazama drugih procesnih modela, dok su ostale faze potpuno specifične za komponentno-usmjereni model, a to su:

- **analiza postojećih komponenti** – traženje komponente pogodne za specifičnu namjenu. Uglavnom se pronađu komponente koje ispunjavaju samo dio traženih funkcionalnosti.
- **promjene postavljenih zahtjeva** – 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 iskoristiti pronađene komponente.
- **oblikovanje sustava s komponentama** – oblikovanje strukture sustava uzimajući u obzir dostupne komponente. Za pojedine zahtjeve definiraju se nove komponente kao i elementi integracije sustava.
- **razvoj i integracija sustava s komponentama** – razvija se programska potpora koja nedostaje i zajedno s pribavljenim komponentama integrira u novi sustav.



Slika 5.8 Proces programskog inženjerstva temeljen na ponovnoj iskoristivosti komponenti.

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 kada 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 održavanja komponente 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 uporabivosti 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 pri komponentnom pristupu razvoju je sklonost inženjera da napiše svoj kôd umjesto da koristi 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.

Više o konceptu i aplikaciji ponovne iskoristivosti dijelova programske potpore može se pronaći u poglavlju 6.

5.5. Unificirani proces (UP)

Razvoj oblikovanja pomoću modela započeo je ranih 1980-ih u tvrtci Ericsson kao odgovor na potrebu za novom projektnom metodologijom koja će optimalno odgovarati tvrtki pri razvoju novih proizvoda u programskom inženjerstvu, ali i ne samo u njemu. Prema verziji razvoja oblikovanja pomoću modela 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.

Definicija 5.2. Unificirani proces (engl. *Unified Process*, UP) 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.

Nakon 2003. godine, razvoj UP-a nastavljen je u tvrtci Rational Software unutar IBM-a te je takav model bio dugo vremena poznat kao *Rational Unified Process* (RUP). Tek kasnije, pojavom drugih i otvorenijih rješenja, govori se o samostalnom modelu procesa programskog inženjerstva pod nazivom UP, koji se uz različite nadogradnje održao sve do danas.

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 UP-a vodilo računa o fleksibilnosti i budućim proširenjima, jer se time omogućuju razne strategije životnog ciklusa projekta. Na primjer, moguće je odabrati koje artefakte treba proizvesti, mogu se definirati nužne aktivnosti i potrebni resursi (ukupni trošak, potrebno vrijeme, nužni razvojni inženjeri, vještine i znanja, programska podrška, sklopovlje, itd.), te modelirati koncepte sustava.

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

1. **početak** (engl. *inception*) – definiraju se doseg projekta, razvoj modela poslovnog procesa i specifikacija početnih zahtjeva
2. **elaboracija** (engl. *elaboration*) – definiraju se plan projekta, specifikacija značajki i temelji arhitekture sustava
3. **izgradnja** (engl. *construction*) – sastoji se od oblikovanja, implementacije i ispitivanja
4. **prijenos proizvoda korisnicima** (engl. *transition*) – konačni proizvod se 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 arhitecture*)
3. početna sposobnost (engl. *initial operational capability*)
4. izdavanje izvršne inačice programa ili proizvoda (engl. *release*).

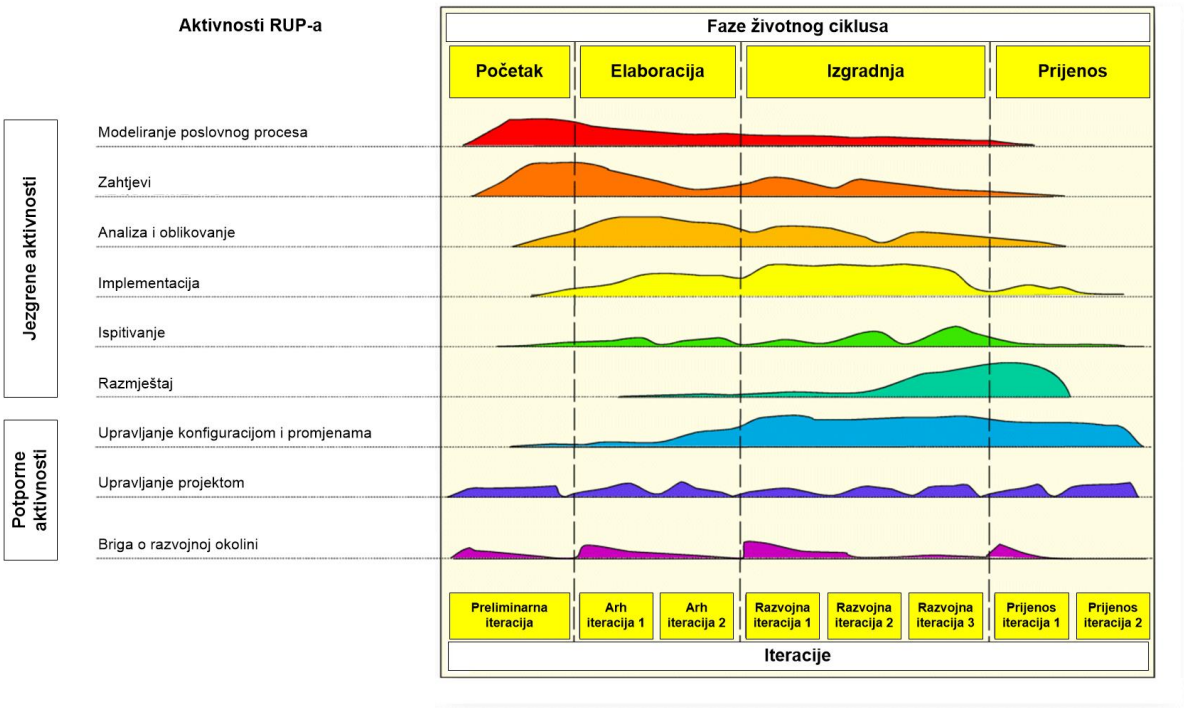
Svaka faza sastoji se od barem jedne iteracije, a moguće i više njih. Stoga, u kontekstu UP-a, iteracija je slijed aktivnosti u okviru prihvaćenog plana i kriterija evaluacije. Ishod svake iteracije je dokument, a na kraju i jedna izvršna inačica (tj. izdanje) programa ili sustava.

Osim faza životnog ciklusa, UP definira i niz aktivnosti koje se mogu odvijati u svim fazama, ali obično s različitim intenzitetom. **Jezgrene aktivnosti UP-a** (engl. *core workflows*) 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. *testing*)
6. razmještaj (engl. *deployment*).

U nekim verzijama UP-a, skup aktivnosti može biti drugačije definiran. Vrlo često se određuju dodatne, potporne aktivnosti vezane uz organizaciju i provedbu životnog ciklusa, 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 povoljnu i produktivnu interakciju dionika 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 izvršavanja prikazani su na slici 5.9. Osim što se u svojoj strukturi temelji na iteracijama u svakoj pojedinoj fazi, UP intenzivno koristi obrasce uporabe (engl. *use cases*) i usmjeruje se na arhitekturu sustava, odnosno na strukturu programske potpore koja se izrađuje.

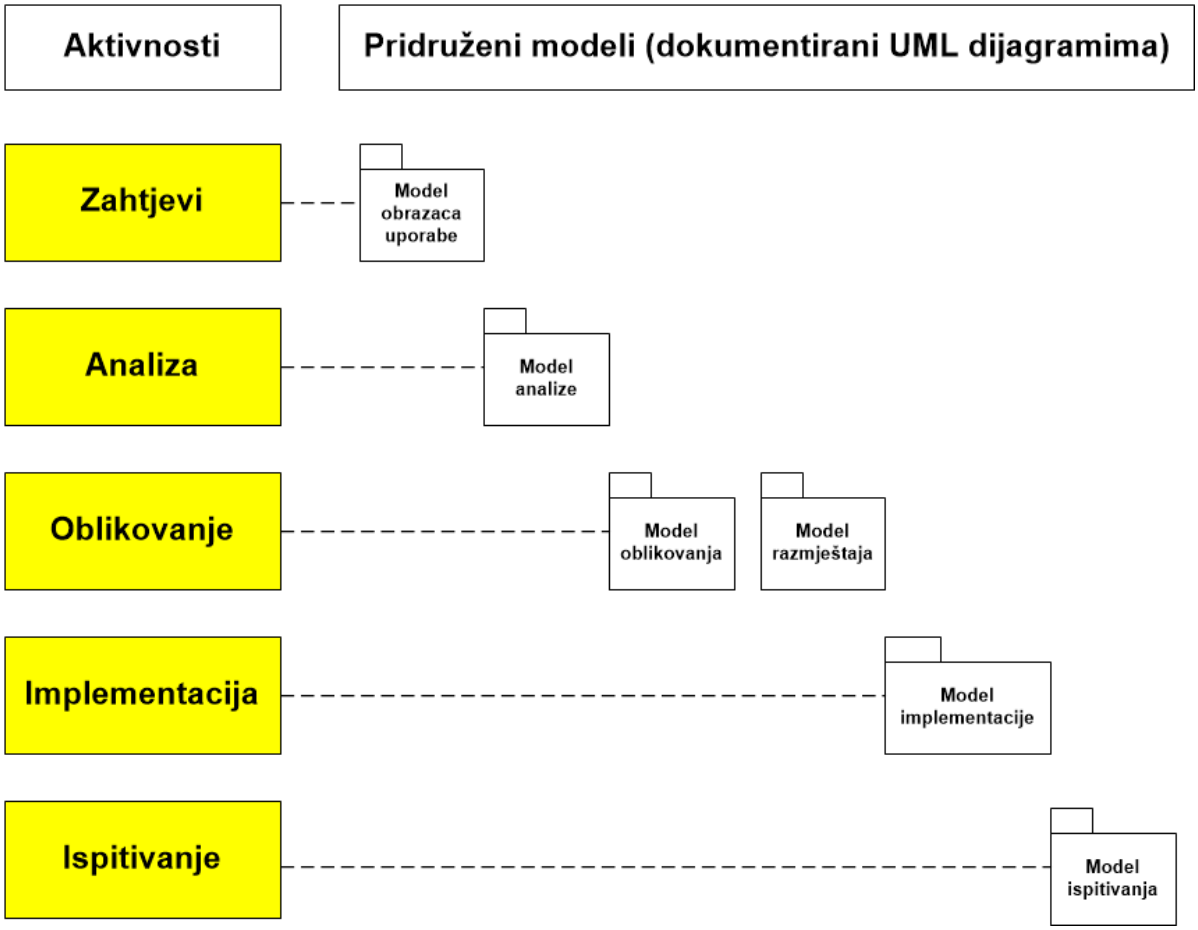


Slika 5.9 Faze i aktivnosti, s pretpostavljenim intenzitetom životnog ciklusa unificiranog procesa

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

Za opis pojedinih aktivnosti koriste se odgovarajući modeli. Modeli su dokumentirani s jednim ili, u praksi češće, više dijagrama programske potpore. Prema UP-u, dijagrami moraju biti definirani UML standardom. Stoga, oblikovana arhitektura sustava sadrži skup UML dijagrama kojima prikazuju različite poglede (engl. *views*) na sustav. Svakoj aktivnosti pridružen je jedan ili više modela za opis, koji su

dokumentirani s jednim ili više UML dijagrama (tj. pogleda), što je ilustrirano na slici 5.10 .



Slika 5.10 Primjer aktivnosti i pridruženih UML modela u unificiranom procesu.

UP je najčešće opisan kroz tri perspektive koje se ogledavaju i u korištenim UML dijagramima:

- dinamička perspektiva – pokazuje slijed faza procesa kroz vrijeme
- statička perspektiva – pokazuje aktivnosti u pojedinim fazama procesa
- praktična perspektiva – sugerira aktivnosti kroz iskustvo i dobru praksu.

5.6. Agilni pristup razvoju programske potpore

Kako bi odgovorile na dinamičnost suvremenog tržišta, tvrtke trebaju moći brzo reagirati i prilagoditi se promjenama. Budući da se programska potpora nalazi u središtu svake poslovne operacije, razvoj novih poslovnih aplikacija treba uspješno pratiti zahtjeve tržišta. Pri tome se kao najvažniji zahtjev često nameće sposobnost

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 programske potpore. Stoga klasični modeli procesa 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 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:

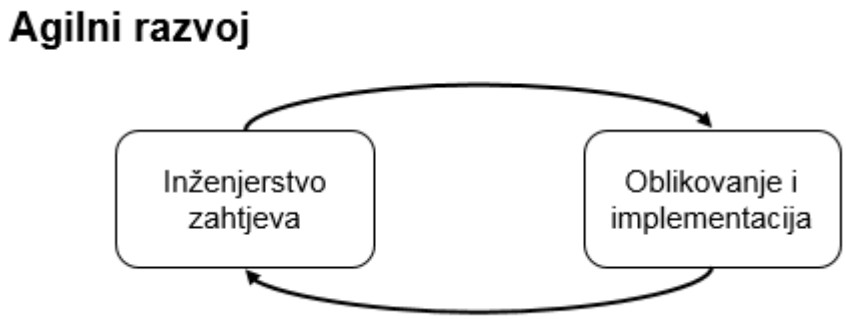
- Pojedinci i njihovi međusobni odnosi – u agilnom pristupu važni su samoorganizacija, motivacija i interakcija među članovima tima.

- 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 i bitnija od detaljnih pregovora oko ugovora.
- 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 koriste 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 kako 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. Potrebno je primijetiti da nema posebnog naglaska na fazama validacije i verifikacije i evolucije programskog proizvoda, budući da su te faze implicitno sadržane u interakciji izmjena na programski proizvod i izdavanja inkremenata.



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 2001., a neke su novijeg datuma. Neke od poznatijih i povijesno najviše korištenih su:

- Ekstremno programiranje (XP)
- Scrum
- "Čisti" razvoj (engl. *Lean development*)
- Kamban i Scrumban
- Disciplinirana agilna isporuka (DAD)
- Scrum na velikoj skali (LeSS)
- Adaptivni razvoj programske potpore (ASD)
- Crystal clear i ostale metode *crystal*
- Metoda dinamičnog razvoja sustava (DSDM)
- Razvoj vođen karakteristikama (FDD)
- Agile Unified Process (AUP).

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 „velikih“ programskih sustava koji zahtjevaju suradnju između više razvojnih timova ili pak razvoj programske potpore s posebnim zahtjevima (sigurnost, pouzdanost, odaziv u stvarnom vremenu), uobičajene agilne metode ne mogu dati zadovoljavajuće rješenje te je potreban strukturirani razvojni proces s višom razinom formalnosti u komunikaciji. Iako postoje noviji pokušaji da se postigne uspješna provedba velikih programskih projekata agilnim metodologijama, pristupi još nisu normirani i često zahtijevaju velike organizacijske izmjene u tvrtkama (npr. LeSS, SAFe, DAD).

Također, agilne metode 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 dugo nakon što je gotov proizvod isporučen. U nastavku su opisani modeli razvoja zasnovani na agilnom pristupu: ekstremno programiranje, čisti razvoj i Scrum, kao danas najčešće korištenim metodologijama.

5.6.1. Ekstremno programiranje (XP)

Ekstremno programiranje (engl. *Extreme Programming*, XP) vjerojatno je jedna od najpoznatijih i najstarijih metoda agilnog pristupa. Javlja se sredinom 1990-ih kao otpor strukturiranom (ponajviše vodopadnom) procesu programskog inženjerstva za koji se smatra da unosi previše birokracije u proces oblikovanja. U XP-u jedina mjera napretka je funkcionalni programski proizvod. Ovaj pristup zasniva se na razvoju, oblikovanju i brzoj isporuci funkcionalnih dijelova, uz stalno jedinično ispitivanje i često integracijsko ispitivanje. Pristup 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 XP-a.

Slika 5.12 Model ekstremnog programiranja u okviru agilnog razvoja programske potpore.

Dobre značajke XP-a su stalna komunikacija dionika projekta, jednostavnost razvijenih rješenja, brzi odziv i od sustava i od dionika, hrabrost i upornost programera u odabiru najboljih rješenja problema te međusobno poštivanje dionika. Glavni nedostatci ekstremnog programiranja su nestabilnost zahtjeva i nerazumljivost sustava zbog nedokumentiranja, što rezultira da postojeće rješenje uglavnom ne podupire ponovnu uporabu (*engl. reuse*) rješenja.

5.6.2. Čisti razvoj programske potpore

Čisti razvoj programske potpore (engl. *Lean development*, *Lean*) odnosi se na metodu koja je izvorno nastala u produkcijskom sustavu Toyote, a čiji je cilj razviti u što kraćem vremenu sustav koji će maksimalno zadovoljiti korisnike uz minimalni nepotrebni angažman. 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 ispitivanja i brze nadogradnje)
3. odlaganje donošenja odluke (predlaganjem opcija klijentu, skupljanjem činjenica)
4. brza isporuka (niz metoda, uglavnom mali inkrementi, više timova radi isto)
5. uvažavanje tima (motivacija i suradnja unutar tima)

6. ugradnja cjelovitosti u sustav (kupac mora biti zadovoljan sa sustavom u cjelini: funkcionalnošću, intuitivnošću korištenja, cijenom)
7. optimizacija cjeline (svaki član tima mora znati kako i zašto čisti razvoj programske potpore treba funkcionirati).

5.6.3. Scrum

Scrum (engl. *Scrum*) je radni okvir agilnog razvoja programske potpore koji je strukturiran tako da podrži razvoj kompleksnih proizvoda. Utemeljen je na empirizmu, gdje znanje dolazi iz iskustva i odlučivanja temeljenog na poznatome. To znači da se priznaje da problem koji se proizvodom nastoji riješiti nije odmah u potpunosti razumljiv i da je stoga potrebno kroz iskustvo, stalnu komunikaciju i iteracija razvoja postepeno doći do rješenja.

Scrum koristi iterativni, inkrementalni pristup za optimizaciju predvidivosti i kontrolu rizika razvoja. Scrum se sastoji od **Scrum timova** i njihovih pridruženih **uloga**, **događaja**, **artefakata** i **pravila**. Timovi su samoorganizirajući i višefunkcijski i obično uključuju do desetak članova, a uloge su:

- **vlasnik proizvoda** (engl. *product owner*) – jedini odgovoran za upravljanje projektnim dnevnikom zaostataka (engl. *product backlog*)
- **Scrum vođa** (engl. *Scrum master*) – odgovoran za razumijevanje i primjenu teorije, prakse i pravila Scruma
- **razvojni tim** (engl. *development team member*) – samoorganizirajući profesionalci koji isporučuju potencijalno isporučiv inkrement.

Scrum organizira razvoj programske potpore prema događajima (engl. *Scrum events*), koji su vremenski ograničeni i čija se osnovna jedinica razvoja naziva **sprintom** ili etapom (engl. *sprint*). Jedan sprint ograničen je period koji traje između dva tjedna i mjesec dana. Događaji su redom:

- sastanak planiranja sprinta (engl. *sprint planning*)
- sprint kao posao razvoja, unutar kojega se organizira dnevni sastanak Scrum tima (ili dnevni Scrum, engl. *daily Scrum, stand-up*)
- revizija (ili recenzija) sprinta (engl. *sprint review*)
- restrospektiva sprinta (engl. *sprint retrospective*).

Na sastanku planiranja sprinta dogovara se koliki dio funkcionalnosti programske potpore iz dnevnika zaostataka se namjerava pokriti s jednim sprintom i koji bi bio

cilj toga sprinta. U pravilu, svaka korisnička priča (engl. *user story*) dobiva broj (tzv. *velocity* ili brzinu) koji određuje relativnu težinu priče u odnosu na ostale. Uz to vezan je i pojam brzine cijelog razvojnog tima – koji definira količinu posla koju tim može obaviti u jednom sprintu. Kada se planira sprint, uzima se onoliko korisničkih priča koje sa svojom ukupnom brzinom približno odgovaraju brzini tima. Pritom se brzina tima vrednuje svakih pet sprinteva i po potrebi mijenja.

Sastanak planiranja sprinta obično traje nekoliko sati do jedan radni dan. Nakon dogovora oko detalja pokrivanja funkcionalnosti, prelazi se na razvoju dogovorenog inkrementa. Svaki dan tijekom trajanja sprinta organizira se dnevni sastanak Scrum tima, na kojem se obično u trajanju od 15 minuta kratko izvještava o napretku od jučerašnjeg sastanka, planiranom poslu za taj dan i mogućim poteškoćama na koje se naišlo. Ako postoje veće poteškoće, one se obično diskutiraju u manjim skupinama nakon sastanka. Nakon završetka sprinta, radi se revizija (2 – 4 sata), tijekom koje se pregledava što je sve napravljeno, a što se nije stiglo napraviti, demonstrira se napravljeno svim dionicima i dogovaraju se sljedeće nadogradnje. Detalji oko provedbe zadnjeg sprinta (što je bilo dobro, što ne, na čemu bi trebalo poraditi a koju praksu bi trebalo zadržati i sl.) diskutiraju se na kraju, tijekom retrospektive sprinta (1 – 3 sata). Iako retrospektiva nije uvijek nužna, preporuka je da se održava nakon svakog sprinta.

Artefakti u Scrumu predstavljaju zapisane zadatke i vrijednosti neophodne za uspjeh Scrum tima u razvoju projekta. Postoje tri glavna artefakta u Scrumu, a to su:

- **projektni dnevnik zaostataka** (engl. *product backlog*)
- **sprint dnevnik** (engl. *sprint backlog*) je skup stavaka odabrane za Sprint
- **inkrement**.

Projektni dnevnik zaostataka je prioritizirana sortirana lista svih zahtjeva na proizvod i jedino mjesto zahtjeva za bilo kakvim promjenama u sustavu. Zahtjevi su najčešće poluformalno ili neformalno predstavljeni putem obrazaca uporabe (engl. *use case*) i korisničkih priča. Vlasnik proizvoda je odgovoran za sadržaj, raspoloživost i sortiranje zahtjeva po prioritetu u projektnom dnevniku zaostataka.

Sprint dnevnik sadržava onaj popis funkcionalnosti za koji je razvojni tim procijenio da ga može implementirati u sljedećem inkrementu kao i planirani posao koji je potreban za njihovu realizaciju. Plan implementacije u okviru sprinta treba imati dovoljno detalja predočenih u sprint dnevniku da bi se na dnevnom Scrumu mogle razumjeti aktualne promjene. Iako Scrum dozvoljava izmjene sprint dnevnika tijekom sprinta, u praksi se takve promjene u pravilu ne provode, jer se time kompliciraju organizacija i implementacija.

Inkrement je zbroj svih stavki završenih tijekom sprinta i integriranih sa svim prethodnim sprintova. Rezultat na kraju sprinta je novi inkrement, koji mora biti završen. To znači da mora biti upotrebljiv za korisnika, što se rješava time da se uz svaku korisničku priču definira kriterij prihvaćenosti (engl. *acceptance criteria*). Sprint je po definiciji završen ako su sve korisničke priče koje su ušle u sprint završene. Interno, u implementacijskom smislu, završenost se određuje nekim pravilima Scruma dogovorenima unutar svake tvrtke koja prakticira Scrum. Naprimjer, inkrement treba biti *commitan* u razvojnoj grani u sustavu koji koristi Git (vidjeti poglavlje 8), ovjeren od strane ispitnog tima, spojen s glavnom granom i, konačno, ovjeren od strane vlasnika proizvoda.

6. Arhitektura programske potpore

Tijekom procesa razvoja programske potpore, korak između specifikacije zahtjeva i implementacije naziva se oblikovanje arhitekture programske potpore. Zadatak oblikovanja arhitekture je povezati apstraktne zahtjeve navedene kao pisane riječi s programom koji će imati sposobnost izvođenja u računalu.

Pojam **arhitekture programske potpore** uključuje sve one resurse koji opisuju i definiraju strukturu od koje će se programski proizvod na kraju sastojati. Različiti modeli procesa programskog inženjerstva imaju različitu razinu formalizacije kada je riječ o arhitekturi. Tako, kao što smo vidjeli ranije, UP ima dosta dobro definiran način modeliranja arhitekture sustava, fazu u kojoj se razrađuje i postupak dokumentiranja. S druge strane, agilni procesi se, u odnosu na UP, više fokusiraju na implementacijska pitanja, uz relativno slabije dokumentiranje arhitekture. Međutim, i jedni i drugi modeli razvoja programske potpore koriste dobru praksu prilikom oblikovanja arhitekture.

Arhitektura, kao apstrakcija sustava na visokom nivou, sastoji se u najopćenijem slučaju od programskih komponenata i njihovih konektora. Ona strukturira projekt i programsku potporu i osnovni je nositelj kvalitete sustava. Iako postoje različite definicije arhitekture programske potpore, može se dati sljedeću.

Definicija 6.1. Arhitektura programske potpore je struktura ili strukture sustava koje sadrže elemente, njihova izvana vidljiva obilježja i odnose između njih.

Elementi (engl. *elements*) koji se spominju su **komponente**, koje imaju svoja **sučelja**, a odnosi između njih određeni su vezama – **konektorima**. Arhitekturu određuju komponente i konektori, pri čemu **struktura** (engl. *structure*, *form*) definira ne samo povezanost komponenata već i njihovu unutrašnjost, ovisno o razini na kojoj se arhitektura modelira. Konačno, arhitekturu određuju i razlozi za odabir (engl. *rationale*). Nije svejedno kakve zahtijeve imamo (bilo poslovne bilo tehničke), jer mnogi od njih mogu utjecati na izbor arhitekture.

Razvoj arhitekture sustava smatra se kapitalnom investicijom koja se može ponovno iskoristiti. Upravo je ponovna uporaba programske potpore (nasuprot razvoja ispočetka) doprinjela kvaliteti i širini razvoja programske potpore. Više o ponovnoj uporabi može se pronaći u poglavlju 6.1. Promatrano s poslovne strane, arhitektura se izrađuje prije detaljne specifikacije programske potpore. Izrađena osnovna, konceptualna arhitektura sustava često predstavlja preduvjet za potpisivanje ugovora

na projektu izrade programske potpore. Arhitektura je osnova za komunikaciju različitih dionika, jednako kao što su to i zahtjevi na programsku potporu.

Upravo iz tih razloga, važno je uočiti da je opis i prikaz arhitekture programske potpore rezultat dokumentiranih pogleda raznih dionika. **Pogled** predstavlja djelomično obilježje razmatrane arhitekture programa i dokumentiran je **dijagramom** (nacrtom) koji opisuje strukturu sustava i sadrži elemente, njihovu povezanost i izvana vidljiva obilježja.

U povijesti razvoja programske potpore, modeliranje, detaljna razrada i dokumentiranje arhitekture programske potpore javili su se relativno kasno. Melvin Conway je 1968. primijetio da „postoji povezanost između strukture organizacije (tvrtke) u kojoj je programska potpora napravljena i strukture same programske potpore." Ta je empirijski potvrđena opservacija poznata kao **Conwayov zakon**. Naime, nedostatci u komunikaciji između članova razvojnog tima ogledali su se u nedostatcima kompatibilnosti između programskih komponenata koje su oni razvili.

Ova i slične opservacije uzrokovale su da se s vremenom arhitekturi počelo pristupati na sustavniji, organiziraniji način. Krajem 1980-ih bilo je mnogo istraživanja u području razrade arhitekture programske potpore s ciljem analize velikih programskih sustava. U počecima, istraživanja su bila temeljena na kvalitativnim opisima empirijski promatranih organizacija sustava, a kasnije se čitavo polje arhitekture programske potpore razvilo i obuhvatilo formalnije opise, alate i tehnike analize. Prije širenja objektno usmjerenih jezika 1990-ih, oblikovanje arhitekture u programskim sustavima nije bilo uobičajeno, a pogotovo ne grafički razrađeno. Umjesto toga, arhitektura (rasporedi programskih modula i njihova povezanost) se dokumentirala nakon izrade, s time da nije bilo mnogo definiranih i normiranih načina za njezino predočavanje. Tijekom 1990-ih a posebice u 2000-ima, uz širenje objektnih jezika, ojačali su i napredniji jezici za oblikovanje i modeliranje arhitekture (npr. UML) i počelo se govoriti o **oblikovanju arhitekture zasnovanom na modelima** (engl. *Model Driven Architecture*, MDA). Također, počele su se mnogo šire prihvaćati i koristiti dobre prakse pri izradi arhitekture programske potpore kao i oblikovni obrasci u programiranju.

U suvremenom razvoju programske potpore veliko značenje ima **arhitekt programske potpore**. Arhitekt je najčešće razvojni inženjer s prethodnim višegodišnjim iskustvom u programiranju, koji se kroz projekte dovoljno profilirao kako bi bio u stanju kvalitetno osmisliti arhitekturu programskog proizvoda koji će tvrtka ponuditi. Pogled arhitekta na programsku potporu je takav da omogućuje sagledavanje strukture kao rezultata skupa implementacijskih zahtjeva, uzimajući u

obzir dinamičke interakcije elemenata te njihov odnos s okolinom. Neke od značajki dobrog arhitekta programske potpore su:

- razumije potrebe poslovnog modela i zahtjeve projekta
- svjestan je različitih tehničkih pristupa u rješavanju danog problema
- vrednuje dobre i loše strane tih pristupa
- preslikava potrebe i vrednovane zahtjeve u tehnički opis arhitekture programske potpore
- vodi razvojni tim u oblikovanju i implementaciji
- koristi "meke" vještine kao i tehničke vještine.

6.1. Klasifikacija, modeli i proces izbora arhitekture programske potpore

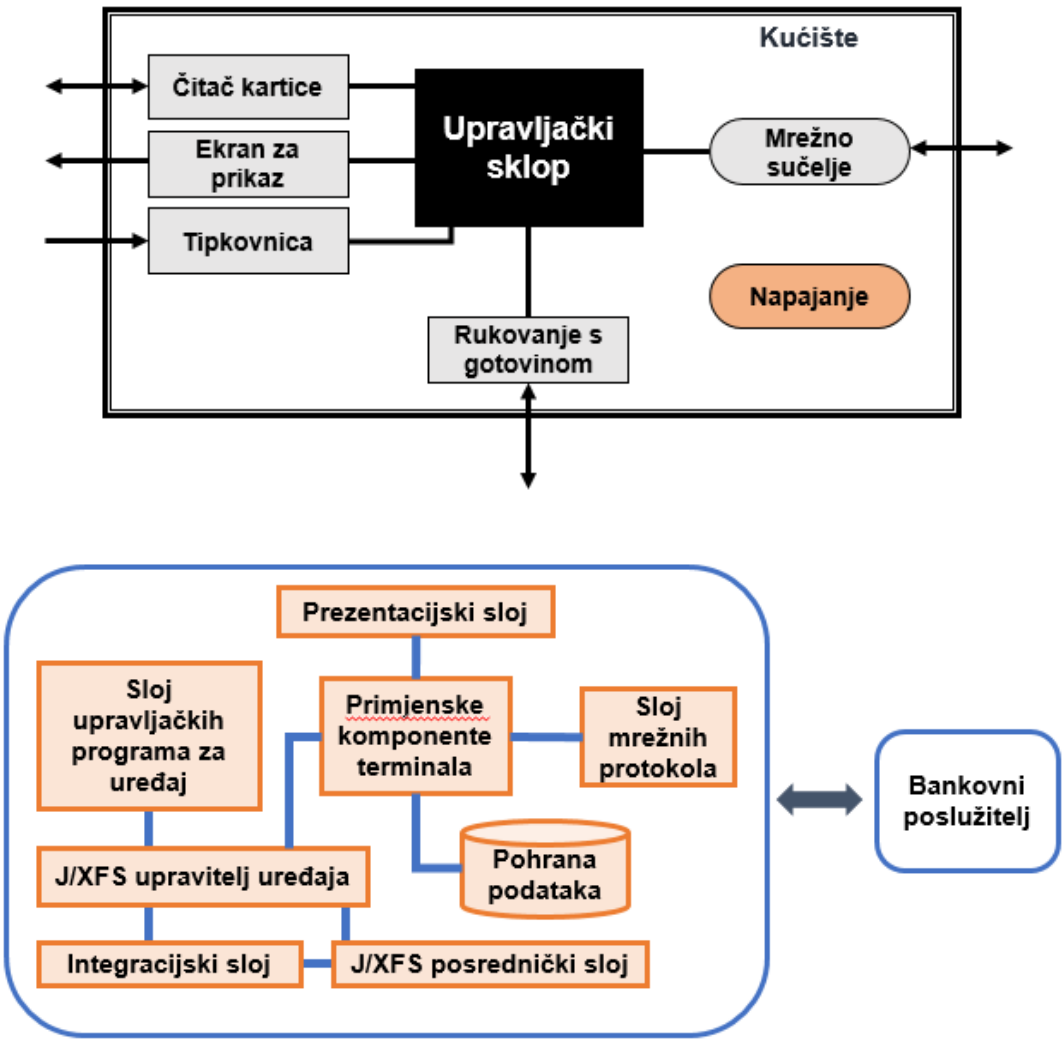
Arhitektura programske potpore najčešće se **klasificira prema dosegu** i dijeli se na:

- **koncepcijsku** (engl. *conceptual architecture*)
- **logičku** (engl. *logical architecture*)
- **izvršnu** (engl. *execution architecture*) razinu.

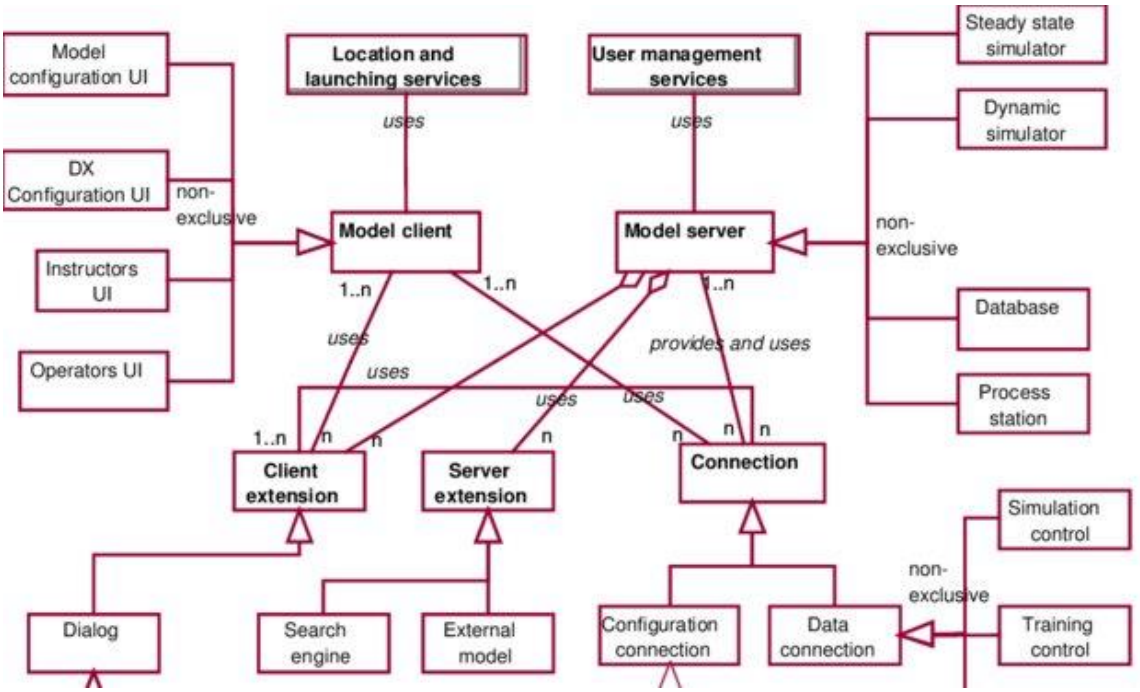
Koncepcijska arhitektura je ona najviše razine. Kod nje se pažnja usmjerava na pogodnu dekompoziciju sustava u podsustave te se određuje osnovni nacrt pojedinih podsustava. Ovakav opis arhitekture služi za komunikaciju s netehničkim dionicima na projektu (uprava, prodaja, korisnici). Za prikaz se koriste različiti dijagrami: neformalni strukturni dijagrami ili dijagrami toka i poluformalni UML dijagrami (npr. dijagram komponenata na visokoj razini, dijagram paketa, neformalna specifikacija dijagrama razreda putem tzv. CRC (engl. *Class-Responsibility-Collaboration*) kartica). Primjer konceptualnog modela arhitekture programske potpore bankomata prikazan je na slici 6.1.

Logička arhitektura je precizno dopunjena koncepcijska arhitektura. Ona predstavlja detaljan nacrt pogodan za implementaciju komponenti. Za prikaz logičke arhitekture koriste se uglavnom UML dijagrami, i to UML komponentni dijagrami koji prikazuju komponente sa sučeljima (uz specifikaciju komponenti i sučelja), UML dijagrami razreda i objekata te UML sekvencijski i komunikacijski dijagrami. Primjer logičke arhitekture sustava prikazan je na slici 6.2.

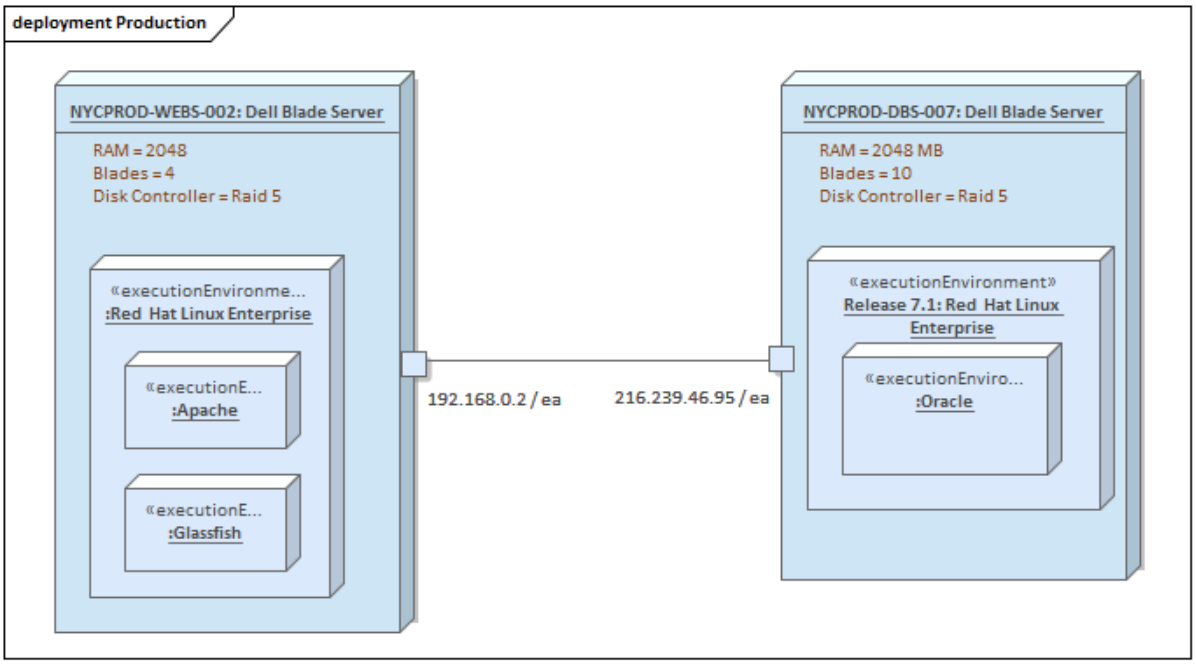
Izvršna arhitektura sustava namijenjena je prikazu sustava u izvođenju, odnosno pridruženosti programske potpore fizičkom, sklopovskom sustavu. Najčešće se koristi za prikaz raspodijeljenih i paralelnih sustava, kod kojih je sustav raspoređen na više lokacija koje međusobno komuniciraju. Za prikaz izvršne arhitekture uobičajeno se koriste UML dijagrami razmještaja, ali mogu se koristiti i drugi nenormirani dijagrami, sve dok prikazuju komponente u stvarnoj realizaciji na fizičkim strojevima. Primjer izvršne arhitekture sustava prikazan je na slici 6.3.



Slika 6.1 Konceptualni model arhitekture bankomata; gore – prikaz funkcionalnosti, dolje – prikaz programskih komponenti i njihove povezanosti, prilagođeno iz [9].



Slika 6.2 Logički model arhitekture sustava, korišten je UML dijagram razreda, a prikazan je dio sustava za simulaciju procesa, prilagođeno iz [10].



Slika 6.3 Izvršna arhitektura sustava, primjer UML dijagrama razmještaja instanci za prikaz komunikacije između dva konkretna poslužitelja, prilagođeno iz [11].

6.1.1. Modeli arhitekture

Slična klasifikacija arhitekture programske potpore u odnosu na onu prema dosegu jest ona **prema modelu** ili **pogledu na sustav**. Naime, model sustava čini više pogleda dionika na arhitekturu programske potpore. Dionici najčešće razmatraju strukturu programskog sustava, načine na koji dijelovi sustava komuniciraju u dinamičkom okruženju kao i gdje će se u konačnici ti dijelovi naći. Stoga se prema modelu (ili tipu pogleda na sustav), razlikuju:

- **statički strukturni model arhitekture – moduli** (engl. *module view type*)
- **dinamički procesni model** (engl. *component and connector view type*)
- **model alociranih elemenata** (engl. *allocation view type*).

Statički strukturni model prikazuje dekompoziciju sustava u module. U objektnoj arhitekturi module najčešće čine razredi, tako da se za modeliranje strukture sustava najčešće koristi UML dijagram razreda. Dinamički procesni model prikazuje komponente i konektore koji su na neki način organizirani tijekom izvođenja (engl. *runtime*), kao što su procesi, objekti, klijenti, poslužitelji, baze podataka. Ovi modeli uključuju puteve interakcije, kao što su komunikacijske poveznice i protokoli te pristup dijeljenoj pohrani podataka. Takvi modeli obično se prikazuju UML interakcijskim dijagramima, i to najčešće UML sekvencijskim dijagramima ili UML komunikacijskim dijagramima. Konačno, model alociranih elemenata, slično arhitekturi izvršne razine, prikazuje odnos između programske potpore i okoline (sklopovlja) u kojoj se ona izvodi, tako da se za takav model obično koriste UML dijagrami razmještaja.

Nešto stariji, ali još uvijek primijenjiv model pogleda na arhitekturu programske potpore je tzv. **„4+1 model pogleda“** (engl. *4+1 view model*) [12]. Ovaj način razmatranja arhitekture razlikuje sljedeće poglede dionika:

- **logički pogled** (engl. *logical view*) – ponašanje sustava i dekompozicija, modelirano UML dijagramima razreda/objekata, sekvencijskim i komunikacijskim dijagramima
- **procesni pogled** (engl. *process view*) – opis procesa i njihove komunikacije, modelirano UML dijagramima stanja i aktivnosti
- **razvojni pogled** (engl. *development view*) – opisuje module sustava, modelirano UML dijagramima komponenata i paketa

- **fizički pogled** (engl. *physical view*) – instalacija i izvršavanje u stvarnom, najčešće mrežnom okruženju, modeliranu UML dijagramima razmještaja
- **scenariji / obrasci uporabe** (onaj +1 pogled, engl. *Scenarios / Use cases*) – opisuju podskup obrazaca uporabe korištenih u razvoju arhitekture, modelirano UML dijagramima obrazaca uporabe. Obrasci uporabe povezuju sve ostale poglede u zajedničku funkcionalnu cjelinu.

6.1.2. Proces izbora, oblikovanja i vrednovanja arhitekture programske potpore

Proces izbora, oblikovanja i vrednovanja arhitekture programskog sustava uključuje donošenja niza odluka oko najpogodnijih rješenja za zadani problem. Arhitekt programske potpore suočava se s rješavanjem niza **problema oblikovanja** (engl. *design issues*) koji su sve potproblemi cjelokupnog problema koji mu je zadan. Budući da klijenti i korisnici ne znaju ispravno procijeniti težinu i složenost implementacije pojedinih potproblema, zahtjevi na sustav često nisu jednake težine, a očekivanja korisnika koji put nisu u skladu s mogućnostima tvrtke u izvedbi. Zadatak razvojnog inženjera – arhitekta programske potpore jest da na temelju problema iznađe više **inačica rješenja** (engl. *design options*) razmatrajući **prostor oblikovanja** koji mu je na raspolaganju (engl. *design space*) te da zatim napravi odabir najbolje opcije.

Primjer 6.1. Zadani problem uključuje izradu korisničkog foruma za jednu veliku medijsku kuću. Prostor oblikovanja programske potpore u ovom primjeru pojednostavljeno bi uključivao izbor između vrste aplikacije, programskog jezika na poslužiteljskoj strani i radnog okvira za pomoć bržem razvoju aplikacije. Vrsta aplikacije koju arhitekt razmatra je web ili mobilna aplikacija. Programske jezike koje razmatra na strani poslužitelja su Java, Python, Go i PHP. Recimo, neka se arhitekt odluči koristiti web aplikaciju i PHP. Konačno, izbor između korištenja web aplikacijskih radnih okvira, koji su potrebni za ubrzanje razvoja (vidjeti poglavlje 6.2), za PHP su: Laravel, Symfony ili Yii. Izbor ovisi o tome smatra li arhitekt da je potrebno koristiti arhitekturni obrazac MVC ili ne te je li potrebna velika modularnost programske potpore ili ne. U konačnici, arhitekt se odlučio razviti PHP web aplikaciju, koristeći radni okvir Symfony s podrškom za MVC i ponuđenom velikom modularnosti.

Proces donošenja odluke za arhitekta u općem slučaju nije jednostavan. Za donošenje ispravne odluke potrebna su znanja o zahtjevima korisnika na sustav, trenutačno već oblikovanoj arhitekturi koja mu je na dostupna, raspoloživoj tehnologiji ili tehnološkog stogu (engl. *technology stack*, često ograničenom internim pravilima tvrtke), principima oblikovanja i najboljom praksom u određenom području te dosadašnjim vlastitim dobrim rješenjima u sličnim projektima.

Zadaće donošenja odluka uključuju: postavljanje prioriteta unutar sustava, dekompoziciju sustava u podsustave te razmatranje istih, definiranje svojstava koje sustav treba imati, postavljanje sustava u kontekst (okolinu, s kojom treba komunicirati) i osiguravanje cjelovitosti sustava. Često, arhitekt treba računati na to da su tehnička i netehnička (poglavito ljudska) pitanja isprepletena te mora voditi računa o raznim dionicima koji će sustav u konačnici koristiti, njihovim preferencijama i specifičnostima. Također, različiti dionici izravno utječu na to kakvu arhitekturu treba koristiti. Potrebno je napomenuti da ponekad jednostavno nema idealnog rješenja za sve te u tom slučaju arhitekt treba predložiti najbolji kompromis.

Kako bi arhitekt donio najbolju odluku o izboru arhitekture, u praksi se često koristi sljedećim postupkom:

1. pobroji i opiše alternative odluka oblikovanja
2. pobroji prednosti i nedostatke svake opcije u odnosu na prioritete i ciljeve
3. odredi opcije koje su u sukobu s ciljevima
4. odabere opcije koje najbolje zadovoljavaju ciljeve
5. prilagodi prioritete za daljnje donošenje detaljnijih odluka.

Odluke oblikovanja arhitekture najčešće se mogu popisati tablično, svaka opcija sa svojim svojstvima. Iako se većina arhitekata na temelju iskustva odlučuje za neku od opcija, objektivniju procjenu mogu dati neki od uvriježenih postupaka procjene arhitekture. U načelu, ti postupci mogu biti zasnovnani na **tehničkom** ili na **ekonomskom vrednovanju** alternativa, a moguće je i kombinirati oba pristupa.

Kod tehničkog vrednovanja, procjena opcija radi se prema svojstvima kvalitete sustava, npr. izvedbe, sigurnosti i promjenjivosti. Jedna od mogućnosti je primjena metode analize arhitekturnih kompromisa (engl. *Architecture Tradeoff Analysis Method*, kraće: ATAM). Ukratko, ATAM se sastoji od analize poslovnih pokretača (engl. *business drivers*) sustava, što uključuje analizu funkcionalnosti, ciljeva i ograničenja. Dionici raspravljaju o poslovnim pokretačima te iz njih izrađuju scenarije. Arhitekturne odluke koje se predlažu koriste definirane scenarije korištenja kako bi se analizirali kompromisi, osjetljive točke i rizici sustava. U svakom koraku

razrade arhitekture, nanovo se sagledavaju scenariji i ocjenjuju se daljnji rizici, sve dok se svi ne izglade. Arhitektura koja je na kraju analize izabrana trebala bi biti najbolje prilagođena zadanim zahtjevima.

Kod ekonomskog vrednovanja arhitekture, najčešće se arhitektura sagledava u kontekstu ekonomskog kompromisa – sagledavaju se troškovi i koristi te se odlučuje oko toga koja arhitektura ima najpovoljniji omjer dobivenog i utrošenog. Formalno, moguće je provesti metodu analize troškova i dobiti (engl. *Cost Benefit Analysis Method*, CBAM). Kod te metode, za svaku arhitekturnu odluku razmatra se povrat investicije (engl. *Return on Investement*, ROI) te je trošak donošenja određene arhitekturne odluke jedno od njenih najbitnih svojstava. U svakom koraku analize arhitekture CBAM, određuju se prioritetni ciljevi te se nastoji ustanoviti isplati li se uvođenje predloženih komponenti arhitekture ili ne, s obzirom na dana financijska ograničenja, ROI i ostale prioritete. CBAM je najčešće ključno provesti kod arhitektura složenih sustava, gdje nije svejedno koje komponente će se koristiti i po kojoj cijeni.

Arhitekturno značajni zahtjevi (engl. *architecturally significant requirements*, *ASR*, *architecture drivers*) čine onaj skup zahtjeva na programsku potporu koji značajno utječe na izbor arhitekture. ASR su izvedeni iz funkcionalnih i nefunkcionalnih zahtjeva, a ocjenjuju ih dionici s obzirom na prioritet i doseg. To su uglavnom oni zahtjevi koji su tehnički izazovni i ograničavajući te najčešće središnji za svrhu samog sustava. Stil arhitekture (arhitekturni obrazac) najčešće određuju nefunkcionalni zahtjevi, dok funkcionalni zahtjevi određuju elemente dotičnog stila koji su uključeni u projekt te su stoga ASR najčešće nefunkcionalni zahtjevi. Ostali zahtjevi, koji nisu ASR, obično određuju detalje oblikovanja i implementaciju.

Primjer 6.2. Neki primjeri ASR-a bili bi:

- "Sustav treba bilježiti svaku izmjenu korisničkih zapisa u bazi u svrhu revizije." – uvjetuje postojanje komponente dnevnika događaja koja je povezana i s programskom logikom i s bazom podataka.
- "Sustav mora odgovoriti na zahtjev korisnika u roku 5 sekundi" – onemogućava korištenje kompleksnih radnih okvira, sporih komponenti ili sporih programskih jezika.
- "Sustav mora kriptirati sav mrežni promet" – uvjetuje postojanje komponente za računalnu sigurnost.
- "Sustav treba uspješno raditi na operacijskim sustavima Windows 10 i Linux" – uvjetuje razvoj u platformski neovisnom programskom jeziku ili istovremeni razvoj za više operacijskih sustava.

Pri oblikovanju arhitekture programske potpore, moguće je slijediti tri pristupa:

- odozgo prema dolje (od vrha prema dnu, engl. *top-down design*)
- odozdo prema gore (od dna prema vrhu, engl. *bottom-up design*)
- hibridno oblikovanje (engl. *hybrid design*).

Pristup odozgo prema dolje radi tako da arhitekt najprije oblikuje najvišu razinu strukture sustava i zatim postupno razrađuje detalje pojedinih komponenata. Tek na kraju oblikuju se format podataka, strukture podataka i algoritmi. Kod **pristupa odozdo prema gore** oblikovanje započinje donošenjem odluke o komponentama najniže razine. Cilj je razviti komponente koje će omogućiti ponovnu uporabu. Nakon toga, arhitekt donosi odluke o njihovoj integraciji u komponente više razine. **Hibridno oblikovanje** koristi oba pristupa, i oblikovanje od vrha prema dnu koje koristi za definiranje dobre opće strukture sustava kao i oblikovanje od dna prema vrhu koje se da iskoristiti za razvoj komponenti za ponovnu uporabu. U praksi, najčešći je hibridni pristup, no ovisno o problemu, bilo koji od pristupa može biti prigodan. Primjerice, ako je cilj razviti brz i točan algoritam za izračunavanje najbrže biciklističke rute temeljem zadanih točaka, onda se najčešće pristupa razvoju od dna prema vrhu. S druge strane, ako se razvija širok i kompleksan sustav, onda je češći slučaj postupna razrada arhitekture, koja kreće od vrha prema dnu.

Postupna razrada arhitekture složenog sustava započinje s grubom skicom arhitekture zasnovanom na osnovnim zahtjevima i obrascima uporabe. Pritom se uzimaju u obzir arhitekturno specifični zahtjevi te se određuju temeljne potrebne komponente sustava. U ovoj početnoj fazi, odabire se arhitekturni stil (vidjeti

poglavlje 6.4) pri čemu je moguće odabrati samo jedan ili kombinaciju više njih. Dobar savjet je da nekoliko timova, neovisno jedni o drugima, naprave grubu skicu arhitekture te se tijekom rasprave odaberu najbolje ideje. Arhitektura se zatim nadopunjava detaljima, tako da se na temelju korisničkih zahtjeva i prema iskustvu razvojnih inženjera identificiraju osnovni načini komunikacije i interakcije između podsustava i komponenata unutar podsustava. Određuje se kako će se dijelovi podataka i funkcionalnosti raspodijeliti između komponenata. Razmatraju se svi pojedini obrasci uporabe i podešavaju se detalji arhitekture. Konačno, pokušavaju se identificirati dijelovi sustava koji bi se mogli iskoristiti za ponovnu uporabu.

Tijekom oblikovanja arhitekture, moguće je klasificirati arhitekturne odluke, koje se dijele na:

- **strukturne odluke** (engl. *structural decisions*) – odnose se na stvaranje podsustava, nivoa, komponenata, itd.
- **ponašajne odluke** (engl. *behavioral decisions*) – odnose se na formiranje interakcija u sustavu
- **odluke o svojstvima (naputci)** – uključuju smjernice ili vodilje (engl. *design rules, design guidelines*) i ograničenja oblikovanja (engl. *design constraints*)
- **izvršne odluke** – poslovne odluke koje utječu na metodologiju razvoja, ljude, alate.

Kod dobrog oblikovanja programske potpore, ciljevi su:

- smanjenje cijene i povećanje profita
- osiguranje sukladnosti sa zahtjevima
- ubrzanje razvoja i implementacije
- poboljšanje kvalitete i to posebice uporabljivosti, efikasnosti, pouzdanosti, lakoće održavanja i ponovne uporabe.

Načela dobrog oblikovanja programske potpore koji doprinose ostvarenju ovih ciljeva detaljno se razmatraju u poglavlju 6.3.

6.2. Ponovna uporaba programske potpore

6.2.1. Podjela načina ponovne uporabe programske potpore

Tijekom godina napretka u razvoju programske potpore, razvijeni su mnogi koncepti i pristupi koji se u nekoj mjeri temelje, uključuju ili podupiru koncept **ponovne uporabe** (engl. *reuse*) programske potpore (osim zasebno definirane grane komponentno-usmjerenog programskog inženjerstva opisane ranije). Jedan od najčešće korištenih rezultata ovog napretka su **programske knjižnice (ili biblioteke)** (engl. *libraries*). Programske knjižnice ostvaruju različite funkcionalnosti koje se često javljaju u praksi. One su specifične za pojedine programske jezike, a osim kao ugrađene u samu normu programskog jezika, one dolaze i kao neovisne komponente za slobodnu ugradnju u vlastitu programsku potporu. Programske knjižnice su dobro osmišljene, oblikovane i ispitane komponente koje olakšavaju izgradnju određenog novog programskog proizvoda tako što nude određenu funkcionalnost koju se može iskoristiti i ugraditi, a koju bi bilo dugotrajno i skupo reimplementirati u određenom programskom jeziku.

Osim knjižnica, u okviru koncepta ponovne iskoristivosti postoje i druga rješenja. Na višoj razini apstrakcije, pojavljuju se **arhitekturni obrasci ili stilovi**, kao apstrakcije pri ponovnom oblikovanju i ostvarenju sustava. Arhitekturni stilovi se posebno obrađuju u poglavlju 6.4. Osim arhitekturnih stilova, kao koncept ponovne uporabe na nižem nivou razvili su se i **oblikovni obrasci** (engl. *design patterns*), koji predstavljaju opće apstrakcije odnosa između konkretnih objekata, koje se češće primjenjuju i to na razini implementacije sustava.

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

- vremenski okvir u kojem se očekuje da se treba donijeti odluka o korištenju 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
- osobitost domene primjene programske potpore
- izbor platforme za koju je programska potpora predviđena.

Osim knjižnica, arhitekturnih i oblikovnih obrazaca, u raznolikosti oblika ponovne iskoristivosti elemenata programske potpore, nekoliko se njih može izdvojiti kao često korišteni pristupi:

1. računalni razvojni radni okviri (engl. *application framework*) ili kraće, radni okviri (engl. *framework*)
2. primjenske linije proizvoda (engl. *software product line*)
3. gotovi proizvodi programske potpore (engl. *commercial-off-the-shelf*, COTS).

Uz ovdje spomenute pristupe, kao poseban koncept razvilo se 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. Razrađeni modeli mogu se ponovno iskoristiti, a u prilog korištenju modela za ponovnu uporabu govore i generatori programske potpore (engl. *program generators*), koji na temelju razumijevanja funkcioniranja određene vrste aplikacije i modela sustava generiraju kôd konačnog sustava.

U narednim će se potpoglavljima detaljnije analizirati:

- razvojni radni okviri – temelje se na doprinosima objektno usmjerenog pristupa
- primjenske linije proizvoda – temelje se na iskustvima dugotrajnijeg razvoja i izlučivanja komponenti s generičkim svojstvima
- gotovi programskih proizvodi – komercijalno dostupne komponente
- oblikovni obrasci.

6.2.2. Radni okviri

Objektno usmjereni 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 previše specifični da bi ih se lako razumjelo i upotrijebilo na najrazličitijim mjestima te iz toga proizlazi da ih je lakše ponovno implementirati nego prilagoditi svojim potrebama. Zato se koncept ponovne iskoristivosti u objektnom usmjerenom pristupu više rabi na razini apstrakcije **razvojnih radnih okvira,** kao struktura koje u jednom obuhvaćaju brojnije skupove elemenata (razreda, sučelja, komponenti) programske potpore.

Definicija 6.2. Radni okvir je skup integriranih komponenti s jasno definiranim sučeljima koji omogućuje ponovnu uporabu arhitekture za učestalo korišteni dio programske potpore.

Radni okviri daju podršku programima koji imaju osnovnu sličnost te im omogućuju ostvarivanje zajedničkih generičkih funkcionalnosti. Radni okviri često su specijalizirani za određene aspekte funkcionalnosti programa koje se razvijaju. Naprimjer, postoje radni okviri za ostvarenje korisničkih sučelja, koji stoga u sebi uključuju podršku za prihvat i obradu korisničkih akcija u sučelju (engl. *event handling*) i skupove gotovih prozorčića ili okvira (engl. *widgets*) za prikaz.

Razvojni radni okviri nude **okosnicu** (engl. *skeleton*) buduće arhitekture aplikacije, tj. osnovu za razvoj korisničkih programa i programskih knjižnica, što u sebi uključuje i višestruko korištenje elemenata s istom generičkom svrhom. U izvedbi, radni okviri se konkretno svode na organizirane skupove razreda kao elemenata koji su vidljivi i na raspolaganju za daljnju uporabu i primjenu objektno usmjerenog pristupa. Uglavnom se radi o skupovima predefiniranih razreda za pojedine aspekte funkcionalnosti pa se razvoj svodi na proširenja dodavanjem novih razreda koji nasljeđuju razrede ponuđene unutar radnog okvira. U tom kontekstu obično se razlikuju **horizontalni** i **vertikalni** radni okviri. Kod horizontalnih, manji dio funkcionalnosti je implementiran, a očekuje se više proširenja radnog okvira. Kod vertikalnih radnih okvira postoji mnogo implementirane funkcionalnosti, uz mogućnost tek manjih proširenja. Aplikacije razvijene uz korištenje radnih okvira mogu biti dobra osnova za daljnji razvoj različitih primjenskih proizvodnih linija specijalizirane namjene. Razvojni radni okviri su u pravilu potpuno vezani uz pojedine programske jezike.

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 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-temeljene (engl. *object request broker*) radne okvire, koji se koriste za razmjenu poruka i za transakcijske baze podataka. Konkretni primjeri takvih radnih okvira, kao što su Microsoft .NET i Enterprise JavaBeans (EJB) daju podršku za svoje normirane komponentne modele (vidjeti i poglavlje 6.4.5).

- **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, su skuplji, jer je izravnije podržan razvoj korisničkih aplikacija kroz prethodno ugrađeno znanje iz domene primjene. Za razliku od onih prije navedenih, poslovni radni okviri manje su fokusirani na interna pitanja razvoja programske potpore, ali im je dugoročna dobit veća.

U novije vrijeme, kao zasebna vrsta, razvijaju se **radni okviri za razvoj web aplikacija** (engl. *web application frameworks*). Najčešće su zasnovani na arhitekturnom obrascu *Model-View-Controller* (MVC) 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 najčešće u sebi uključuju jedan ili više manjih radnih okvira koji su specijalizirani za određene aspekte funkcionalnosti programske potpore kao što su: sigurnost, podrška za dinamičke web stranice, rad s bazom podataka, upravljanje korisničkim akcijama i sjednicama (engl. *session management*). Primjeri trenutačno popularnih (2019.) radnih okvira za razvoj web aplikacija su:

- Spring i JavaServer Faces (Java)
- Angular i Vue.JS (JavaScript)
- Laravel i Symfony (PHP)
- Django i Flask (Python)
- ASP.NET (CLI jezici)
- Ruby on Rails (Ruby).

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 potrebno za sveobuhvatnije razumijevanje i za naknadne popravke ako se kod implementacije nešto previdjelo. Nešto više o razvoju raspodijeljenih aplikacija putem radnih okvira za web može se pročitati u poglavljima 6.4.4 i 6.4.5.

6.2.3. Primjenske linije proizvoda

U odnosu na razvojne radne okvire, kao jednog pristupa koji podržava koncept ponovne uporabe komponenti, primjenske linije proizvoda imaju izvor u drugačijem pogledu na potencijalnu iskoristivost programskih rukotvorina. Dok razvojni okviri "nude" knjižnice generičkih programskih komponenti u uvjerenju da će biti korištene za široku lepezu aplikacija, linije proizvoda se sastoje od elemenata čija je potreba u dobroj mjeri potvrđena i predviđena u već solidno definiranom proizvodnom procesu. Može se lako zamisliti situacija nastanka proizvodnih linija iz postojeće programske potpore. Primjerice, ako neka organizacija razvije vlastiti programski proizvod i pojavi se potreba za drugim sličnim proizvodima, najbrži zadovoljavajući pristup je ponovna uporaba prethodno razvijenog programskog koda. Učestalijom primjenom takvog pristupa ona osnovna zamišljena struktura koda se s vremenom narušava te je smislena odluka o sačuvanju generičke strukture koda kroz definiciju nove zasebne proizvodne linije. Pritom se uzimaju opće funkcionalnosti s namjerom olakšavanja njihove ponovne uporabe u budućem razvoju novih programskih proizvoda. Korištenjem nove proizvodne linije u odnosu na radne okvire s generičkom jezgrom, skraćeno je vrijeme uključivanja novih razvojnih inženjera u razvojni proces i bitno je potpomognut proces ispitivanja buduće programske potpore.

U odnosu na razvojne radne okvire, kod primjenskih linija mogu se istaknuti neke značajne razlike:

- Razvojni radni okviri u većini slučajeva počivaju na objektno usmjerenoj paradigmi koja se proširuje da bi se ostvarila specifičnija primjena. Sâm programski kôd te strukture se pritom u pravilu ne mijenja. Primjenske linije proizvoda nisu toliko zasnovane na objektno usmjerenoj paradigmi, a kod nadogradnji radi ostvarivanja specifičnijih funkcionalnosti njihov programski kôd se mijenja i nadopunjuje.
- Primjenske linije proizvoda su više usmjerene na domenu primjene nego razvojni radni okviri, koji više predstavljaju tehničku potporu razvoju novih programskih proizvoda. Primjenske linije proizvoda u svojem samom temelju uključuju detalje domene primjene i implementacijske platforme. Tako npr. postoje i linije proizvoda za upravljanje radom knjižnice (s knjigama) temeljene 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.

- 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. Ponekad je moguće iskoristiti razvojni radni okvir za temelj nove primjenske linije. Razvoj daljnjih specifičnih ogranaka koji će činiti novu primjensku liniju proizvoda dolazi tek nakon izrade tog temeljnog programskog proizvoda.

Iako primjenske linije proizvoda u sebi uključuju programsku potporu 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. Naprimjer, mogu se razlikovati inačice koje podržavaju centralizirane i one koje podržavaju raspodijeljene procesne okoline.
- s obzirom na sveobuhvatnost funkcionalnosti – primjenski proizvod oblikuje se u odnosu na to koliki stupanj sveobuhvatnosti proizvoda se nudi. Tako npr. postoji *trial*, *lite*, *pro* i *enterprise* linije proizvoda, svaka sa svojom razinom sveobuhvatnosti podržane funkcionalnosti (i odgovarajućom cijenom).

Prema razgranatosti inačica proizvodnih linija, podrazumijevaju se određeni stupnjevi podesivosti 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 svode 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 činiti novi proizvod, a kod drugih o razvoju više generičkog sustava koji će se moći podešavati od strane korisnika ili dodijeljenog mu konzultanta. Kod ovih drugih se pretpostavlja da je znanje o specifičnosti korisničkih zahtjeva i radne okoline na prikladan način uključeno u obliku konfiguracijskih postavki koje su nadohvat vanjskom korisniku.

6.2.4. Gotovi programski proizvodi

Pod **gotovim programskim proizvodima** ili sustavima (engl. *commercial-of-the-shelf* - COTS) podrazumijevaju se oni koje nije potrebno (a nekad niti moguće) poznavati iznutra niti je moguće mijenjati programski kod da bi ga se koristilo u drugom programskom sustavu. Općenito, u ovu kategoriju pripada većina programske potpore na osobnim računaima i poslužiteljima. Često ta programska potpora ima mnoge funkcionalnosti i načine rada pa stoga i potencijale za korištenje u različitim razvojnim okolinama i kao sastavnica mnogih programskih proizvoda. 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. Omogućen je lakši fokus 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 gdje se korisnički zahtjevi moraju prilagođavati ne bi li se takva programska potpora iskoristila. 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 sam izbor najprikladnijeg programskog paketa za određenu namjenu može biti zahtjevan posao kojim se minimiziraju negativni efekti korištenja. Činjenica da je održavanje programske potpore odgovornost isporučitelja ima i svoju nepovoljnu stranu u određenoj ovisnosti o isporučitelju i vanjskim konzultantima. 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.

Među različitim modelima korištenja gotove programske potpore ističu se dva 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.

6.2.5. Oblikovni obrasci

Oblikovni obrasci (engl. *design patterns*) u programiranju predstavljaju dokazano dobar način ponovne uporabe znanja o čestom problemu i načinu rješavanja u okviru implementacije programskog proizvoda. Oni predstavljaju opće apstrakcije odnosa između konkretnih objekata koje se često primjenjuju na razini implementacije sustava. Obrazac daje opis problema i osnovni predložak rješenja primjenjiv u različitim situacijama na koje razvojni inženjer može naići. Predložak daje određeno rješenje problema, no ono nije kompletno za svaku situaciju, već se naknadno prilagođava specifičnosti svake implementacije.

Oblikovni obrasci predstavljaju vjerojatno najznačajnije unapređenje objektno usmjerenog oblikovanja. Iako su neki oblikovni obrasci iskoristivi i u proceduralnoj i u funkcijskoj paradigmi (a mnogi su i suvišni u funkcijskoj paradigmi), većina oblikovnih obrazaca je namijenjena objektno usmjerenoj paradigmi. Primjerice, oblikovni obrasci vrlo često koriste nasljeđivanje i polimorfizam, što su glavne značajke objektno usmjerenih jezika. Obrasci su, međutim, rijetko kad specifični za pojedine objektno usmjerene programske jezike te ih uglavnom ima smisla upotrijebiti u bilo kojem od njih.

Preduvjet za uporabu oblikovnog obrasca jest taj da razvojni inženjer ima razumijevanje da je problem moguće riješiti primjenom odgovarajućeg obrasca. Iz tog

razloga je važna edukacija u identificiranju i razumijevanju oblikovnih obrazaca (detaljnije o oblikovnim obrascima studenti na FER-u mogu naučiti na kolegiju Oblikovni obrasci u programiranju). Zainteresiranim studentima preporučuje se posjećivanje stranice *Design Patterns* [14].

Minimalni elementi oblikovnog obrasca su:

1. **naziv** (engl. *pattern name*) – razumljivo ime
2. **opis ili namjena** (engl. *description, intent*) – opis problema koji obrazac rješava
3. **rješenje** (engl. *solution*) – opis predloška koji može biti upotrebljen na različite načine
4. **posljedice** (engl. *consequences*) – rezultati i pogodnosti primjene obrasca uporabe.

Detaljnije, obrazac se može sastojati od sljedećih elemenata: naziva (engl. *pattern name*); opisa ili namjene (engl. *description, intent*); sinonima (engl. *also-known-as*) – liste sinonima za isti obrazac; motivacije – primjera problema i načina rješavanja uporabom obrasca; primjenjivosti – popisa slučajeva pogodnih za uporabu; strukture – skupa dijagrama razreda i objekata koji opisuju obrazac; elemenata (engl. *participants*) – opisa razreda i objekata te njihovih odgovornosti; međudjelovanja (engl. *collaborations*) – opisa kako se izvršavaju odgovornosti i posljedica (engl. *consequences*) – opisa uvjeta obrasca, prednosti i ustupaka.

E. Gamma je 1995. identificirao 23 oblikovna obrasca, a danas ih je poznato mnogo više, u svakom slučaju više od 50 oblikovnih obrazaca. Osnovna podjela oblikovnih obrazaca je na:

1. **stvaralačke** (engl. *creational*)
2. **strukturne** (engl. *structural*)
3. **ponašajne** (engl. *behavioral*).

Dodatno se često spominju i četvrti, **obrasci istovremenosti** (engl. *concurrency patterns*).

Namjena stvaralačkih oblikovnih obrazaca jest pomoć pri rješavanju problema vezanih uz načine stvaranja objekata iz razreda. Primjeri su obrasci *builder* (odvajanje konstrukcije objekata od njegovog predstavljanja), *abstract factory* (stvaranje primjerka nekoliko obitelji razreda), *prototype* (potpuno inicijalizirani primjerak za kopiranje ili kloniranje), *singleton* (razred koji smije imati samo jedan objekt u sustavu).

Strukturni obrasci usredotočeni su na odnose između razreda i odnose između objekata razreda. Primjeri su obrasci: *adapter* (podudara sučelja različitih razreda), *bridge* (odvaja sučelje objekta od njegove implementacije), *composite* (formira strukturu stabla za jednostavne i složene objekte), *decorator* (dinamički dodjeljuje odgovornosti objektima), *façade* (jedan razred predstavlja cijeli podsustav), *proxy* (objekt predstavlja drugi objekt).

Konačno, ponašajni obrasci usredotočeni su na probleme međudjelovanja razreda i objekata. Primjeri su obrasci *iterator* (slijedni pristup elementima kolekcije), *mediator* (definira pojednostavljenu komunikaciju među razredima), *observer* (daje način za javljanje promjene koja je nastala u određenim razredima), *strategy* (enkapsulira algoritam unutar razreda), *visitor* (definira novu operaciju razreda bez izmjena u samom razredu).

Stvaralački oblikovni obrazac *singleton* prikazan je, kao primjer, u nastavku.

Primjer 6.3.

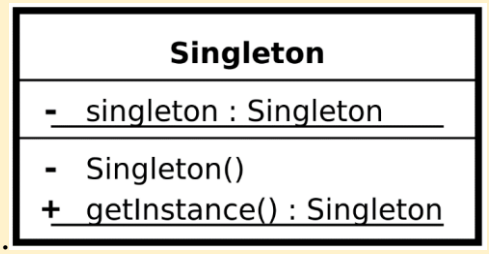
Naziv: Oblikovni obrazac *singleton*.

Opis: U nekim situacijama, potrebno je imati samo jedan primjerak određenog razreda u sustavu, koji je bolje rješenje nego korištenje globalnih varijabli programa. Oblikovni obrazac *singleton* treba riješiti probleme kao što su:

- Kako ograničiti da razred ima samo jedan primjerak?
- Kako razred može kontrolirati stvaranje svojih primjeraka?
- Kako jednostavno pristupiti jednom primjerku razreda?

Rješenje: rješenje problema prikazano je UML dijagramom razreda, a ostvaruje se tako da se:

- svi kontruktori razreda deklariraju kao privatni
- implementira statička metoda koja vraća referencu na jedan primjerak razreda
- primjerak obično pohranjuje kao privatna statička varijabla, a stvara se kada se varijabla inicijalizira, u nekom trenutku prije ili tijekom prvog poziva statičke metode, ovisno o implementaciji.



Posljedice: Na ovaj način, razred postaje odgovoran za kontrolu stvaranja vlastitog jednog primjerka. Privatni konstruktori onemogućuju da se primjerak razreda stvara izvana, a jedini način za pristup primjerku razreda je preko metode `getInstance()`.

6.3.Principi dobrog oblikovanja programske potpore

Za osiguranje kvalitetnog održavanja i pouzdanosti, oblikovana arhitektura mora biti stabilna. Dodavanje novih značajki u programski proizvod treba biti jednostavno i treba donijeti minimalne promjene u arhitekturi. Načela oblikovanja arhitekture

razmatraju osnovne probleme arhitekture iz perspektive smanjenja složenosti i povećanja prilagodljivosti (ili promjenjivosti) arhitekture.

U nastavku se razmatraju principi dobrog oblikovanja programske potpore koji su prilagođeni objektno usmjerenoj arhitekturi, i to prema dva izvora, prvi prema Lethbridgeu i Laganièreu [6], a drugi prema R. C. Martinu [13] i M. C. Feathersu [15]. Iako ovo nisu jedini razmatrani principi dobrog oblikovanja programske potpore u literaturi, ovi koje se ovdje navode pokrivaju dovoljno najbitnijih značajki kvalitetnog oblikovanja u objektno usmjerenoj arhitekturi.

6.3.1. Principi dobrog oblikovanja prema Lethbridgeu i Laganiereu [6]

Principi dobrog oblikovanja su:

1. podijeli pa vladaj (engl. *divide and conquer*)
2. povećaj koheziju (engl. *increase cohesion where possible*)
3. smanji međuovisnost (engl. *reduce coupling where possible*)
4. zadrži (višu) razinu apstrakcije (engl. *keep the level of abstraction as high as possible*)
5. povećaj ponovnu uporabivost (engl. *increase reusability where possible*),
6. povećaj uporabu postojećeg (engl. *reuse existing designs and code where possible*)
7. oblikuj za fleksibilnost (engl. *design for flexibility*)
8. planiraj zastaru (engl. *anticipate obsolescence*)
9. oblikuj za prenosivost (engl. *design for portability*)
10. oblikuj za ispitivanje (engl. *design for testability*)
11. oblikuj konzervativno (defenzivno) (engl. *design defensively*)
12. oblikuj prema ugovoru (engl. *design by contract*).

Princip podijeli pa vladaj

Princip **podijeli pa vladaj** pretpostavlja da je rad jednostavniji ako se podijeli na više manjih dijelova pri čemu je omogućeno da odvojeni timovi rade na manjim problemima. Ideja je organizirati posao tako da se razmatraju najprije manji dijelovi cjelokupnog sustava koji se na kraju postupno integriraju. Ovaj princip omogućuje specijalizaciju na način da neki razvojni inženjeri rade samo na dijelu sustava te se u tom dijelu specijaliziraju i povećavaju učinkovitost. Također, manje komponente

imaju veću razumljivost od složenijih komponenti. Time je olakšana zamjena dijelova bez opsežne intervencije u cijeli sustav. Posao je moguće podijeliti na više načina. Kod raspodijeljenih sustava, zasebno se mogu ostvarivati klijenti i poslužitelji. Dalje je moguće sustav podijeliti u podsustave, podsustave u pakete, pakete u razrede, a razrede u određene metode. Svaka razina omogućava sve veću specijalizaciju, a količina specijalizacije ovisi o potrebama projekta i mogućnostima razvojnog tima.

Princip povećanja kohezije

Kohezija programske potpore označava stupanj povezanosti elemenata unutar nekog modula. Podsustav ili modul ima veliku koheziju ako grupira međusobno povezane elemente, a sve ostalo stavlja izvan te grupe. Povećanje kohezije olakšava razumijevanje, robustnost, pouzdanost i uvođenje promjene u sustavu. Moduli s velikom kohezijom su iskoristiviji za ponovnu uporabu. Uobičajena je uporaba i miješanje različitih tipova kohezija. Tipovi povećanja kohezije koji se nastoje postići poredani su po važnosti:

1. **funkcijska kohezija** (engl. *functional cohesion*) – kôd koji obavlja pojedinu operaciju (funkciju, metodu) je grupiran, sve ostalo što nije izravno vezano uz tu funkcionalnost stavlja se izvan funkcije. Primjer slabe funkcijske kohezije modula bio bi da on mijenja bazu podatka, stvara datoteku i interagira s korisnikom.
2. **razinska kohezija** (engl. *layer cohesion*) – svi resursi za pristup skupu povezanih usluga na jednom mjestu, sve ostalo što nije s tim vezano nalazi se izvan modula. Razine formiraju hijerarhiju, pri čemu viša razina može pristupiti uslugama niže razine putem programskog sučelja (API), ali niža razina ne pristupa višoj. Na taj način postiže se odvajanje briga (engl. *separation of concerns*), jer svaki sloj ima svoj posao koji sakriva (enkapsulira) od ostalih slojeva.
3. **komunikacijska kohezija** (engl. *communicational cohesion*) – svi dijelovi koda koji pristupaju ili mijenjaju određene podatke su grupirani, sve ostalo stavlja se izvan modula. Primjerice, razred ima dobru komunikacijsku koheziju ako sadrži sve usluge neophodne za rad s određenim podacima i ne obavlja ništa drugo osim upravljanja tim podacima.
4. **sekvencijska kohezija** (engl. *sequential cohesion*) – grupira procedure u modulu tako da jedna procedura daje ulaz sljedećoj (procedure se izvode jedna ovisno o drugoj), sve ostale nevezane procedure stavljaju se izvan modula.
5. **proceduralna kohezija** (engl. *procedural cohesion*) – procedure koje se smisleno upotrebljavaju jedna nakon druge, ali za razliku od sekvencijske kohezije, ove ne moraju izmjenjivati podatke.

6. **vremenska kohezija** (engl. *temporal cohesion*) – operacije koje se obavljaju tijekom iste faze rada programa su grupirane, sve ostalo se stavlja izvan modula. Primjer bi bio podizanje sustava (inicijalizacija), koja bi se trebala odvijati u zasebnom modulu.
7. **kohezija pomoćnih programa** (engl. *utility cohesion*) – povezani pomoćni programi (engl. *utilities*) koji se logički ne mogu smjestiti u ostale grupe. Obično su to procedure ili razredi široko primijenjivi na različitite sustave, npr. java.lang.Math ili java.lang.Thread.

Princip smanjenja međuovisnosti

Za razliku od kohezije, međuovisnost se javlja onda kada moduli međusobno ovise jedni o drugima kako bi ispravno radili. U slučaju međuovisnosti modula, promjene na jednom mjestu zahtijevaju i promjene drugdje. Kod velike međuovisnosti teško je jasno raspoznati rad pojedinih komponenti. Tipovi međuovisnosti koji se nastoje izbjeći navedeni su prema važnosti:

1. **međuovisnost sadržaja** (engl. *content coupling*) – jedan modul ovisi o unutarnjem sadržaju (podacima) drugog modula. Najgori slučaj je kad jedan modul prikriveno mijenja interne podatke drugog modula. Međuovisnost sadržaja rješava se ispravnom enkapsulacijom podataka – unutarnje podatke postavi se kao privatne (engl. *private*), a pristup se omogućuje samo putem metoda *get* i *set*.
2. **opća međuovisnost** (engl. *common coupling*) – događa se kad dva ili više modula imaju pristup istim dijeljenim (globalnim) podacima. Najčešći slučaj je postojanje globalnih varijabli, čiju uporabu kod objektno usmjerenog oblikovanja treba izbjegavati ili ograničiti samo na definiranje podrazumijevanih vrijednosti.
3. **upravljačka međuovisnost** (engl. *control coupling*) – događa se kad jedan modul ima izravnu kontrolu nad radom drugog modula, primjerice uporabom zastavice pri pozivanju metode. Upravljačka međuovisnost treba se smanjiti uporabom polimorfnih operacija u objektno usmjerenom pristupu. Tijekom rada određuje se koju proceduru treba pozvati i kako se ona izvodi.
4. **međuovisnost u objektnom oblikovanju** (engl. *stamp coupling*) – javlja se kada složene podatkovne strukture (objekte) dijeli više modula, a svaki modul koristi samo jedan dio takve strukture. U tom slučaju, moduli mogu mijenjati podatke koji im ne koriste, što nije poželjno ponašanje. Ovo se može spriječiti korištenjem sučelja ili nekog jednostavnijeg razreda kao tipa podatka koji se prenosi među modulima.

5. **podatkovna međuovisnost** (engl. *data coupling*) – javlja se kada je tip metode argumenta primitivni tip podatka. Međuovisnost se povećava s većim brojem argumenata metode, jer sve metode koje ju koriste moraju prenijeti sve argumente. Smanjenje nepotrebne uporabe argumenata smanjuje stupanj međuovisnosti. Neophodan je kompromis podatkovne i međuovisnosti podatkovnih struktura u objektnom oblikovanju, jer povećanje međuovisnosti jednog tipa smanjuje drugi.
6. **povezivanje poziva procedura** (engl. *routine call coupling*) – javlja se kada procedura ili metoda u OO sustavu poziva drugu. Procedure time postaju povezane jer ovise o ponašanju druge. Ova međuovisnost je uvijek prisutna u sustavima, a česta je uporaba niza od dviju ili više procedura/metoda. Smanjenje povezivanja postiže se pisanjem jedinstvene procedure koja obuhvaća željeni niz procedura, gdje je to moguće i opravdano s obzirom na druge vrste međuovisnosti i kohezije.
7. **međuovisnost tipova** (engl. *type use coupling*) – događa se kada razred deklarira primjerak varijable ili lokalne varijable kao tipa drugog razreda. Posljedica takve međuovisnosti je potreba za promjenom svih korisnika neke definicije pri njezinoj promjeni. Izbjegavanje ovog tipa međuovisnosti je deklariranje tipa varijable kao najopćenitijeg razreda ili sučelja koje sadrži zahtijevane operacije.
8. **međuovisnost uključivanjem** (engl. *inclusion/import coupling*) – javlja se kada modul uključuje druge datoteke ili pakete. Komponenta koja uključuje drugu komponentu izložena je sadržaju komponenti koje je uključila. Ako uključena komponenta nešto promijeni, može doći do konflikta s komponentom koja ju uključuje. Iako je ovaj tip međuovisnosti teško izbjeći, treba voditi računa o tome da se koriste što poznatije druge komponente i one koje će se očekivano rijetko mijenjati.
9. **vanjska međuovisnost** (engl. *external coupling*) – predstavlja ovisnost modula o elementima izvan sustava (npr. OS, HW, ...). Potrebno je minimizirati broj mjesta na kojima se javlja takva povezanost. U objektnom pristupu, može se i oblikovati malo sučelje prema vanjskim komponentama (vidjeti npr. oblikovni obrazac *Façade*).

Princip zadržavanja (više) razine apstrakcije

Pri dobrom oblikovanju, potrebno je osigurati da oblikovanje omogući sakrivanje ili odgodu razmatranja detalja te na taj način smanji složenost. Dobra apstrakcija podrazumijeva skrivanje informacija (engl. *information hiding*) i predstavlja temelj

objektno usmjerenog oblikovanja. Apstrakcija omogućava razumijevanje suštine podsustava bez poznavanja nepotrebnih detalja.

Razredi su podatkovne apstrakcije koje sadrže proceduralne apstrakcije (metode). Razina apstrakcije se povećava definiranjem privatnih varijabli koje su nedostupne izvan samoga razreda. Osim toga, apstrakcija se poboljšava smanjivanjem broja javnih metoda. Nadrazredi/nasljeđivanja i sučelja (engl. *superclass, interface*) također povećavaju razinu apstrakcije, kao i smanjivanje broja argumenata kod metoda.

Princip povećanja ponovne uporabivosti

Ovaj princip traži da se oblikovanje različitih aspekata sustava provodi tako da može pridonijeti ponovnoj uporabi. Da bi se to postiglo, radi se poopćavanje oblikovanja u što većoj mjeri, što znači da se uporabljuju prethodni principi oblikovanja: povećaj koheziju, smanji međupovezanost i zadrži (višu) razinu apstrakcije. Ideja je izgraditi što neovisniji, zatvoreniji i kompaktniji modul koji stoga ima veću šansu da ga se ponovno iskoristi. Nadalje, oblikovanje takvog modula se maksimalno pojednostavljuje, a da bi se omogućila proširivost pri ponovnoj uporabi, u program se uvode **kopče** (engl. *hooks*), kao mjesta koje treba pozivati ili mijenjati da bi se omogućila nova funkcionalnost postojećeg programskog koda. Kopče se u objektno usmjerenim programskim jezicima najčešće ostvaruju kao metode koje imaju trivijalnu ili vrlo jednostavnu implementaciju te se omogućuje njihovo redefiniranje pri korištenju (nasljeđivanju) komponente koju se ponovno koristi.

Princip povećanja uporabe postojećeg (oblikovanja i koda)

Princip povećanja uporabe postojećeg je komplementaran principu povećanja ponovne uporabivosti. Cilj je što veća aktivna ponovna uporaba komponenti kako bi se smanjio trošak i povećala stabilnost programskog sustava. Ovaj princip naglašava korištenje prethodnih investicija, posebice onih modula koji su oblikovani u skladu s dobrim principima i iscrpno ispitani. Pritom je potrebno napomenuti da kopiranje i umetanje, kloniranje (engl. *cloning*) i slične „operacije“ s postojećim kodom se NE razmatraju kao ponovna uporaba. Korištenje programskih knjižnica, oblikovnih obrazaca, radnih okvira i gotovih programskih proizvoda, s druge strane, smatra se ponovnom uporabom.

Princip oblikovanja za fleksibilnost

Programska potpora treba biti fleksiblna, što znači da je predviđena za buduće moguće promjene. Stoga je potrebno unaprijed provesti pripremu da bi kod učinili fleksibilnim. To se postiže:

- smanjivanjem međuovisnosti i povećanjem kohezije (2. i 3. princip)
- stvaranjem apstrakcija (4. princip)
- ostavljanjem otvorenih opcija za eventualne modifikacije (5. princip)
- upotrebljavanjem postojećeg koda, a u slučaju pisanja novog koda teži se što lakšoj i većoj ponovnoj uporabi (6. princip)
- ne upotrebljavanjem izravnog umetanja podataka ili konfiguracija u izvorni programski kôd (engl. *hard code*).

Princip planiranja zastare

Princip planiranja zastare je sličan principu oblikovanja za fleksibilnost. Naime, ideja kod principa planiranja zastare jest unaprijed očekivati promjene u tehnologiji ili okolini i prevenirati ih tako da program može raditi ili biti jednostavno promijenjen. Razlika između ova dva principa je u tome što se kod planiranja zastare eksplicitno uzima u obzir činjenica da programska potpora stari zbog stalnih promjena u okolini, dok je kod oblikovanja za fleksibilnost veći fokus na izmjenama u samom proizvodu kao posljedici izmijenjenih korisničkih zahtjeva. Pri planiranju zastare prevencija se postiže:

- izbjegavanjem uporabe novih tehnologija ili njihovih novih isporuka (engl. *release*)
- izbjegavanjem knjižnica namijenjenih specifičnim okolinama
- izbjegavanjem nedokumentiranih ili rijetko upotrebljavanih dijelova knjižnica
- izbjegavanjem SW/HW bez izgleda za dugotrajnijom podrškom,
- uporabom tehnologija i jezika podržanih od više dobavljača.

Princip oblikovanja za prenosivost

Princip oblikovanja za prenosivost predviđa da je potrebno omogućiti rad programskom proizvodu na što većem broju različitih platformi. Prilikom razvoja, prilagodba radu na više platformi uglavnom znači da se izbjegavaju specifičnosti neke platforme, kao što su specifičnosti sklopovlja, operacijskog sustava i programskih knjižnica namijenjenih specifičnoj platformi. Prenosivost izvornog koda na različite operacijske sustave ostvaruje se prevođenjem (kompajliranjem) izvornog programskog koda u izvršni kod za ciljni operacijski sustav. Izvedivost takve prenosivosti ovisi o programskom jeziku i o korištenim dodatnim resursima (knjižnicama, radnim okvirima) u programskom kodu. U slučaju razvoja računski nezahtjevnih programa, za bolju prenosivost preporuča se korištenje programskih jezika koji su neovisni o platformi (kao što su Java i Python). Također, pri razvoju web

aplikacija, cijeli razvoj klijentske strane je platformski neovisan, jer se tu koriste samo *markup* i skriptni jezici koje tumači (interpretira) određeni web pretraživač.

Princip oblikovanja za ispitivanje

Princip oblikovanja za ispitivanje želi postići olakšano ispitivanje programske potpore. To znači da bi se program trebao oblikovati tako da se postigne što veći stupanj automatizacije ispitivanja. Primjerice, potrebno je omogućiti odvojeno pokretanje svih metoda uporabom vanjskih ispitnih programa. Tako naprimjer u Javi, u svakom razredu bi se stvorila metoda `main()` koja bi služila za jednostavnije ispitivanje ostalih metoda. Više oko automatizacije ispitivanja može se pročitati u poglavlju 7.3.5 i 7.6.

Princip konzervativnog (defenzivnog) oblikovanja

Defenzivno oblikovanje i defenzivno programiranje temelje se na tome da nije poželjno pretpostaviti kako će netko upotrebljavati oblikovanu komponentu. Umjesto toga, potrebno je proanalizirati i obraditi sve slučajeve u kojima se komponenta može neprikladno upotrijebiti i pripremiti se unaprijed za takve slučajeve. Priprema uključuje implementaciju provjere valjanosti ulaza (najčešće – parametara) u komponentu uz definirane pretpostavke korištenja te implementaciju provjere iznimaka raznih vrsta. Potrebno je napomenuti da, iako je ovaj princip poželjno primijeniti u određenoj mjeri, s druge strane, pretjerano obrambeno oblikovanje često dovodi do nepotrebnih provjera, što rezultira u gubitku vremena i računalnih resursa.

Princip oblikovanja prema ugovoru

Ovaj princip, koji je prvi put zaživio u funkcijskom programskom jeziku Eiffel 1986. godine, jest tehnika koja omogućava učinkovit i sustavni pristup formalnom, konzervativnom oblikovanju. Osnovna ideja je da postoji formalno precizno definirano sučelje kroz koje se pristupa određenoj komponenti. Sve javne metode tog sučelja imaju **ugovor** s bilo kojim pozivateljima, pri čemu je ugovor skup zahtjeva:

- **preduvjeta** (engl. *preconditions*) koje pozvana metoda mora ispuniti kada započinje izvođenje
- **završnih uvjeta** (engl. *postconditions*) koje pozvana metoda mora osigurati po završetku izvođenja
- **invarijanti** (engl. *invariants*) što označava ona svojstva koje pozvana metoda neće mijenjati i ona ograničenja kojih će se pridržavati tijekom izvođenja.

6.3.2. Principi dobrog oblikovanja prema R. C. Martinu [13] i M. C. Feathersu [15]

Martin je identificirao četiri glavna simptoma zašto oblikovana arhitektura sustava propada s vremenom. To su:

- rigidnost ili krutost (engl. *rigidity*)
- lomljivost (engl. *fragility*)
- nepokretnost (engl. *immobility*)
- viskoznost (engl. *viscosity*).

Rigidnost označava da je programska potpora nesklona promjeni. Često bi se moglo očekivati da se neki novi zahtjev na programsku potporu implementira vrlo brzo, ali u stvarnosti prođu tjedni i mjeseci dok se promjene ne implemetiraju do kraja. Lomljivost označava sklonost programske potpore da prestane raditi na puno mjesta svaki put kada se uvede neka promjena. Lomljivi programski proizvod postaje obično sve gori iz promjene u promjenu, lomeći se na sve više lokacija. Nepokretnost označava nemogućnost ponovne uporabe programske potpore, koja je često rezultat prevelike ovisnosti komponente o drugim komponentama. Na kraju, razvojni inženjer se radije odlučuje ponovno implementirati neku funkcionalnost, nego koristiti dijelove postojećih komponenti i prilagođavati ih za potrebe projekta. Viskoznost dolazi u dva oblika: viskoznost oblikovanja i viskoznost okoline. Viskoznost oblikovanja očituje se u tome da je teže mijenjati oblikovanje arhitekture nego prekršiti postojeću strukturu i ponuditi alternativno, direktno rješenje (engl. *hack*). Viskoznost okoline odnosi se na sporost i neučinkovitost razvojne okoline, npr. izgradnje izdanja (engl. *make*) programske potpore. U tom slučaju, često se koriste neoptimalni načini da se ta neučinkovitost smanji, koji idu nauštrb kvalitete.

Sprječavanje ovih simptoma propadanja programske potpore može se postići slijedeći principe s akronimom **SOLID**, koje je na temelju Martinovih razmatranja predložio Michael C. Feathers:

1. princip jedne odgovornosti (engl. *Single responsibility principle*)
2. princip otvoren-zatvoren (engl. *Open-closed priniciple*)
3. princip Liskovine supstitucije (engl. *Liskov substitution principle*)
4. princip segregacije sučelja (engl. *Interface segregation principle*)
5. princip inverzije ovisnosti (engl. *Dependency inversion principle*).

Princip jedne odgovornosti

Princip jedne odgovornosti glasi: „svaki razred kod objektno usmjerenog oblikovanja treba imati samo jednu odgovornost.“ Alternativno, ovaj se princip može formulirati i ovako: „svaki razred treba imati samo jedan razlog za mijenjati se.“ Odgovornost je, stoga, mogućnost promjene ili glavna (i jedina) funkcionalnost koju razred ima, a koju taj razred mora raditi dobro. Odgovornosti mogu biti različitih tipova, od čitanja / pisanja na tvrdi disk, preko komuniciranja s bazom podataka do izračunavanja određenog algoritma (primjerice strojnog učenja). Suština je da razred ne bi smio raditi ništa drugo bitno osim te jedne odgovornosti. Ako razred ima više odgovornosti, onda je vrlo izgledno da će te odgovornosti biti međusobno povezane, a to onda otežava izmjenu pojedinih odgovornosti. Slijeđenjem ovog principa bitno se olakšava izmjena razreda i ispitivanje, jer je lako utvrditi koji razred je odgovoran za koji zadatak.

Princip otvoren-zatvoren

Princip otvoren-zatvoren glasi: „svaki programski entitet (komponenta, modul, razred) treba biti otvoren za proširenja, ali zatvoren za izmjene.“ Martin smatra da je ovaj princip najvažniji. Ukratko, moduli se trebaju moći proširivati, ali proširenje treba napraviti tako da nema potrebe za izmjenama izvornog koda modula. Kako je modul očekivano dobro ispitan i s dobro definiranim sučeljem, sve modifikacije koje bi ga interno mijenjale ne smatraju se poželjnima. Glavni način za postizanje ovog principa je apstrakcija, što znači korištenje nasljeđivanja, izdvajanja sučelja, dinamičkog polimorfizma i generičkih tipova podataka. Metode koje želimo da se nadograđuju u proširenjima definiramo bilo samo potpisom u slučaju sučelja bilo trivijalnom ili generičkom implementacijom u slučaju nasljeđivanja.

Princip Liskovine supstitucije

Princip Liskovine supstitucije (nazvan prema prof. Barbari Liskov s MIT-ja) kaže: „svaki objekt u programu treba biti zamijenjiv s primjercima njegovih podtipova bez potrebe za izmjenama u programu.“ Na drugi način rečeno, neki vanjski entitet koji prema deklaraciji (u nekoj svojoj metodi) koristi objekt određenog baznog razreda treba normalno funkcionirati i u slučaju kada umjesto objekta baznog razreda dobije objekt bilo kojeg njegovog podrazreda. Liskovin princip supstitucije može se formulirati i u kontekstu oblikovanja prema ugovoru (vidjeti ranije poglavlje 6.3.1). Tako se može reći da ugovor koji nudi bazni razred treba biti ispoštovan i od strane njegovog podrazreda. U tom kontekstu, podrazred se može koristiti umjesto baznog razreda ako:

- njegovi preduvjeti nisu jači od onih baznog razreda za sve metode i

- njegovi završni uvjeti nisu slabiji od onih baznog razreda za sve metode.

Drugim riječima, metode podrazreda ne smiju očekivati više ili davati manje u odnosu na metode baznog razreda. Kršenje principa Liskovine supstitucije može imati teške posljedice za oblikovanje arhitekture. Takve pogreške je često teško otkriti, a popravci obično zahtjevaju velike izmjene u strukturi koda aplikacije. Tipičan primjer krivog oblikovanja je kada razred *Krug* nasljeđuje razred *Elipsu*. Detalje oko problematičnosti ovakvog nasljeđivanja studenti mogu sami proučiti u literaturi [13].

Princip segregacije sučelja

Princip segregacije sučelja kaže: „bolje je imati puno sučelja koja su specifična za klijente koji ih koriste nego jedno veliko sučelje opće namjene." Naime, ako jedan razred ima više klijenata, tada, umjesto učitavanja čitavog razreda sa svim metodama koje klijenti trebaju, stvaraju se specifična sučelja za svakog klijenta. Ta se sva sučelja višestruko nasljeđuju u razredu. Kod ovog principa nije ideja imati skupinu metoda za doslovno svakog klijenta, već se prilikom oblikovanja određuju tipične skupine klijenata. Metode se podijele po sučeljima koja će odgovarati pojedinim skupinama. Ako dva ili više tipova klijenata trebaju istu metodu, tada se ta metoda specificira u svim sučeljima gdje je to potrebno. Ako se ovaj princip ne slijedi, komponente i razredi sa svojim velikim sučeljima postaju manje korisni i prenosivi. S druge strane, ne treba se pretjerati s količinom različitih (i donekle sličnih) sučelja istoga razreda, već je potrebno naći odgovarajući kompromis između broja sučelja i broja metoda koje se nude u njima. Ovaj princip omogućuje lakše dodavanje nove funkcionalnosti jednostavnim uvođenjem novih sučelja, umjesto izmjena u postojećima, kada se pokaže da je takva dorada potrebna.

Princip inverzije ovisnosti

Princip inverzije ovisnosti glasi: „potrebno je ovisiti o apstrakcijama, a ne o konkretnim entitetima." Ova strategija naglašava uporabu sučelja, apstraktnih razreda i apstraktnih metoda umjesto konkretnih razreda i metoda gdjegod u kodu se međuovisnost pojavljuje. Moduli koji sadržavaju detaljne implementacije nisu više predmet tuđih ovisnosti, nego i oni sami ovise o apstrakcijama. Tako je ovisnost o konkretnim modulima zapravo invertirana. Razlog za ovisnost o apstraktnijim programskim entitetima je jasan: apstraktni entiteti se rjeđe mijenjaju od konkretnih entiteta. Također, apstraktni entiteti predstavljaju mjesta koja se mogu lakše proširivati bez izmjena u kodu tog entiteta (vidjeti i princip otvoren-zatvoren). Tako, ako neka komponenta ovisi o apstraktnijim komponentama, ima veće šanse za duljim korištenjem i manjom potrebom za čestim izmjenama.

6.4. Arhitekurni stilovi programske potpore

Arhitekturni stilovi (ili familije, arhitekturni obrasci, engl. *architectural style, architectural pattern*) su uočeni i određeni načini visoke apstraktne razine kako se često rješavaju određeni arhitekturni problemi programske potpore. U pojedinim modelima arhitekture mogu se prepoznati često upotrebljavane forme i oblici skupova srodnih arhitekturnih rješenja. U jednom programskom proizvodu mogu postojati kombinacije više arhitekturnih stilova (tzv. heterogena arhitektura). Svaki arhitekturni stil opisuje se:

- **rječnikom** – tipovima komponenata i konektora
- **topološkim ograničenjima** – koja moraju zadovoljiti svi članovi stila.

Primjeri često korištenih arhitekturnih stilova su:

- arhitektura klijent-poslužitelj (engl. *client-server architecture*)
- arhitekturna ravnopravnih sudionika (engl. *peer-to-peer architecture*)
- stil arhitekture zajedničkog računanja (engl. *collaborative computation*)
- arhitekturni stil model-pogled-nadzornik (engl. *Model-View-Controller,* MVC)
- stil višeslojne arhitekture (engl. *multi-layer architecture*)
- arhitektura posrednika (engl. *middleware*)
- uslužna arhitektura (engl. *service-oriented architecture*)
- arhitektura zasnovana na događajima (engl. *event-based architecture*)
- oblikovanje zasnovano na aspektima (engl. *aspect-oriented design*)
- arhitekturni stil protoka podataka (engl. *dataflow architecture*)
- arhitekturni stil repozitorija podataka (engl. *data repository*)
- arhitekturni stil virtualnih strojeva (engl. *virtual machine*)
- arhitektura programa u upravljanju procesa (engl. *process control*).

Ponekad se među arhitekturne stilove ubraja i objektno usmjereni arhitekturni stil. Umjesto toga, bolje je govoriti općenitije o objektno usmjerenoj paradigmi, budući da se objektno usmjereni način rješavanja problema proteže sve od organizacije zahtjeva, preko oblikovanja arhitekture (objektno usmjereno oblikovanje / modeliranje) do implementacije (objektno usmjereno programiranje), tako da je objektna usmjerenost širi pojam od arhitekturnog stila.

Svaki od navedenih stilova usmjerava organizaciju, elemente i kompoziciju arhitekture sustava. U nastavku su detaljnije opisani svi ovdje pobrojani stilovi arhitekture programske potpore. Objektno usmjereni arhitekturni stil niti modeliranje objektno usmjerene arhitekture ne razmatraju se u ovoj skripti.

6.4.1. Arhitektura klijent-poslužitelj

Arhitektura klijent-poslužitelj (engl. *client-server architecture*) temeljna je arhitektura današnjeg interneta i temeljni stil raspodijeljenih sustava (engl. *distributed system*). Kod ovog arhitekturnog stila, neka računala služe kao centri za razmjenu informacija i pružanje usluga. Takva računala nazivaju se **poslužitelji** (engl. *server*). Druga računala služe za pristup informacijama i dobivanje usluga te se ona nazivaju **klijenti** (engl. *client*). Uobičajeni je slučaj da više klijenata pristupa jednom poslužitelju, ali moguće je i da više klijenata pristupa više poslužitelja istovremeno, ovisno o tome kako je točno ostvarena određena usluga na strani poslužitelja.

Model arhitekture klijent-poslužitelj uključuje razmatranje funkcioniranja klijenta i poslužitelja prije, tijekom i po završetku zajedničke komunikacije. Razlikuju se dva pogleda na arhitekturu klijent-poslužitelj. U prvom pogledu, pod pojmom klijenta i poslužitelja podrazumijevaju se fizička računala koja komuniciraju. U drugom pogledu, i klijent i poslužitelj su programi na fizičkim računalima koja se povezuju radi zajedničke komunikacije. U nastavku podrazumijevat će se drugi pogled, osim ako nije drugačije navedeno.

Komunikacija između klijenta i poslužitelja pokreće se na sedmoj razini OSI modela, aplikacijskoj razini. Uobičajeni redoslijed komunikacije je sljedeći:

1. Administrator sustava pokreće aplikaciju poslužitelja na nekom računalu.
2. Pri pokretanju, poslužitelju se dodjeljuje ime (engl. *host name*), IP adresa i *port* na kojem poslužitelj sluša.
3. Poslužitelj počinje slušati pristižu li zahtjevi od klijenata.
4. Korisnik pokreće aplikaciju klijenta na nekom računalu (fizičko računalo je najčešće različito, ali može biti i isto kao i računalo gdje je poslužitelj).
5. Pri pokretanju, klijentskoj aplikaciji se dodjeljuju IP adresa i *port* putem kojih će klijent komunicirati s poslužiteljem.
6. Klijent u nekom trenutku šalje zahtjev za spajanjem (poruku) poslužitelju.

7. Po primitku zahtjeva od nekog klijenta, ako s tim klijentom već nije u komunikaciji, poslužitelj pokreće zasebni komunikacijski kanal za razgovor s njime i šalje mu odgovor da komunikacija može početi.
8. Poslužitelj i klijenti komuniciraju putem komunikacijskog kanala, izmjenjujući poruke (zahtjev-odgovor) i slijedeći odgovarajuće protokole aplikacijskog sloja (npr. http, https, ftp, smtp, itd.).
9. Komunikacija završava u tri slučaja: 1) klijent šalje poruku poslužitelju da želi završiti s komunikacijom (odspajanje), 2) poslužitelj šalje klijentu poruku da prekida slušanje daljnjih poruka ili 3) isteklo je zadano vrijeme za komunikaciju.

Komunikacija između klijenata i poslužitelja ima sljedeća ograničenja:

- broj klijenata koji istovremeno mogu komunicirati zasebnim komunikacijskim kanalima s poslužiteljem (npr. 15)
- duljinu reda čekanja prije uspostave komunikacije između klijenta i poslužitelja (npr. najviše 10 klijenata može čekati u redu na početak komunikacije)
- vrijeme trajanja razgovora – npr. ako nema primljene nove poruke u roku 10 s, poslužitelj ili klijent prekidaju komunikaciju.

Sva ova ograničenja najčešće postavlja administrator sustava na strani poslužitelja.

Ovisno o količini računanja i obrađivanja poruka koja se događa na klijentu i poslužitelju tijekom njihovog komuniciranja, postoje neformalno definirana dva odnosa između njih. Prvi odnos, koji se naziva **tanki klijent** (engl. *thin client*), podrazumijeva da aplikacija na strani klijenta ne obavlja zahtjevne proračune niti pohranu podataka, već služi samo za vizualni (grafički ili komandno-linijski) pristup funkcionalnostima poslužitelja. Tipični i danas najčešći primjer su web klijenti – web pretraživači (engl. *web browser*). Web pretraživač omogućuje korisniku pristup funkcionalnostima poslužitelja, dajući mu grafički prikaz web stranica i mogućnost interaktivnosti.

Drugi odnos naziva se **debeli klijent** (engl. *fat client, thick client*). Ovdje klijenti obavljaju većinu računanja i obrade podataka, dok poslužitelj najčešće služi samo za ostvarivanje interakcije između različitih klijenata, pohranu zajedničkih podataka i sl. Budući da se kod debelog klijenta većina proračuna obavlja na klijentskoj strani, mrežni promet je manji u odnosu na tankog klijenta. Tipični primjeri aplikacija debelog klijenta su računalne igre, posebice one koje su grafički zahtjevne i koje se instaliraju na lokalno računalo klijenta, a koje omogućuju zajedničko igranje u mreži.

Postoje mnoge prednosti arhitekture klijent-poslužitelj. Neke od njih su:

- posao se može raspodijeliti na više računala (strojeva)
- klijenti udaljeno pristupaju funkcionalnostima poslužitelja
- klijent i poslužitelj mogu se oblikovati odvojeno
- oba entiteta mogu biti jednostavnija
- svi podaci mogu se držati na jednom mjestu (na poslužitelju)
- obrnuto, podaci se mogu rasporediti na više udaljenih klijenata i poslužitelja
- poslužitelju može istodobno pristupiti više klijenata
- klijenti mogu ući u natjecanje za uslugu poslužitelja (a i obrnuto).

Rizici arhitekturnog stila klijent-poslužitelj su posebice značajni s obzirom na široku primjenu ovih sustava, a to su **sigurnost** i **potreba za adaptivnim održavanjem**. Sigurnost arhitekturnog stila klijent-poslužitelj je veliki problem bez savršenog rješenja. Za postizanje sigurnosti potrebno je koristiti enkripciju, vatrozidove, ovjeru korisnika i dr. Potreba za adaptivnim održavanjem označava da treba osigurati da sva programska potpora koja se koristi u ovoj arhitekturi bude kompatibilna prema unatrag, kompatibilna prema unaprijed i kompatibilna s različitim inačicama programa na klijentima i poslužitelju.

U praksi postoje mnoge vrste poslužitelja, u ovisnosti od toga koji komunikacijski protokol podržavaju (npr. HTTP, SMTP, IMAP, FTP, DNS i drugi). Ipak, u novije vrijeme, raspodijeljena arhitektura klijent-poslužitelj najviše se koristi na webu korištenjem HTTP protokola. U tom kontekstu, najčešće se razmatraju dva tipa poslužitelja:

- web poslužitelj (engl. *web server*)
- aplikacijski poslužitelj (engl. *application server*).

U nastavku se ukratko opisuju značajke i tipični primjeri ovih dvaju tipova poslužitelja.

6.4.2. Web poslužitelj

Web poslužitelj ima zadatak da pohranjuje, obrađuje i dostavlja klijentima web stranice. Komunikacija između web poslužitelja i web klijenta (najčešće, web pretraživača) odvija se korištenjem **HTTP i drugih sličnih protokola** (HTTPS, HTTP/2 i različitih nadogradnji tih protokola). U komunikaciji putem HTTP-a, web

poslužitelj tipično sluša nadolazeće zahtjeve klijenata na portu 80. Web stranice koje web poslužitelj isporučuje klijentu su HTML dokumenti, koji uključuju tekst, slike, stilove, skripte i drugi sadržaj. Osim za isporuku web stranica, web poslužitelji mogu služiti i za nadgledanje ili administriranje uređaja koji su spojeni na web. Takvi web poslužitelji su programi koji su ugrađeni u uređaj koji je spojen na web, npr. mrežni pisač, usmjernik, web kamera i sl.

Web poslužitelji u novije vrijeme često podržavaju skriptne jezike, kao što su PHP, ASP.NET i JSP. Ovi skriptni jezici omogućuju web poslužiteljima upravljanje interaktivnim sadržajima koji stižu od klijenata. Skriptni jezici na webu služe najčešće za generiranje HTML dokumenata dinamički, tijekom interakcije s korisnikom, umjesto da se korisniku dostavljaju statičke HTML stranice. Pomoću skriptnih jezika, korisniku se omogućuje slanje formi, dohvat i izmjena podataka u bazi podataka koja se nalazi na računalu poslužitelju te slanje ili dohvaćanje datoteka.

Neka web sjedišta koja imaju veliki promet mogu koristiti veći broj web poslužitelja koji isporučuju isti sadržaj velikom broju klijenata. Svaki web poslužitelj ima definirane granice opterećenosti, budući da u svakom trenutku može obrađivati samo ograničeni broj klijentskih konekcija (najčešće između 2 i 80.000, a u pravilu do 1000) po IP adresi i *portu* te može posluživati samo određeni maksimalni broj zahtjeva po sekundi, što ovisi o njegovim postavkama. Nakon što prijeđe svoje kapacitete, web poslužitelj postaje nedostupan, o čemu nastoji obavijestiti sve spojene klijente. Neki web poslužitelji u slučaju većeg opterećenja usmjeravaju svoj promet drugim web poslužiteljima koji nude istu uslugu.

Balansiranje opterećenja (engl. *load balancing*) između više poslužitelja koji nude istu uslugu važno je kod velikih web sjedišta i provodi se najčešće sa specijaliziranom programskom potporom koja se u komunikacijskom slijedu nalazi između klijenata i web poslužitelja. Takav programski proizvod, ponekad i na zasebnom sklopovlju, uz pomoć vatrozida i propitivanja opterećenja web poslužitelja pažljivo raspoređuje posao među njima.

Web poslužitelji mogu biti besplatni ili komercijalni programi. Od besplatnih web poslužitelja danas su najpoznatiji Apache HTTP Server (ili kraće: Apache), Microsoft IIS, nginx i Google GWS. Komercijalni web poslužitelji se najčešće sastoje od osnovnog web poslužitelja i od aplikacijskog poslužitelja.

6.4.3. Aplikacijski poslužitelj

Aplikacijski poslužitelj je program koji služi za dohvat resursa koji zahtijevaju više od osnovnih naredbi HTTP protokola, npr. rezultat izračuna kroz aplikaciju pisanu u

Javi ili nekom .NET jeziku (npr C#). Aplikacijski poslužitelj tako, uz HTTP protokol, podržava i druge protokole (npr. za pozive udaljenih procedura koriste se protokoli RPC/RMI). Ako web poslužitelj surađuje s aplikacijskim poslužiteljem, onda web poslužitelj služi samo za dohvat statičkog HTTP sadržaja, dok se sva dinamičnost prepušta aplikacijskom poslužitelju. Web poslužitelj može biti integriran u aplikacijski poslužitelj, a može se nalaziti i na zasebnom sklopovlju u odnosu na aplikacijski poslužitelj. Interakcija između aplikacijskog poslužitelja i web poslužitelja ostvaruje se putem API-ja u određenom programskom jeziku. Neki aplikacijski poslužitelji integriraju web poslužitelj i više usluga u okviru dane tehnologije određenog programskog jezika koja podržava dinamičnost web sadržaja. Primjerice, u kontekstu usluga, komercijalni aplikacijski poslužitelj Oracle Weblogic podržava cjelokupnu programsku infrastrukturu Java EE, što uključuje niz tehnologija: API za *servlete*, programske komponente *Enterprise Java Beans* (EJB), razmjenu poruka (JMS), transakcijske usluge (JTA), kontekste i ubacivanje ovisnosti (CDA) i drugo, kao i integraciju s web poslužiteljem. Besplatni aplikacijski poslužitelj Apache Tomcat, s druge strane, ne nudi izravnu integraciju s web poslužiteljem, ali se povezanost može konfigurirati tako da radi s bilo kojim web poslužiteljem.

6.4.4. Arhitektura ravnopravnih sudionika

Arhitektura ravnopravnih sudionika (engl. *peer-to-peer architecture*) naziv je za mrežnu arhitekturu kod koje postoji podjela poslova i podataka između računala s jednakim privilegijama i sličnim karakteristikama. Takva računala čine čvorove u mreži ravnopravnih sudionika. Svako ***peer*** računalo ostavlja dio svojih resursa (procesnu moć, prostor na tvrdom disku ili mrežnu propusnost) izravno dostupnu drugim *peerovima* u mreži, bez potrebe za centralnom koordinacijom *peerova* od strane poslužitelja. Za razliku od arhitekture klijent-poslužitelj, *peerovi* su i davatelji i korisnici mrežnih resursa.

Ideja u ovakvim sustavima je da se u mrežu uključuju različiti *peerovi* koji unose nove i jedinstvene resurse i mogućnosti kako bi ukupna mreža bila što korisnija svim sudionicima. Sustavi ravnopravnih sudionika koristili su se za različite primjene. Popularizaciju ovih sustava izvorno je ostvario sustav za dijeljenje datoteka (uglavnom glazbenih mp3 datoteka) Napster (1999. – 2002.). Sustavi ravnopravnih sudionika su kroz svoju kratku povijest uglavnom imali problema s kršenjem autorskih prava nad datotekama koje su se razmjenjivale, tako da su često ili prestajali s radom ili su postali legalni dućani za pojedina područja od interesa (npr. *online* dućani glazbenih albuma i pjesama).

6.4.5. Arhitekturni stil zajedničkog računanja

Zajedničko računanje (engl. *collaborative computation*) naziv je za mrežnu arhitekturu u kojoj više raspodijeljenih računala provode zajedničko računanje nad određenim podacima kako bi ostvarili isti zajednički cilj. Ovakvi sustavi primjer su obrnute (inverzne) arhitekture klijent-poslužitelj, u kojoj poslužitelj služi za slanje podataka i prikupljanje rezultata izračunavanja koje se provode na više klijenata. Klijenti se mogu smatrati debelim klijentima u kontekstu klasične arhitekture klijent-poslužitelj, budući da oni obavljaju cijelo računanje. Pritom, računanje se provodi na računalima klijentima nakon što korisnik lokalno instalira odgovarajuću aplikaciju koja će provoditi izračunavanje. Aplikacija koristi dio resursa računala klijenata, ovisno o postavkama.

Ovaj arhitekturni stil koristi se uglavnom u znanstvene svrhe, budući da veliki broj slabijih računala često ima veću ukupnu računalnu snagu nego što to ima jedno ili nekoliko jakih računala. Problemi koji se ovim stilom rješavaju zahtijevaju mnogo računalne snage. Najpoznatiji primjer primjene ove arhitekture je SETI@home, aplikacija koja traži dokaz o izvanzemaljskoj inteligenciji analizom radiosignala koji se prikupljaju s velikih teleskopa. Novija verzija aplikacije zajedničkog računanja naziva se BOINC. Osim za namjenu SETI programa, BOINC služi i za analizu drugih velikih znanstvenih problema (klimatoloških promjena, oblikovanja proteina i sl.).

6.4.6. Arhitekturni stil model-pogled-nadglednik

Arhitekturni stil model-pogled-nadglednik (engl. *Model-View-Controller*, kraće: **MVC**) danas je često korišten stil pri razvoju web i mobilnih aplikacija. Kod ovog stila, korisničko sučelje je odvojeno od ostatka sustava. Razinska kohezija elemenata postiže se kroz tri sloja, jednog na strani klijenta koji se naziva **pogled** (engl. *view*) i dva na poslužiteljskoj strani, koji se nazivaju **nadglednik** (engl. *controller*) i **model** (engl. *model*). Model, kao glavna komponenta ovog stila, sadrži razrede koji opisuju strukturu podataka domene primjene i koji sadrže pravila i poslovnu logiku. Model je često blisko povezan s bazom podataka aplikacije. Nadglednik sadrži razrede koji upravljaju i rukuju korisničkim zahtjevima prema modelu kao i odgovorima modela nazad prema pogledima. Pogledi sadrže razrede (ili komponente) koji služe za prikaz podataka modela i za interakciju s korisnikom kroz grafičko sučelje.

Postoji značajan broj radnih okvira za razvoj web i mobilnih aplikacija koji podržavaju arhitekurni obrazac MVC. Većina radnih okvira tako slijedi arhitekturu MVC zasnovanu na **guranju** (engl. *push*) ili **akcijama** (engl. *actions*). Kod ovih okvira,

akcije se koriste da bi se obavila zadana obrada podataka te se, nakon što je obrada završila, obrađeni podaci guraju putem nadglednika u sloj pogleda kako bi se prikazao rezultat. Primjeri razvojnih okvira koji podržavaju akcije su Spring, Django, Ruby on Rails, Symfony i drugi. Drugi način je arhitektura MVC zasnovana na **povlačenju** (engl. *pull*) ili **komponentama** (engl. *component-based*). Ovi radni okviri počinju od sloja pogleda, čije komponente povlače rezultate po potrebi s većeg broja nadglednika. Primjeri ovih radnih okvira su JavaServer Faces (JSF) i Apache Tapestry.

Tijekom razvoja radnih okvira za web i mobilne aplikacije, arhitekturni obrazac MVC mijenjao se i razrađivao na različite načine. Tako postoji nekoliko varijacija ovog osnovnog stila, primjerice:

- **model-pogled-pogled na model** (engl. *model-view-viewmodel*, kraće: **MVVM**) – izvorno Microsoftova implementacija stila MVC, kod koje sloj „pogled na model“ (engl. *viewmodel*) pretvara podatkovne objekte modela tako da se mogu lakše predstaviti pogledu i da je olakšano upravljanje njima.
- **model-pogled-predstavljač** (engl. *model-view-presenter*, kraće: **MVP**) – izvorno Taligentova implementacija stila MVC, kod koje je fokus na razvoju korisničkih sučelja i gdje je predstavljač međusloj koji preuzima podatke od modela, transformira ih i šalje pogledu na prikaz.
- **model-pogled-adapter** (engl. *model-view-adapter*, kraće: **MVA**) – kod ove varijacije, adapter ili posrednik odvaja model od pogleda tako da jedan nema znanja o drugome, već svu komunikaciju, uključujući preinake i transformacije podataka obavlja adapter, time omogućujući da se model i pogled razvijaju neovisno jedno o drugome.

Dodatno, s obzirom da je arhitekturni stil MVC na webu u određenoj mjeri povezan sa stilom višeslojne arhitekture, mogućih varijacija stila MVC ima mnogo, ovisno o tome kako se odgovornosti slojeva u višeslojnoj arhitekturi preslikavaju u komponente MVC stila. Nešto više o nekim varijacijama opisano je u sljedećem poglavlju.

6.4.7. Stil višeslojne arhitekture

Arhitekturni stil klijent-poslužitelj smatra se **dvoslojnim** (engl. *two-layer*) stilom, kod kojeg se jedan logički sloj implementacije programske potpore nalazi na klijentu, a drugi na poslužitelju. I klijent i poslužitelj mogu biti programi instalirani na istom računalu ili na dva odvojena računala. Pojam **razine** (engl. *tier*) odnosi se na fizičku

organizaciju programske potpore, dok se pojam **sloja** (engl. *layer*) odnosi na njezinu logičku organizaciju. Stoga bi dvoslojni stil arhitekture mogao biti **jednorazinski** (engl. *single-tier*) ako su oba programa, i klijent i poslužitelj, na istom računalu ili **dvorazinski** (engl. *two-tier*) ako su programi na zasebnim računalima.

Na obje strane, kod dvoslojnog modela, cjelokupna programska potpora je integrirana u samo jedan sloj. Ovakva (pre)jednostavna organizacija programske potpore koristila se, povijesno promatrano, dulje vrijeme, sve dok se nisu uvidjela neka ograničenja ovog stila. Primjerice, organizirati u istom logičkom sloju i prihvat poruka od klijenata i programsku logiku i komunikaciju s datotečnim sustavom i pristup bazi podataka i samu bazu podataka je u praksi vrlo teško postići, budući da su to sve odvojive i logički cjelovite komponente. Upravo iz tog razloga, sredinom 1990-ih uvodi se **troslojni stil arhitekture** (engl. *three-layer architecture*) kao prvi od **višeslojnih arhitekturnih stilova** (engl. *multi-layer architecture*), a koji se do danas zadržao u većini jednostavnijih web rješenja. Troslojni stil sastoji se od sljedećih slojeva:

1. **korisnički ili prezentacijski sloj** (engl. *presentation layer*) – sloj s korisničkim sučeljem
2. **sloj poslovne logike** (engl. *logic layer*) – sloj s implementacijom poslovnih procesa i izračuna
3. **podatkovni sloj** (engl. *data layer*) – sloj za pohranu podataka u bazu ili u datotečni sustav.

Sloj s korisničkim sučeljem kod web aplikacija uglavnom se sastoji od web poslužitelja koji isporučuje statički i u nekim slučajevima dinamički sadržaj (web stranice), a sadržaj tih stranica prikazuje web preglednik na klijentskoj strani. Sloj poslovne logike kod web aplikacija podržan je s web poslužiteljem ili s aplikacijskim poslužiteljem koji obrađuje i generira dinamički sadržaj. Pohrana podataka obično je organizirana na zasebnom poslužitelju koji sadržava sustav za upravljanje bazom podataka i/ili sustav za pohranu podataka u datotečni sustav.

Osim troslojnog stila, u praksi je moguće uvođenje i dodatnih slojeva. U tom slučaju, i klijentsku i poslužiteljsku stranu moguće je organizirati u više slojeva, pri čemu svaki sloj pruža uslugu nekom drugom sloju, a sakriva (enkapsulira) svoj skup usluga i svoje implementacijske detalje kao i implementacijske detalje sljedećeg sloja u nizu. Niz slojeva se proteže od sloja koji je najbliže korisničkom sučelju do sloja koji služi za trajnu pohranu podataka. Funkcionalne odgovornosti, koje čine najvišu razinu apstrakcije funkcionalnih zahtjeva, se raščlanjuju i pridjeljuju pojedinim slojevima. Organizacija višeslojne arhitekture ovisi izravno o funkcionalnim odgovornostima koje treba ostvariti, ali i o tehnološkom stogu koji određena tvrtka koristi. Primjerice,

za ostvarenje neke poslovne logike možda su potrebna tri odvojena sloja, a ne samo jedan, ovisno o složenosti same logike. Preslikavanje odgovornosti pridijeljenih određenom sloju na komponente u izvođenju nije jednoznačno, tako da je moguće istu višeslojnu arhitekturu fizički ostvariti na različitom broju računala, od jednog do više njih.

Jedan od suvremenih primjera organizacije višeslojne arhitekture je obrazac koji se koristi u okviru radnog okvira Spring Boot, a koji koristi principe programiranja:

- inverziju upravljanja (engl. *inversion of control*) i
- ubacivanje ovisnosti (engl. *dependency injection*).

Inverzija upravljanja funkcionira tako da korisnički napisani dijelovi (web) aplikacije dobiju upravljački tok i izvode se kada ih prozove neki generički radni okvir. To je inverzno u odnosu na tradicionalno programiranje u kojem korisnički kôd poziva određene knjižnice kako bi se mogao izvršavati. U slučaju inverzije upravljanja, radni okvir za web aplikacije (u ovom slučaju, Spring Boot) poziva korisnički kod.

Ubacivanje ovisnosti je najčešći način kako u praksi funkcionira inverzija upravljanja. Određeni korisnički kod (klijentski ili poslužiteljski) obavezno prima kao argumente metoda određene objekte, koji se u ovoj terminologiji zovu uslugama (engl. *service*, ne brkati s uslugama u uslužnoj arhitekturi), a koji su specificirani od strane radnog okvira kako bi sustav uspješno radio. Tako radni okvir, koji se u ovoj terminologiji naziva ubacivač ili injektor (engl. *injector*) ubacuje ovisnost o svojem određenom ugrađenom objektu u korisnički kod i definira sučelje (engl. *interface*) putem kojeg korisnički kôd pristupa usluzi.

Višeslojna arhitektura se tako, u slučaju korištenja Spring Boota, sastoji od slojeva:

1. sloj korisničke strane – implementiran primjerice u JavaScriptu, recimo koristeći knjižnicu React koja omogućuje prikaz korisničkog sučelja
2. sloj nadglednika (engl. *controller*) – povezuje korisničku stranu s poslužiteljskom stranom
3. sloj usluge (engl. *service*) – obavlja svu poslovnu logiku i potrebne izračune
4. sloj domene (engl. *domain*) – ima razrađeni model podataka domene
5. sloj za pristup podatcima (engl. *data access object*, kraće: DAO) – koji omogućuje spremanje i dohvat podataka iz određene baze podataka te razmjenu tih podataka sa slojem domene
6. sloj baze podataka – koji omogućuje stvarnu pohranu podataka u neku bazu, primjerice relacijsku bazu Postgre ili H2.

U usporedbi s arhitekturnim stilom MVC, višeslojna arhitektura može koristiti neke od dijelova stila MVC u većoj ili manjoj mjeri, primjerice sloj modela kod MVC-a odgovarao bi sloju domene kod Spring Boota, sloj nadglednika kod MVC-a sloju nadglednika kod Spring Boota, a sloj pogleda bio bi sloj korisničke strane. Druge arhitekturne varijacije sloja MVC mogu također odgovarati više ili manje konkretnoj višeslojnoj arhitekturi.

Prednosti višeslojne arhitekture su sljedeće:

- Oblikovanje se ostvaruje na temelju više razine apstrakcije.
- Podržano je jednostavno povećanje i poboljšanje (skalabilnost) sustava.
- Promjene u nekom od slojeva utječu samo eventualno na okolne slojeve.
- Timovi se mogu fokusirati na razvoj zasebnih slojeva.
- Podržana je ponovna uporaba.

Višeslojna arhitektura nema mnogo nedostataka, ali neki bi bili:

- Teško je odrediti optimalno preslikavanje odgovornosti na slojeve i na razine u izvođenju.
- Ponekad se izračunavanje i funkcionalnosti sustava ne mogu razbiti u slojeve.
- Razvoj često ovisi o većem broju različitih web tehnologija koje je potrebno savladati.

6.4.8. Arhitektura posrednika

Arhitektura posrednika ili **posrednička arhitektura** (engl. *middleware*, *broker*) je računalna arhitektura kod koje se uvodi tzv. **posrednički sloj**, koji se u mrežnoj usluzi nalazi između klijenta i poslužitelja i omogućuje da klijent i poslužitelj nesmetano komuniciraju. Općenito, posrednik je svaka programska potpora koja omogućava uzajamno djelovanje aplikacija bez potrebe za poznavanjem i kodiranjem operacija nužnih za implementacijske detalje usluge. Posrednik tako skriva detalje operacijskih sustava, komunikacijske mreže i druge specifičnosti implementacije razvojnim inženjerima koji oblikuju raspodijeljeni sustav. Time je omogućena veća usredotočnost na primjenski (aplikacijski) dio, što značajno olakšava oblikovanje i razvoj sustava.

Posrednička arhitektura razvijala se tijekom povijesti još od 1960-ih. U ranijim godinama, a pogotovo prije pojave raširenog interneta (do sredine 1990-ih),

posrednici su bili takvi računalni programi koji su skrivali detalje jezgre operacijskog sustava od aplikacija koje koriste korisnici. Najčešće, tu se radilo o nizu programskih knjižnica koje su olakšavale grafičko programiranje, obradu zvuka i slične primjene. Kasnije, u doba interneta, posrednicima su se smatrali svi oni programi (jednostavniji ili složeniji) koji pokrivaju značenje znaka „-“ kod arhitekturnih stilova klijent-poslužitelj i *peer-to-peer*. Konkretno, tu se radi o programskoj potpori koja se u svakom slučaju nalazi iznad četvrtog sloja OSI modela (TCP-a). To uključuje programsku potporu kao što su web poslužitelji, aplikacijski poslužitelji, sustavi za upravljanje sadržajima (engl. *content management system*) i drugi, ali ne i programsku potporu konkretne web aplikacije.

Raspodijeljene posredničke arhitekture mogu se podijeliti u tri vrste:

- transakcijski usmjerene (engl. *data integration*) – služe za olakšanu komunikaciju s bazama podataka, npr. ODBC, JDBC ili, na višoj razini apstrakcije: Hibernate, ADO.NET.
- zasnovane na porukama (engl. *message-oriented-middleware*, kraće: MOM) – služe za pouzdanu, asinkronu komunikaciju porukama između različitih usluga na internetu, npr. JMS, NMS.
- objektno usmjerena posrednička komunikacija (engl. *object request broker*, kraće: ORB) – služe za sinkronu komunikaciju između raspodijeljenih objekata kod objektno usmjerenih programskih jezika, npr. CORBA, DCOM, EJB RMI.

Komunikacija u raspodijeljenoj posredničkoj arhitekturi funkcionira tako da pošiljatelj poruke pokreće transakciju ili zove udaljenu proceduru pri čemu ne poznaje detalje izvođenja takve akcije na drugoj strani komunikacijskog kanala. Zadatak posrednika je da ostvari funkcionalnost poziva udaljenog resursa kao da je taj resurs prisutan lokalno na računalu pošiljatelja. U praksi, poziv se pokreće tako da pošiljatelj pozove metodu određenog API-ja posrednika. Najčešće, poziv takve virtualne metode, koja se čini lokalnom, iziskuje poziv stvarne metode na nekom udaljenom računalnom resursu. Konkretna izvedba udaljene metode i odgovor koji se dobiva ovisi o vrsti raspodijeljene posredničke arhitekture i implementaciji odgovarajućeg posrednika koji se koristi.

Pri raspodijeljenoj komunikaciji, u praksi se razlikuju implementacije posrednika:

- na niskoj razini API-ja, kod koje je naglasak na paralelizaciji raspodijeljene arhitekture i brzini odziva, primjer protokola je sučelje za prijenos poruka (engl. *Message Passing Interface*, MPI), ostvareno u punoj kompatibilnosti za jezike C/C++ i Fortran.

- na visokoj razini API-ja, kod koje je naglasak na kompatibilnosti između različitih implementacijskih platformi kod raspodijeljenih sustava i gdje nije potrebno poznavati implementacijske detalje udaljenog računala, primjer protokola je poziv udaljenih procedura (engl. *remote procedure call*, RPC). Novija implementacija protokola RPC za objektno usmjerenu posredničku komunikaciju naziva se pozivanje udaljenih metoda (engl. *Remote Method Invocation*, RMI).

Poziv udaljenih procedura podržava interoperabilnost u heterogenim sustavima kod kojih, u općem slučaju, implementacijski programski jezik na udaljenim lokacijama kao i organizacija memorije u operacijskom sustavu (npr. *little endian*, *big endian*) nisu isti. RPC se ostvaruje ispravnom implementacijom i pravilnom konfiguracijom posrednika koji se instalira i kod pošiljatelja (najčešće klijenta) i kod primatelja (najčešće poslužitelja). Sučelje u smislu podržanih metoda između pošiljatelja i primatelja najčešće se specificira s određenim apstraktnim jezikom za opis sučelja (engl. *interface description language*, IDL), a ostvaruje se u odgovarajućim programskim jezicima s obje strane.

Klijent pri pozivu udaljene procedure/metode koristi gotove knjižnice udaljenih komponenti te poziva metodu predajući joj lokalne argumente programa. Argumenti i naziv metode se na odgovarajući način pakiraju (takav paket koda se naziva *client stub*) i spremaju za slanje mrežom na strani pošiljatelja najčešće binarnim kodiranjem toga paketa (to se naziva *marshalling*); na odredišnoj strani se otpakiraju i pripreme za izvođenje (takav kôd se naziva *server skeleton*) te se potom izvede metoda na odredištu. Povrat poruke, kao rezultat izvršavanje metode, ide u obrnutom smjeru, od odredišta do pošiljatelja, sve dok pošiljateljeva metoda ne primi odgovor.

Prednosti arhitekture posrednika su:

- omogućena je transparentnost lokacija
- izmjenjivost i proširivost komponenti koji čine posrednika
- prenosivost na različite platforme
- interoperabilnost različitih sustava posrednika
- ponovna uporaba.

Nedostatci su:

- smanjena učinkovitost, budući da posrednik, ovisno o složenosti heterogene implementacije iziskuje dodatno vrijeme za provođenje komunikacije

- veća osjetljivost na pogreške, jer su posrednici složeni programski sustavi s puno konfiguracije.

6.4.9. Uslužna arhitektura

Uslužna arhitektura (engl. *service-oriented architecture*, SOA) organizira programsku potporu kao kolekciju usluga koje međusobno komuniciraju uporabom dobro definiranih sučelja putem mrežnih protokola. Uslužna arhitektura slična je posredničkoj arhitekturi u mnogim svojim značajkama. Glavna razlika je u konceptu **usluge**. Svaka usluga ima dobro definiranu funkciju, samodostatna je i njezino unutarnje funkcioniranje ne ovisi o drugim uslugama ili stanju njenog okruženja. Iako se kod posredničke arhitekture može govoriti o usluzi određene odredišne komponente, fokus tog arhitekturnog stila nije na samoj usluzi, već na raspodijeljenoj komunikaciji.

Uslužna arhitektura temelji se na trima entitetima: zahtjevatelju ili klijentu usluge, pružatelju usluge i posredniku (registru, repozitoriju) usluge. Klijent putem različitih operacija pronalazi podatke o usluzi u registru usluga i zatim se povezuje s pružateljem usluge kako bi pozvao neku od njegovih usluga. Klijent može koristiti više usluga ako ih pružatelj nudi. Pružatelj usluge stvara uslugu i oglašava sve potrebne informacije o njoj u registru usluga. Posrednik (registar) služi za otkrivanje informacije o postojanju usluge potencijalnom klijentu. Javni registri omogućuju pristup usluzi bilo kojem klijentu, dok privatni registri omogućuju pristup usluzi samo određenim klijentima.

U internetskom okruženju, najčešće se koristi pojam **web usluge** (engl. *web service*). Web usluga podrazumijeva web aplikaciju koja je izgrađena i postavljena u web okruženje (na aplikacijski ili web poslužitelj) u kojoj je omogućena njezina povezanost s drugim web aplikacijama kako bi određenom klijentu pružila potrebnu uslugu. Jezici za opis web usluga i njihovu komunikaciju su uglavnom normirani. Uobičajeno, za komunikaciju između web usluga koristi se protokol **SOAP** (engl. *Single Object Access Protocol*), a umjesto njega, može se koristiti i arhitekturni stil **REST** (engl. *representational state transfer*). Za opis web usluge koristi se jezik **WSDL** (engl. *web service description language*).

Protokol SOAP je komunikacijski protokol neovisan o platformi koji je zasnovan na **XML**-u i namijenjen je komunikaciji web aplikacija. Jedna SOAP poruka sastoji se od omotnice (engl. *envelope*), koja definira strukturu poruke i način kako se poruka treba obraditi. Unutar omotnice, definirani su zaglavlje i tijelo SOAP poruke. U zaglavlju se nalazi detaljnije tumačenje poziva, kao i čvorovi (nazivi poslužitelja)

kojima poruka prolazi do odredišta. Tijelo poruke sadrži detalje poziva, kao što su nazivi metoda koje se pozivaju. SOAP poruku, koja je jedna XML datoteka, može poslati bilo koji protokol aplikacijskog sloja (HTTP, SMTP, FTP i sl.). Ipak, u većini slučajeva SOAP se koristi u kombinaciji s HTTP-om. Za binarne podatke koje treba poslati (npr. slike), definirana su proširenja koja omogućuju da se uz XML poruku pošalju i po potrebi više rascjepkanih poruka koje čine određeni resurs u binarnom obliku.

REST je arhitekturni stil za specifikaciju ograničenja u komunikaciji komponenata koje čine web usluge. Potrebno je naglasiti da, za razliku od SOAP-a, REST nije norma, već REST samo koristi druge norme kako bi omogućio učinkovitu komunikaciju između web aplikacija. Tako REST koristi norme HTTP, XML, JSON i URI. Bitne značajke stila REST, koje određenu web uslugu čine da podržava stil REST (i da bude tzv. *RESTful* web usluga) su:

- Koristi se arhitekturni stil klijent-poslužitelj, s jasno definiranim sučeljem između njih.
- Ne pamti se stanje klijenta na poslužitelju između dva zahtjeva (sve potrebno je zapakirano u zahtjevu).
- Odgovori prema klijentu se trebaju moći čuvati u priručnoj memoriji dok se ne isporuče.
- Potrebno je podržati slojevitost usluge, o kojoj klijent ne treba voditi računa (može biti više međuslojeva) od klijenta do konačnog poslužitelja.
- Postoji uniformnost sučelja kod web usluga, što osigurava da se resursi interno razvijaju neovisno o odgovoru koji pružaju kroz svoje sučelje.

Pri usporedbi SOAP-a i REST-a, može se ustanoviti da je REST jednostavniji od SOAP-a, koristi se direktno s HTTP-om ili drugim protokolima aplikacijskog sloja, bez dodatnih podataka (omotnice) karakterističnih za SOAP. Kada se zatraži neki URI, vraća se izravna reprezentacija resursa, nad kojom se mogu izvršiti HTTP operacije POST, GET, PUT, ili DELETE. Podaci o resursu ne moraju (i često nisu) predstavljeni XML formatom. Česti format za prikaz objekata koristeći stil REST je JavaScript Object Notation (**JSON**). U odnosu na XML, JSON je nešto sažetiji tekstualni prikaz podataka.

WSDL je normirani jezik za opis web usluge koja se nudi na nekom računalu u mreži. Po svojem obliku, WSDL je XML dokument pisan prema WSDL specifikaciji. On opisuje:

- što usluga radi – operacije (metode) koje može pružiti

- način pristupa web usluzi – format podataka i opis protokola koji usluga koristi za komunikaciju (npr. SOAP)
- smještaj/lokaciju usluge – URL adresa.

WSDL dokument može poslužiti za automatsko generiranje koda web usluge, a vrijedi i obrat: WSDL specifikacija može se generirati na temelju koda web usluge. Podržanost toga ovisi o programskom jeziku i značajkama programa generatora.

U novije vrijeme, u 2010-ima, umjesto strogo definiranih web usluga koje su karakterističnije za velika poduzeća, počele su se pojavljivati manje i neovisne web usluge koje međusobno surađuju kako bi pružile određenu uslugu i koje obično koriste manje ili srednje tvrtke. Ove web usluge nove generacije nazivaju se **mikrouslugama** (engl. *microservices*). Mikrousluge su malene web usluge, koje:

- komuniciraju porukama putem laganih protokola (ne koriste normu SOAP nego arhitekturni stil REST)
- imaju ograničeni kontekst korištenja
- autonomno se razvijaju i neovisno se pokreću u mreži
- samodostatne su i nude poslovnu funkcionalnost putem jasnog sučelja.

Mikrousluge se mogu koristiti u okviru različitih širih poslovnih procesa, primjerice u vidu ponude programske potpore kao usluge (engl. *software as a service*, SaaS) u okviru računarstva u oblaku (engl. *cloud computing*).

6.4.10. Arhitektura zasnovana na događajima

Arhitektura zasnovana na događajima (engl. *event-based architecture, implicit invocation*) je takva arhitektura kod koje je programska potpora organizirana tako da postoje komponente koje stvaraju događaje i komponente koje reagiraju na određene događaje. Temeljne značajke ove arhitekture su:

- Komponente se međusobno ne pozivaju eksplicitno.
- Neke komponente generiraju signale = događaje.
- Neke komponente su zainteresirane za pojedine događaje te se prijavljuju na strukturu za povezivanje komponenata.
- Model izvođenja je takav da se događaj javno objavljuje te se pozivaju prijavljene procedure za obradu tog događaja.

- Komponente koje objavljuju događaj nemaju informaciju o tome koje će sve komponente reagirati i kako na događaj.
- Rukovanje događajima je asinkrono, nema jamstva da će se neki događaj obraditi prije nekog drugog događaja (nedeterminizam).

Važna značajka je implicitno pozivanje procedure/metode, što znači da jedna metoda ne zove izravno drugu, već se druga metoda poziva (od strane sustava/strukture za povezivanje komponenti) ako je prijavljena za određeni događaj.

Prednosti arhitekturnog stila zasnovanog na događajima su:

- omogućuje razdvajanje i autonomiju komponenata
- podupire evoluciju, uvođenjih novih komponenti i ponovno korištenje.

Nedostatci su:

- komponente koje objavljuju događaje nemaju jamstva da će dobiti odziv
- komponente koje objavljuju događaje nemaju utjecaja na redoslijed odziva
- apstrakcija događaja ne vodi prirodno na postupak razmjene podataka
- teško je rasuđivanje o ponašanju komponenata koje objavljuju događaje i pridruženim komponentama koje su registrirane za te događaje (zbog nedeterminizma odziva).

Primjeri sustava koji koriste arhitekturu zasnovanu na događajima su oni koji obuhvaćaju komunikaciju s korisnikom putem sučelja (npr. klik mišem je događaj) te web aplikacije, primjerice one koje koriste obrazac MVC. Događajem kod obrasca MVC bi se smatralo slanje poruke odgovarajućeg potpisa od klijenta (na temelju akcija u pogledu) za koji postoji odgovarajuća metoda na poslužitelju (u nadgledniku) koja će tu poruku prihvatiti i usmjeriti na odgovarajući model. Općenito, arhitektura zasnovana na događajima rijetko se kada koristi sama za sebe, već je gotovo uvijek dio šireg i složenijeg sustava ili arhitekturnog stila.

6.4.11. Oblikovanje arhitekture zasnovano na aspektima

Kao jedna zasebna grana programskog inženjerstva predstavlja se i aspektno-usmjereni razvoj programske potpore (engl. *aspect-oriented software development*), čiji važni dio čini **oblikovanje arhitekture zasnovano na aspektima** (engl. *aspect-oriented design*). Umjesto objekata kao središnje razine apstrakcije kod objektno usmjerenog razvoja, aspektno usmjereni razvoj razmatra aspekte korištenja

sustava. U programskom smislu, aspekti su značajke programa koje se protežu kroz niz povezanih komponenti, a koje nisu kritične za funkcionalnost izvođenja tog programa. Aspekti se, slično kao i objekti, temelje na razdvajanju područja odgovornosti (engl. *separation of concerns*), tako da se aspekti oblikuju neovisno jedan od drugoga i skrivaju svoju unutarnju funkcionalnost. Aspektno oblikovanje arhitekture ima za zadatak modelirati nefunkcionalne zahtjeve na sustav i značajke kvalitete sustava. Iako aspekti nisu vezani uz specifičnu funkcionalnost, oni kategoriziraju funkcionalnost u cjeline tako da provode određene nefunkcionalne zahtjeve. Kako se područja odgovornosti najčešće protežu u dubinu kroz arhitekturu sustava zasnovanoj na objektima, blisko povezujući različite objekte, organizacija arhitekture putem aspekata ostvaruje reorganizaciju i smanjenje količine koda, koji bi inače za ostvarivanje pojedinog nefunkcionalnog aspekta sustava bio raspršen i isprepleten.

U praktičnom smislu, aspekti se programiraju specifičnim programskim jezikom za definiranje svojeg izvođenja pa se tako za tehnologije u Javi koristi jezik AspectJ, koji predstavlja nadjezik jezika Java, uz dodatak aspekata kao modula koda. Oblikovanje zasnovano na aspektima najčešće se koristi kod velikih i složenih sustava, kod kojih je važno podržati različite aspekte programskog proizvoda. Tipični primjeri aspekata su vođenje dnevnika događaja (engl. *logging*), ovjera korisnika tijekom provođenja određene akcije (engl. *authorization*), slanje obavijesti (notifikacija) (engl. *notifications*) i slično.

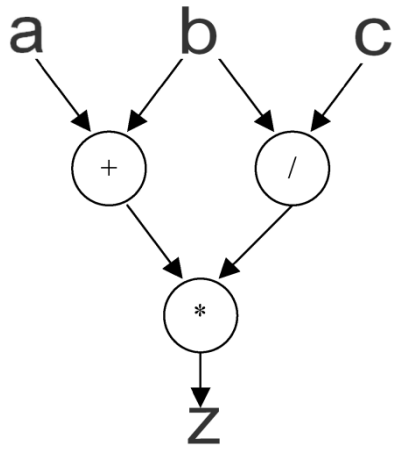
6.4.12. Arhitekturni stil protoka podataka

Arhitekturni stil protoka podataka (engl. *dataflow architecture*) je takva arhitektura kod koje je glavni cilj organizacija obrade podataka kroz kolekciju modula za izračunavanje. Naime, za razliku od upravljačkog toka programa, kod kojeg je dominantno pitanje kako se središte upravljanja, instrukciju po instrukciju, pomiče kroz program, kod stila protoka podataka dominantno se razmatraju operacije koje se obavljaju nad podacima. Izvršavanje „instrukcija“ (u stvarnosti: modula ili komponenti izračunavanja) u stilu protoka podataka potpuno je određeno s dostupnim podacima koji predstavljaju ulazne argumente za tu instrukciju. Redoslijed izvršavanja instrukcija nije određen programskim brojilom (engl. *program counter*, PC), nego je nedeterministična i ovisan samo o načinu kako su instrukcije međusobno povezane i koji podaci su dostupni. S dostupnosti podataka aktivira se daljnje upravljanje.

Arhitektura protoka podataka temelji se na mreži aktora (procesnih modula) koji razmjenjuju podatke i paralelno obavljaju obradu. Osnovne prednosti ove arhitekture su:

- razdvajanje procesnih modula i
- paralelna obrada.

Aktori predstavljaju funkcijske jedinice (procesne module) arhitekture protoka podataka i često su fino granulirani (engl. *fine-grained*), što znači da obavljaju jednostavne operacije uz potrebu za čestom komunikacijom rezultata. Aktori su uglavnom organizirani tako da podržavaju paralelni rad. Na slici 6.5 prikazan je primjer mreže aktora u obliku grafa toka podataka (engl. *dataflow graph*).



Slika 6.4 Mreža aktora (procesnih modula: „+“, „/“ i „*“) u obliku grafa toka podataka s ulaznim (a, b, c) i izlaznim (z) podacima.

U praksi, razlikujemo sklopovske i programske implementacije arhitekturnog stila protoka podataka. Sklopovske arhitekture protoka podataka bile su popularne u 1970-ima i 1980-ima (npr. *Manchester Dataflow Machine* 1976. – 1983.). Tu su razvijana specijalna programabilna računala sa sklopovljem optimiranim za podacima upravljano izračunavanje. Razvijani su:

1. sinkroni (statički) protok podataka – broj ulaznih podataka koji pristižu poznat je unaprijed
2. Booleov asinkroni protok podataka – broj ulaznih podataka koji se obrađuju nije posve određen unaprijed, ali je kontroliran upravljanjem (selekcijskim ulazima)

3. potpuno asinkroni protok podataka – broj ulaznih podataka koji se obrađuju nije poznat prije izvođenja, već se doznaje programski, kroz interakciju s okolinom za vrijeme izvođenja.

U praksi, pokazalo se da općenite, potpuno asinkrone (nedeterminističke) sklopovske arhitekture protoka podataka ne funkcioniraju zato što:

- nije moguće učinkovito objavljivanje rezultata izračunavanja u masivno paralelnom sustavu
- dinamičko upravljanje (i sinkronizacija) procesnih elementata je presporo u masivno paralelnom sustavu
- nije moguća izgradnja dovoljno velike brze asocijativne memorije (engl. *content-addressable memory*) da zapamti sve potrebne podatke za rad paralelnog sustava.

Bez obzira na praktičnu neupotrebljivost ovakvih sustava, specijalni slučajevi sklopovske implementacije protoka podataka postoje i razvijali su sve do danas, a to su primjerice:

- **izvođenje izvan poretka** (engl. *out-of-order execution*) – oblik ograničene arhitekture protoka podataka, u procesorima (CPU), gdje unutar prozora od *n* instrukcija redoslijed izvođenja ovisi o podacima. Izvođenje izvan poretka ubrzava izvođenja, instrukcije se izvode čim su podaci dostupni. Starije instrukcije mogu se izvesti i kasnije od novijih, a na kraju se mora poredati redoslijed rezultata kako bi se zadržala konzistentnost. Ovo zahtijeva skuplje sklopovlje, a specijalizirane jedinice zauzimaju više mjesta na čipu.
- **grafičko sklopovlje** (engl. *graphics processing unit*, GPU) – oblik ograničene arhitekture protoka podataka na zasebnom sklopu kod koje postoji jasna organiziranost podjedinica za paralelnu obradu velikih količina podataka transformacijama sadržaja u vlastitoj memoriji. GPU je vrlo pogodan za brzo izvođenje jednostavnih algoritama koji primaju puno podataka, a koji se mogu paralelizirati (nemaju iterativne ili rekurzivne ovisnosti o prethodnom koraku računanja).
- **specijalizirano sklopovlje za slijednu obradu podataka** – primjer je sklopovski sustav za prijenos i obradu zvuka, koji u nizu koraka primjenjuje različite filtere i transformacije podataka kako bi na kraju uklonio neželjene šumove.

Programsko ostvarenje arhitekture protoka podataka je danas široko rasprostranjeno u praksi. Pritom se za definiciju arhitekture koriste specifični jezici za opis protoka

podataka (tekstualni, grafički ili kombinirani) koji obično podrazumijevaju paralelizam (engl. *implicit parallelism*) u izvedbi. Jezici za opis protoka podataka su često funkcijski po tipu, a neki od poznatijih primjera su:

- Id (Irvine dataflow language) (1975.)
- LUCID (1976.)
- SISAL (Streams and Iteration in Single Assignment Language) (1986.)
- HDL (Verilog (1984.), VHDL (1987.), System Verilog (2002.))
- LabVIEW, G jezik (1986.)
- Simulink – Matlab (1984.)
- Microsoft Visual Programming Language (MVPL) (2006.)
- RapidMiner (2006.), Knime (2006.)
- Ptask (Microsoft) (2011.)
- TensorFlow (Google) (2015.).

Težnja je prema razvoju grafičkih jezika koji podržavaju paralelnu obradu na kojem god sklopovlju da je dostupno, po mogućnosti bez potrebe da korisnik o tome vodi brigu. Uspješnost programa najviše ovisi o samim podacima kao i o stupnju mogućnosti paralelizacije algoritama za obradu. Jezici za arhitekturu protoka podataka najviše uspjeha doživljavaju u području znanosti o podacima (engl. *data science*), odnosno inženjerstva podataka (engl. *data engineering*).

Neovisno od sklopovskog ili programskog ostvarenja arhitekture protoka podataka, prema vrsti izvršavanja obradbe podataka, arhitektura protoka podataka dijeli se na:

- **skupno-sekvencijsku** (engl. *batch-sequential*) – podaci se između koraka (podsustava za obradu) prenose u cijelosti. Svaki podsustav se izvodi do kraja prije prelaska na idući korak (nema paralelizma) i neovisan je o drugim podsustavima.
- **cjevovode i filtere** (engl. *pipes-and-filters*) – inkrementalna, konkurentna i po mogućnosti paralelna transformacija podataka kroz korake (filtere), čim podaci postanu dostupni. Komponente ovog sustava su: izvori podataka, filteri, cjevovodi i ponori podataka; filteri su neovisni o ostalim filterima i ne pamte stanje, a cjevovodi su FIFO-međuspremnici u obliku U/I tokova podataka (*input/output stream*).

Primjene skupno-sekvencijske arhitekture su u diskretnim transakcijama (npr. bankovnim), obradi podataka koja nije pod strogim vremenskim ograničenjima i skupna obrada (npr. obračun plaća). Cjevovodi i filteri primjenjuju se u operacijskim sustavima Unix (i Linux) za ostvarenje komunikacije između procesa. Tamo se cjevovodi (u oznaci „|“) koriste za povezivanje filtera (procesa) koji obrađuju podatke. Ovisno o vrsti, filter mogu čitati podatke iz cjevovoda i/ili pisati podatke u cjevovod. Neki filteri (npr. *sort*, *grep*) konzumiraju sve ulazne podatke prije generiranja izlaza, a time narušavaju pretpostavku o inkrementalnom radu filtera čim određeni podatci postanu dostupni.

6.4.13. Arhitekturni stil repozitorija podataka

Arhitekturni stil repozitorija podataka (engl. *data repository*) predstavlja veliku skupinu sustava koji su, uz mnoge varijacije, koncentrirani na temu upravljanja i dijeljenja podataka. Arhitektura repozitorija podataka obuhvaća načine prikupljanja, rukovanja i očuvanja podataka. Karakteristične komponente ovog sustava su:

- **centralizirani repozitorij podataka** (engl. *data store*, *data repository*) koji određuje stanje sustava
- **skup neovisnih komponenti** koje upotrebljavaju repozitorij.

Temeljna prednost ove arhitekture je jednostavno dodavanje i povlačenje proizvođača i korisnika podataka. Problemi pri razvoju i korištenju ove arhitekture koncentrirani su oko:

- sinkronizacije sadržaja središnjeg repozitorija
- konfiguracije i upravljanje shemama struktura podataka u repozitoriju
- atomičnosti izvođenja transakcija prema središnjem repozitoriju
- konzistencije podataka u središnjem repozitoriju
- očuvanja (perzistencije) podataka u središnjem repozitoriju
- performansi cjelokupnog sustava.

Dva značajna primjera ovog stila su **baza podataka** i **oglasna ploča**. Kod baze podataka, vanjske komponente modificiraju ili čitaju podatke preko transakcijskog upita (engl. *query*). Baza podataka pohranjuje perzistentne podatke između različitih transakcija. Ona nema očekivanog fiksnog uređenja redoslijeda kojim transakcije dolaze. Umjesto toga, sustav za upravljanje bazom podataka upravlja svim pristupima

prema bazi podataka i odlučuje u slučaju potrebe za paralelnim zapisivanjima/izmjenama pojedinih dijelova baze podataka.

Oglasna ploča (engl. *blackboard*) koristi se kada postoji određeni težak problem koji nema jasno određeno rješenje ili postupak rješavanja. Primjer takvog sustava je sustav za prepoznavanje ljudskog govora s velikim vokabularom (sustav Hearsay i njegovi nasljednici). Temeljni koncepti ovog stila su:

- mnogi eksperti (procesi u računalu) promatraju rješavanje zajedničkog problema na ploči (također proces u računalu),
- svaki ekspert dodaje svoj dio u rješavanju cjelokupnog problema.

Osnovni dijelovi sustava oglasne ploče su:

1. **izvori znanja** (engl. *knowledge sources*) – odvojeni i neovisni dijelovi primjenskog znanja (eksperti). Oni ne surađuju međusobno izravno već samo putem oglasne ploče.
2. **oglasna ploča** – sadržava podatke o stanju problema, organizirane prema primjenskoj hijerarhiji u više razina. Izvori znanja, aktivirani promjenom sadržaja na oglasnoj ploči, mijenjaju sadržaje na određenoj razini što inkrementalno dovodi do rješenja problema.
3. **sustav za upravljanje stanjem oglasne ploče** – izvori znanja se odazivaju oportunistički kada se promjene na oglasnoj ploči na njih odnose.

Važna razlika između stila baze podataka i stila oglasne ploče je ta da vanjski procesi iniciraju promjenu sadržaja kod baze podataka, dok kod oglasne ploče promjena sadržaja inicira vanjske procese (izvore znanja).

U arhitekturi stila oglasne ploče nije unaprijed specificirano:

- struktura znanja (kako je znanje na ploči predstavljeno)
- kako izvori znanja dohvaćaju podatke s ploče
- kako izvori znanja zapisuju djelomične odgovore na oglasnu ploču
- kako izvori znanja koriste podatke s oglasne ploče
- kako se određuje kvaliteta rješenja problema
- kako se upravlja ponašanjem izvora znanja (koji je izvor znanja sljedeći na redu).

Znanje na oglasnoj ploči može biti predstavljeno na različite načine:

- jednostavnim matematičkim funkcijama,
- kolekcijom složenih logičkih izraza, pravilima i sl.

Upravljanje znanjem na oglasnoj ploči može biti:

- podatkovno inicirano (engl. *data driven*) – sljedeći korak je definiran stanjem podataka, te se poziva odgovarajući izvor podataka.
- inicirano ciljem (engl. *goal driven*) - upravljačka funkcija poziva izvor znanja koji najviše doprinosi pomaku prema cilju. Općenito je teško kodirati ciljeve.

U praksi se koristi kombinacija ovih dviju strategija.

6.4.14. Arhitekturni stil virtualnih strojeva

Virtualni stroj (engl. *virtual machine*, VM) je računalno okruženje (engl. *environment*) čiji su skup resursa i funkcionalnosti izgrađeni kroz programski sloj iznad nekog drugog programskog okruženja. Time se postiže apstrakcija računalnih resursa na određenom računalu gdje je virtualni stroj instaliran. Temeljna prednost ove apstrakcije je u odvajanju primjenskog programa od ostatka sustava izvan virtualnog stroja, a temeljni nedostatak je slabljenje performansi programa koji je pokrenut na virtualnom stroju.

VM se može koristiti u različitim kontekstima. Tako je moguće simulirati funkcionalnost nepostojeće sklopovske platforme pri čemu VM pomaže u razvoju programske potpore za nove uređaje. Konfiguracija više ne slijedi tradicijsku organizaciju (sklopovlje – OS – primjenski program) već jednu od nekoliko mogućnosti pod zajedničkim nazivom VM:

- hipervizorski VM
- udomaćen VM
- primjenski VM.

Hipervizorski virtualni stroj (engl. *hypervisor*, *bare-metal hypervisor*) je dodatni programski sloj koji se instalira na računalu odmah iznad sklopovlja. Na njega se oslanjaju jedan ili više operacijskih sustava (OS), gdje svaki smatra da je jedini OS u računalu. Sklopovske resurse, međutim, dijele svi OS-ovi koji su istovremeno pokrenuti na računalu. Prednost hipervizorskog VM-a je visoka učinkovitost, što mu daje prednost kod izvršavanja većeg broja zahtjevnijih, specifičnih aplikacija na pojedinim OS-ovima. Korištenjem hipervizorskog VM-a poboljšana je stabilnost – pad jednog OS-a ne uzrokuje pad cijelog računala. Primjeri

hipervizorskih VM-ova su proizvodi: Oracle VM server, VMware ESXi server i IBM z/VM.

Udomaćen virtualni stroj (engl. *hosted virtual machine*) izvodi se kao i svaki drugi primjenski program, kao proces instaliran na određenom OS-u. Pritom OS koji je domaćin ovom VM-u osigurava pristup sklopovlju. Međutim, udomaćeni VM definira zasebne resurse u vidu memorije, tvrdog diska i druge koje OS domaćin njemu dodjeljuje prilikom pokretanja. Udomaćeni virtualni stroj pokreće zaseban OS koji se izvodi iznad OS-a domaćina. Time je omogućeno pokretanje jednog OS-a (npr. Linuxa) nad drugim OS-om (npr. Windowsima). Udomaćeni virtualni strojevi nude manje učinkovito okruženje od hipervizorskog VM, ali osnovna prednost odvajanja primjenskog programa od sklopovlja i OS-a domaćina ostaje. Primjeri udomaćenih VM-ova su proizvodi: VMware Workstation, VMware Player, Oracle VirtualBox, Microsoft Virtual PC.

Virtualni stroj primjenske razine (engl. *process virtual machine*) je sličan udomaćenom VM-u, međutim on ne omogućuje definiranje instalacije novog OS-a, već samo pokretanje određenih primjenskih programa. Naprimjer, Javin VM je primjenski program i virtualni stroj primjenske razine (izvodi se na izvornom OS-u), dok se Javini primjenski programi izvode na Javinom VM-u. Docker je drugi primjer virtualnog stroja primjenske razine. Docker omogućuje pokretanje određene aplikacije unutar slabo izoliranog okruženja koje se naziva sadržajnik (engl. *container*). Dockerov sadržajnik je proces čija razina izolacije i sigurnosti omogućuje pokretanje više takvih programa na određenom OS-u domaćina, pa i na udomaćenom VM-u. Primjena Dockera je u udaljenom pristupu aplikacijama, budući da je cijeli razvijen u arhitekturnom stilu klijenta-poslužitelja.

6.4.15. Arhitektura programa u upravljanju procesima

Arhitektura programa u upravljanju procesima (engl. *process control*) treba biti razvijena tako da podržava sustav automatskog upravljanja. Sustav automatskog upravljanja može se ostvariti na dva načina:

- upravljanjem bez povratne veze (engl. *feedforward control*)
- upravljanjem s povratnom vezom (engl. *feedback control*).

U oba slučaja, programski sustav se sastoji od nadglednika (engl. *controller*) i procesa. Proces prima skup ulaznih varijabli i podskup od tih varijabli čije vrijednosti mijenjamo putem komponentne nadglednika. Po izlazu iz procesa dobivaju se vrijednosti upravljanih varijabli. Razlika između upravljanja bez povratne veze i onog

s povratnom vezom je u tome što kod upravljanja s povratnom vezom, izlazne vrijednosti upravljanih varijabli su ponovno dostupne komponenti nadglednika te mogu utjecati povratno na proces, što nije slučaj kod upravljanja bez povratne veze.

Primjena arhitekture programa u upravljanju procesima je, primjerice, u sustavima:

- termostata (za grijanje) u kućanstvima – izlazna temperatura stana utječe na nadglednik – termostat
- tempomata u vozilima – izlazna brzina vozila utječe na tempomat
- automatske navigacije plovila – izlazni smjer i brzina plovila utječu na kormilo za korekciju smjera i brzine, itd.

Kod upravljanja procesima, tehnička shema fizikalnog procesa je razumljiva. Međutim, postoje poteškoće koje programske jedinice uporabiti. Za odabir prikladne programske potpore potrebna je općenitija analiza programskih i sklopovskih dijelova te okoline izvođenja. Za ostvarenje takvih sustava može se koristiti stil arhitekture zasnovan na događajima kao i arhitektura protoka podataka. Primjerice, često se u sustavima za automatsko upravljanje koriste programske komponente koje implementiraju pravila neizrazite logike (engl. *fuzzy logic*).

7. Ispitivanje programske potpore

7.1. Temelji ispitivanja programske potpore

Programsku potporu nužno je ispitati budući da programi ne daju nikakvo jamstvo da će raditi pod svim mogućim okolnostima. U okviru programskog inženjerstva postoji posebno potpodručje koje se bavi isključivo ispitivanjem programske potpore (engl. *software testing*). SWEBOK (engl. *Software Engineering Body of Knowledge*), u inačici 3.0 iz 2013. sistematizira ispitivanje kao jedno od 15 područja znanja u okviru programskog inženjerstva [16]. Ovo područje znanja uključuje temelje ispitivanja programske potpore, tehnike ispitivanja, ispitivanje sučelja između čovjeka i računala, mjere vezane uz ispitivanje, razine ispitivanja, proces ispitivanja i praktična razmatranja vezana uz ispitivanje. Praktična razmatranja se obično odnose na alate koji podupiru proces ispitivanja kao i na specifičnosti ispitivanja za pojedine stilove arhitekture programske potpore (npr. za objektno usmjereni stil). U nastavku ovog poglavlja prikazat će se temelji ispitivanja programske potpore, razine ispitivanja, organizacija ispitivanja, strategije i pristupi ispitivanju, tehnike ispitivanja te automatizacija ispitivanja.

7.1.1. Definicija i ciljevi

Ispitivanje je aktivnost koja služi za vrednovanje i poboljšavanje kvalitete programskog proizvoda. Cilj ispitivanja je poboljšanje kvalitete proizvoda tako što se **pronalaze i otklanjaju kvarovi u implementaciji**. Ispitivanje u užem smislu jest dinamička verifikacija ponašanja programa, što znači da se program ispituje tijekom izvođenja u svrhu pronalaska pogrešaka. Formalna definicija glasi [16]:

Definicija 7.1. Ispitivanje je dinamička verifikacija programa koja predočava očekivana ponašanja na konačnom skupu ispitnih slučajeva koji su na odgovarajući način odabrani u okviru moguće beskonačne domene izvođenja.

Kao glavni pojam u ispitivanju navodi se ispitni slučaj.

Definicija 7.2. Ispitni slučaj (engl. *test case*) u užem smislu je uređeni par (ulaz, izlaz), gdje je ulaz ulazni podatak, a izlaz očekivani izlazni podatak iz programa, zabilježen prije provođenja ispitivanja.

Ulaznim podatkom se smatra onaj podatak koji se predaje određenoj komponenti sustava koja takve podatke obrađuje. Ulazni podatci se jasno definiraju, što znači da su im zadane točne vrijednosti. Izlazni podatak je očekivani rezultat obrade ulaznih podataka korištenjem komponente koja se ispituje. Uređeni par ulaz-izlaz definira se prije pokretanja ispitivanja. Nakon što se provede ispitivanje, zabilježi se stvarni izlaz koji se zatim uspoređuje s očekivanim izlazom. Osim ispitnog slučaja, potrebno je unaprijed poznavati i kriterij uspješnog prolaska ispitnog slučaja. Najčešći kriterij jest da izlazni podatak potpuno odgovara stvarno dobivenom rezultatu provođenja ispitivanja. Institut inženjera elektrike i elektronike (IEEE) definira ispitni skup (engl. *test set*) 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.

Prilikom ispitivanja postoji uvijek konačan broj ispitnih slučajeva koji se mogu provjeriti, zbog realnih ograničenja vremena i resursa koji su na raspolaganju ispitivačima. Međutim, sustav koji se ispituje može biti beskonačan u svojem broju stanja ili barem daleko prevelik da bi ga se moglo ispitati u cijelosti pa se stoga ispitni slučajevi odabiru na odgovarajuće načine kako bi što bolje ispitali sustav. Odabir ovisi o više faktora, o čemu će više riječi biti kasnije u okviru ovog poglavlja.

Osim za provjeru obavlja li program ispravno željenu funkcionalnost, ispitivanje se koristi i da bi se utvrdilo izvodi li se program u skladu sa zadanim nefunkcionalnim zahtjevima (npr. dovoljno velikom brzinom, podrškom za istovremeni pristup određenom broju korisnika). Za pokazivanje ispravnosti rada programa koriste se razni alati: drugi programi, mjerni instrumenti, analizatori i slično te logika i matematika. Pri tome se izrađuju modeli sustava te se eksperimentira s različitim pristupima izgradnje programskog proizvoda u svrhu ostvarivanja visoke razine funkcionalnosti i kvalitete.

Važno je naglasiti da ispitivanje može ustanoviti postojanje pogreške, ali nikada ne može ustanoviti njezino nepostojanje, što je utvrdio još Dijkstra 1972. To znači da se pregledom programa u izvođenju može pronaći i zatim, po mogućnosti, ukloniti pogreška, no nije moguće na temelju provedenog ispitivanja utvrditi da program nema pogrešaka.

7.1.2. Statička, dinamička i formalna verifikacija programa

Ispitivanje programa u užem smislu naziva se **dinamičkom verifikacijom**, što znači da se program ispituje tijekom svojeg izvođenja. Osim dinamičke verifikacije, postoje još **statička verifikacija** i **formalna verifikacija programa**. Budući da

je dinamičkoj verifikaciji posvećeno cijelo poglavlje, ovdje će se ukratko objasniti samo statička i formalna verifikacija.

Statička verifikacija programske potpore jest postupak u kojem se program i ostali dijelovi sustava ne izvode, već ih se samo analizira kako bi se utvrdili nedostatci i potencijalno pronašli kvarovi. Ona se provodi na specifikaciji zahtjeva, različitim razinama oblikovanja sustava i poglavito na programskom kodu.

Dijelovi projekta (artefakti) koji nisu programski kôd, kao što su dokumenti zahtjeva i arhitekture, analiziraju eksperti domene, članovi projektnog tipa kao i profesionalni ispitivači programske potpore, pri čemu svi pokušavaju pronaći nedosljednosti, dvosmislenosti ili druge vrste kvarova koji su uvedeni tijekom razvoja projekta.

Za analizu programskog koda statičkom verifikacijom najčešće se koriste specijalni CASE-alati, tzv. **statički analizatori** (engl. *static analyzer*), npr. LINT, PMD, SpotBugs, StyleCopAnalyzers i drugi. Statički analizatori mogu pronaći i označiti različite sintaksne i semantičke pogreške u programima, a određene inačice statičkih analizatora su već ugrađene u IDE okruženja za razvoj programa pa čak i u uređivačima teksta opće namjene.

Osim automatskih analizatora koda, statičku verifikaciju mogu provesti i razvojni inženjeri koji su pisali program, neovisni ispitni timovi ili drugi dionici na projektu. Pritom statička verifikacija može biti manje ili više formalna. Neki od neformalnijih pristupa su:

- **uklanjanje kvarova uz pomoć gumene patkice** (engl. *rubber duck debugging*) – metoda u kojoj razvojni inženjer provjerava vlasititi programski kôd pričajući naglas samome sebi (ili gumenoj patkici) što napisani program radi i na taj način pokušava ustanoviti odgovara li to onome što je trebalo napraviti.
- **prolazak ili češljanje programskog koda** (engl. *software walkthrough*) – metoda u kojoj razvojni inženjer ili njegov nadređeni organiziraju sastanak na kojem se prolazi kroz važnije dijelove koda uz objašnjenje što taj dio kôda radi. Razlika u odnosu na *rubber duck debugging* jest ta da u ovom slučaju razvojni inženjer objašnjava svoj kôd više ljudi, čime se povećava mogućnost uočavanja mogućih kvarova.

Formalniji pristup je **nadzor i inspekcija koda** (engl. *code inspection, code review*). Svrha inspekcije je utvrđivanje usklađenosti sa svim potrebnim zahtjevima i normama koji su definirani projektom. Nadzor koda najčešće radi zasebni ispitni tim, što zahtjeva dobro planiranje i raspodjelu zadataka na projektu, formalno bilježenje rezultata inspekcije i obradu rezultata.

Formalna verifikacija primjenjuje formalne metode matematičke logike kako bi dokazala ispravnost programa s određenom matematičkom izvjesnošću. Ona se najčešće radi nad modelima programa, a ne nad samim programima (osim u slučaju jednostavnih programa). Formalna verifikacija zahtijeva korištenje alata koji izvorni program mogu automatski pretvoriti u neku apstrakciju (npr. različitih strojeva s konačnim brojem stanja) ili koriste posebni programski jezik za izradu modela izvornog programa te potom primijenjuju propozicijsku, predikatnu i/ili vremensku logiku kako bi ustanovili vrijede li određene specifikacije za koje bi program trebao raditi.

7.1.3. Pojmovi pogreške, kvara i zatajenja

Korisnici često govore da postoji problem s programom, da program ne radi ili da izbacuje neku pogrešku. Osviješteniji korisnici kao i razvojni inženjeri kolokvijalno će reći da program ima *bug* (engl. *bug*). Nedosljednost ovakve terminologije dovodi do težeg utvrđivanja što je uzrok, a što posljedica te koliko je zapravo situacija problematična. Budući da je ispitivanje inženjerska aktivnost, nužno je biti precizan prilikom definiranja osnovnih pojmova koji se koriste u tom području. U području programskog inženjerstva razlikuju se termini **kvara, pogreške i zatajenja**.

Definicija 7.3. Kvar (engl. *fault*) je bilo koji uvjet zbog kojeg sustav ne obavlja ispravno zahtjevanu funkciju.

Primjer kvara na razini koda bila bi petlja koja ne uzima u obzir posljednji element u polju brojeva. Kvar je skriven sve dok ga netko ne uči, bilo proučavanjem projektne dokumentacije bilo izvođenjem programa pod određenim uvjetima.

Definicija 7.4. Pogreška (engl. *error*) je dio stanja sustava (programa u izvođenju) koje je prouzrokovalo odstupanje od željenog ponašanja. Ona je manifestacija kvara.

Pogrešku može, ali i ne mora vidjeti krajnji korisnik, što ovisi o načinu na koji je program pripremljen na pojavu pogrešaka (primjerice, hvatanje iznimke). Primjer pogreške bila bi prekoračenje dozvoljene količine memorije. Uzrok te pogreške bio je primjerice kvar u pozivu za alociranjem memorije.

Definicija 7.5. Zatajenje (engl. *failure*) je pogreška koju vidi krajnji korisnik.

Kod zatajenja, sustav ne zadovoljava specifikaciju i problem je vidljiv svima koji ga koriste. Uvjeti pod kojima se događa zatajenje su:

1. **doseg** (engl. *reachability*) – mjesto kvara u programu mora biti dosegnuto
2. **infekcija** (engl. *infection*) – stanje programa mora biti neispravno
3. **propagacija** (engl. *propagation*) – inficirano stanje mora uzrokovati promjenu nekog izlaza programa.

Potrebno je uvijek težiti što manjem broju zatajenja, čime postižemo veću pouzdanost sustava.

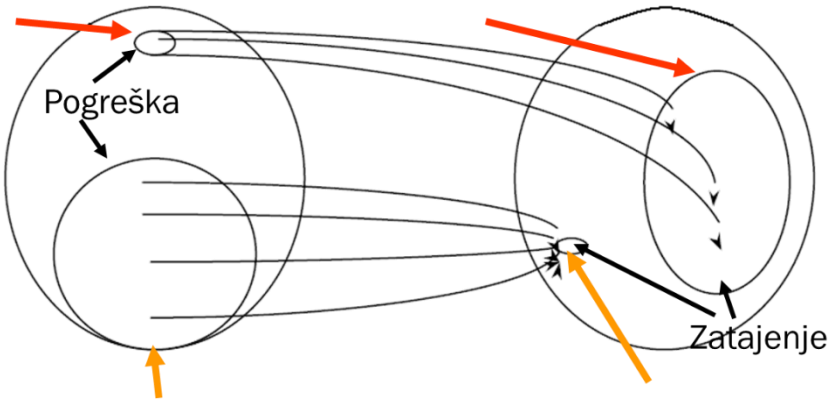
Zatajenje programskog proizvoda može biti objektivno ili subjektivno. Objektivno zatajenje je ono koje ne ispunjava zadane zahtjeve. Subjektivno zatajenje je ono koje zadovoljava sve zadane zahtjeve, no ne i očekivanja korisnika. U tom slučaju, zahtjevi se trebaju nanovo definirati i sustav doraditi. Razlozi zatajenja mogu biti:

- zahtjevi su nepotpuni, nekonzistentni, nemogući za implementaciju
- pogrešna interpretacija zahtjeva
- kvar u oblikovanju arhitekture
- kvar u oblikovanju programa (npr. upotrijebljen neodgovarajući algoritam)
- kvar u programskom kodu
- dokumentacija nekorektno opisuje ponašanje sustava.

Prema posljedicama zatajenja, pogreške u programima se mogu kategorizirati uzlazno po razini štete na sljedeće tipove:

- blage (engl. *mild*)
- dosadne (engl. *annoying*)
- uznemiravajuće (engl. *disturbing*)
- ozbiljne (engl. *serious*)
- granične (engl. *extreme*)
- katastrofalne (engl. *catastrophic*)
- zarazne (engl. *infectious*).

Inženjeri su uočili da se pogreške i zatajenja odnose približno po **Paretovom načelu** (engl. *Pareto principle*) ili po **Zakonu ključne manjine** (engl. *Law of the vital few*), slika 7.1.



Slika 7.1 Paretovo načelo

Ovo načelo glasi da približno 20% pogrešaka u programu dovodi do 80% zatajenja sustava. Drugim riječima, približno 20% koda sadrži kvar čijim rješavanjem će se moći spriječiti 80% svih zatajenja. Nadalje, uočeno je da su kvarovi često grupirani unutar nekog dijela cjelokupnog sustava, dok ih u drugim dijelovima ima relativno malo. Stoga je dobra praksa identificirati upravo onu komponentu ili nekoliko njih koje uzrokuju pad sustava. Iskustvo pokazuje da su uglavnom najkompleksnije komponente one koje dovode do najviše zatajenja te je stoga važno takve komponente najdetaljnije ispitati.

7.1.4. Upravljanje pogreškama, pouzdanost i ispitljivost programa, norme ispitivanja

Pogreške u složenim sustavima su neizbježne te se uvijek traže načini kako da se što više smanji njihov broj. Ugrubo, pristupi smanjenju broja pogrešaka mogu se podijeliti u tri skupine:

1. prevencija (engl. e*rror prevention*)
2. pronalaženje (engl. *error detection*)
3. oporavak (engl. *error recovery*).

Prevencija pogrešaka uključuje: uporabu pogodnih metoda oblikovanja programske potpore kako bi se smanjila složenost sustava i time olakšalo pronalaženje pogrešaka, sprečavanje nekonzistentnosti u inačicama programske

potpore (npr. korištenjem sustava za kontrolu inačica), primjena formalne verifikacije za sprečavanje kvarova u kritičnim dijelovima sustava i drugo.

Pronalaženje pogrešaka odvija se tijekom rada programa pri čemu se provodi ispitivanje, pronađene pogreške se ispravljaju (engl. *debugging*), a provodi se i nadzor rada programa (engl. *monitoring*) kako bi se pratilo izvođenje nakon ispravljanja pogreške.

Oporavak od pogreške vrlo je bitan dio prilikom projektiranja sustava. Oporavak od pogreške u radu programa (npr. dijeljenje s nulom, korištenje neinstanciranog objekta i sl.) provodi se u ovisnosti o programskom jeziku. Važno je predvidjeti moguća mjesta pojave pogrešaka te implementirati mehanizam oporavka kako ne bi došlo do pada cijelog sustava. Prilikom rada s bazom podataka, oporavak od pogreške treba osigurati ponavljanje neuspješno provedenih transakcija kad god je to moguće.

Pouzdanost programske potpore treba rasti tijekom razvojnog ciklusa da bi nakon puštanja u upotrebu dosegla visoku razinu. To znači da se kvarovi, ako i postoje, ne bi smjeli u konačnici manifestirati u obliku zatajenja nakon završetka razvojnog procesa. Različiti programski proizvodi pokazuju različitu razinu pouzdanosti. Kritični trenutci koji utječu na smanjenje pouzdanosti najčešće su različite nadogradnje, odnosno uvođenje promjena u sustav. Razlog tome su u pravilu nekompatibilnosti s vanjskim sustavima, s čime projektanti i razvojni inženjeri nisu računali prilikom nadogradnje. U slučaju kritičnih sustava, brza reakcija razvojnog tima je nužna kako bi se sustav što prije osposobio.

U uporabi se nalazi velik broj definiranih **normi ispitivanja** (engl. *testing standards*) koje specificiraju način provođenja ispitivanja. Veće tvrtke mogu se odlučiti za slijeđenje normi ispitivanja pri čemu imaju prednost što mogu klijentu tvrditi da im se ispitivanje zaista provodilo na normom određeni način. Osim povećane organiziranosti, dodatna prednost korištenja normi je postojanje zapisa, odnosno dokumentacije ispitivanja. U slučaju da nešto u sustavu pođe po krivu i dogodi se ozbiljno zatajenje, lakše je ustanoviti izvor zatajenja. Nedostatak provođenja ispitivanja putem određenih normi jest to što su potrebni resursi (ljudi i vrijeme) koje je nužno za to odvojiti.

Od postojećih normi potrebno je istaknuti dvije:

1. IEEE Std. 829-2008 Standard for Software Test Documentation
2. ISO/IEC 29119 Software Testing.

IEEE Std. 829 definira sadržaj dokumenta ispitivanja programske potpore. Dokumentacija ispitivanja sastoji se, prema toj normi, od sljedećih dijelova:

- plana ispitivanja (engl. *test plan*)
- specifikacije oblikovanja ispitivanja (engl. *test design specification*)
- specifikacije ispitnih slučajeva (engl. *test case specification*)
- specifikacije procedure ispitivanja (engl. *test procedure specification*)
- dnevnika ispitivanja (engl. *test log*)
- izvješća o odstupanjima (engl. *test incident report*)
- sažetka izvješća ispitivanja (engl. *test summary report*).

Plan ispitivanja uključuje kraći opis procesa ispitivanja, sljedivost zahtjeva, elemente ispitivanja (što i zašto se ispituje), vremenski raspored ispitivanja, postupak bilježenja rezultata ispitivanja, zahtjeve okoline (programske i sklopovske) i opis planiranih ograničenja ispitivanja. U **specifikaciji oblikovanja ispitivanja** detaljnije se razrađuje što se treba ispitivati, koja su svojstva toga što se ispituje te pristupi i očekivani rezultati ispitivanja. U **specifikaciji ispitnih slučajeva** obrađuju se svi ispitni slučajevi koji se provode, što u širem smislu uključuje i navođenje podataka o tome tko provodi ispitivanje, nad kojom komponentom ili jedinicom se provodi ispitivanje, koja se mjera za podudaranje rezultata ispitivanja koristi i slično. U **proceduri ispitivanja** navode se svi koraci provođenja ispitivanja. U **dnevnik** se zapisuje tijek provođenja ispitivanja. Odstupanja od zadanih mjera podudaranja između dobivenih i očekivanih rezultata za pojedine ispitne slučajeve navode se u **izvješću o odstupanjima**. Na kraju se najčešće navodi i **sažetak izvješća ispitivanja**, koji sumira ostale dijelove ispitne dokumentacije.

Norma ISO/IEC 29119 se postupno nadograđivala od 2007. do 2016. s ciljem razvoja jedinstvene norme koja bi pokrila korištenje ispitivanja programske potpore unutar bilo kojeg životnog ciklusa programske potpore i bilo koje organizacije. Namjena joj je da zamijeni normu IEEE 829 i nekoliko drugih manje značajnih normi. Ova norma sastoji se od 5 dijelova:

- ISO/IEC 29119-1: Koncepti i definicije (engl. *Concepts & Definitions*)
- ISO/IEC 29119-2: Procesi ispitivanja (engl. *Test Processes*)
- ISO/IEC 29119-3: Dokumentacija ispitivanja (engl. *Test Documentation*)
- ISO/IEC 29119-4: Tehnike ispitivanja (engl. *Test Techniques*)
- ISO/IEC 29119-5: Ispitivanje vođeno ključnom riječi (engl. *Keyword Driven Testing*).

Detalji o ovoj normi dostupni su na web sjedištu [17].

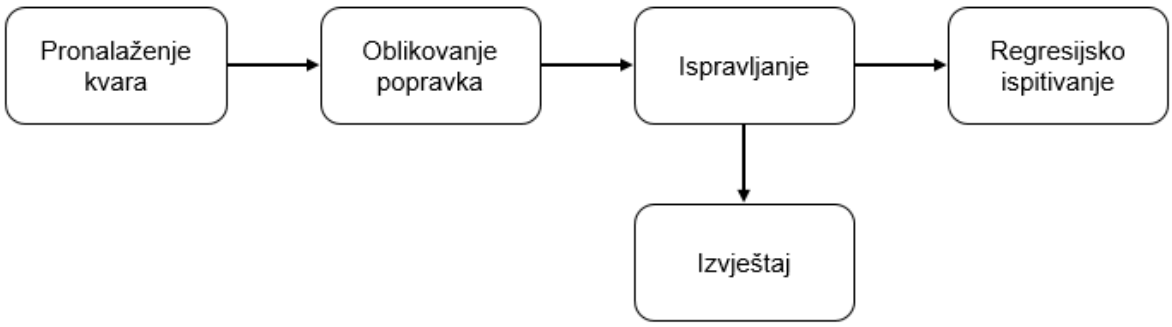
7.1.5. Aktivnosti ispitivanja

Ispitivanje je proces koji se sastoji od nekoliko ključnih koraka:

1. **planiranje ispitivanja** – uključuje postavljanje ciljeva ispitivanja, prikupljanje informacija relevantnih za ispitivanje (npr. ispitna okolina, tehnologije i jezik implementacije sustava i sl.) te modeliranje sustava. Pri tome se definira koji su prioriteti ispitivanja i određuje se koja će se strategija ispitivanja primijeniti (npr. iscrpno, slučajno, istraživačko ispitivanje...).
2. **oblikovanje ispitnih slučajeva** (engl. *test design*) – ispitni slučajevi se oblikuju kako bi pokrili svu potrebnu funkcionalnosti programskog proizvoda.
3. **automatizacija ispitivanja** (engl. *test automation*) – uključuje pripremu razvojne okoline za ispitivanje te izradu ispitnih slučajeva. Metode programa mogu se izravno ispitivati ili se mogu ispitivati njihovi pojednostavljeni modeli.
4. **provođenje ispitivanja** (engl. *test execution*) – izvođenje ispitnih slučajeva i prikupljanje rezultata.
5. **vrednovanje rezultata ispitivanja** (engl. *test evaluation*) – utvrđuje se odgovaraju li rezultati ispitivanja očekivanima, pri čemu je važno dobro poznavati domenu za koju je programski sustav napravljen kao i sam postupak ispitivanja kako bi se rezultati ispitivanja ispravno vrednovali.

Ovisno o rezultatima, nakon što je ustanovljeno mjesto kvara, kreće se u njegovo uklanjanje (engl. *debugging*), slika 7.2. Nakon uklanjanja kvara provodi se ponovno ispitivanje kako bi se utvrdilo je li rješavanjem jednog kvara uveden novi kvar u sustav. Ovo ponovno ispitivanje nakon unošenja promjena naziva se još i **regresijsko ispitivanje** (engl. *regression testing*).

Definicija 7.6. Regresijsko ispitivanje je ispitivanje koje se provodi nakon što je ispravljanjem pogreške unesena promjena u sustav.



Slika 7.2 Otkrivanje i uklanjanje kvara te regresijsko ispitivanje.

Važnost regresijskog ispitivanja je velika, budući da ispravljanje kvara ili onoga što je razvojni inženjer percipirao kao kvar može u nekim sustavima prouzročiti uvođenje novog kvara u sustav. Pri regresijskom ispitivanju obično se uklanjaju neki ispitni slučajevi koji više nisu potrebni nakon što je sustav izmijenjen, a neki novi se prema potrebi dodaju kako bi pokrili sve napravljene izmjene u kodu.

7.1.6. Ispitni tim

Manje tvrtke uglavnom provode ispitivanje tako što razvojni inženjer ujedno ispituje svoj dio koda. Osim toga, provodi se i unakrsno ispitivanje, gdje razvojni inženjeri međusobno provjeravaju kodove, tako povećavajući šansu pronalaska pogrešaka koje sami nisu uočili. Veće organizacije u pravilu formaliziraju proces ispitivanja tako da oforme tim koji je specijaliziran za ispitivanje programske potpore. Takav ispitni tim može se sastojati od unutarnjih članova projektnog tima ili od vanjskih članova. Prednost sastavljanja ispitnog tima od vanjskih članova jest nepristrana i neovisna perspektiva prilikom ispitivanja, a nedostatak je moguće nedovoljno snalaženje u projektu koji treba ispitivati. Neki ispitni timovi sastoje se i od vanjskih i od unutarnjih članova. Veličina ispitnog tima ovisi o veličini projekta koji se ispituje kao i o vremenskim ograničenjima.

Na velikom projektu, **voditelj projekta** (engl. *project manager*) upravlja svim resursima na projektu pa ujedno i procesom ispitivanja. **Analitičar na projektu** (engl. *project analyst*), između ostalog, planira kad se ispitivanje provodi i kako se uklapa u cjelokupni proces. Nadalje, **voditelj upravljanja kvalitetom** (engl. *quality assurance manager*) određuje normu prema kojoj se provodi ispitivanje i provjerava sukladnost provođenja ispitivanja i zadane norme. Imenuje se **voditelj ispitnog tima** (engl. *test manager*) čiji je zadatak analizirati zahtjeve na ispitivanje, oblikovati strategiju i metodologiju ispitivanja te predložiti ispitne slučajeve i podatke koji će se provjeravati. Konačno, **profesionalni ispitivači** (engl. *testers*), odnosno

članovi ispitnog tima pripremaju i provode ispitivanje te nastoje pronaći i ispraviti pogreške.

Ispitivanje mogu naknadno provoditi i **korisnici programskog sustava** kod kojih je to najčešće neformalno, uz mogućnost dojavljivanja pojave zatajenja sustava.

7.2. Faze i razine ispitivanja

Ispitivanje programske potpore složen je i dugotrajan proces koji započinje u ranim fazama razvoja programske potpore kada se ispituje rad dijelova (komponenti) programske potpore, nastavlja se ispitivanjem potpuno izgrađenog sustava na kraju procesa razvoja i prije puštanja u pogon te konačno završava ispitivanjem od strane korisnika koji odlučuju o prihvatljivosti izgrađenog sustava.

Faze ispitivanja se mogu ugrubo podijeliti na:

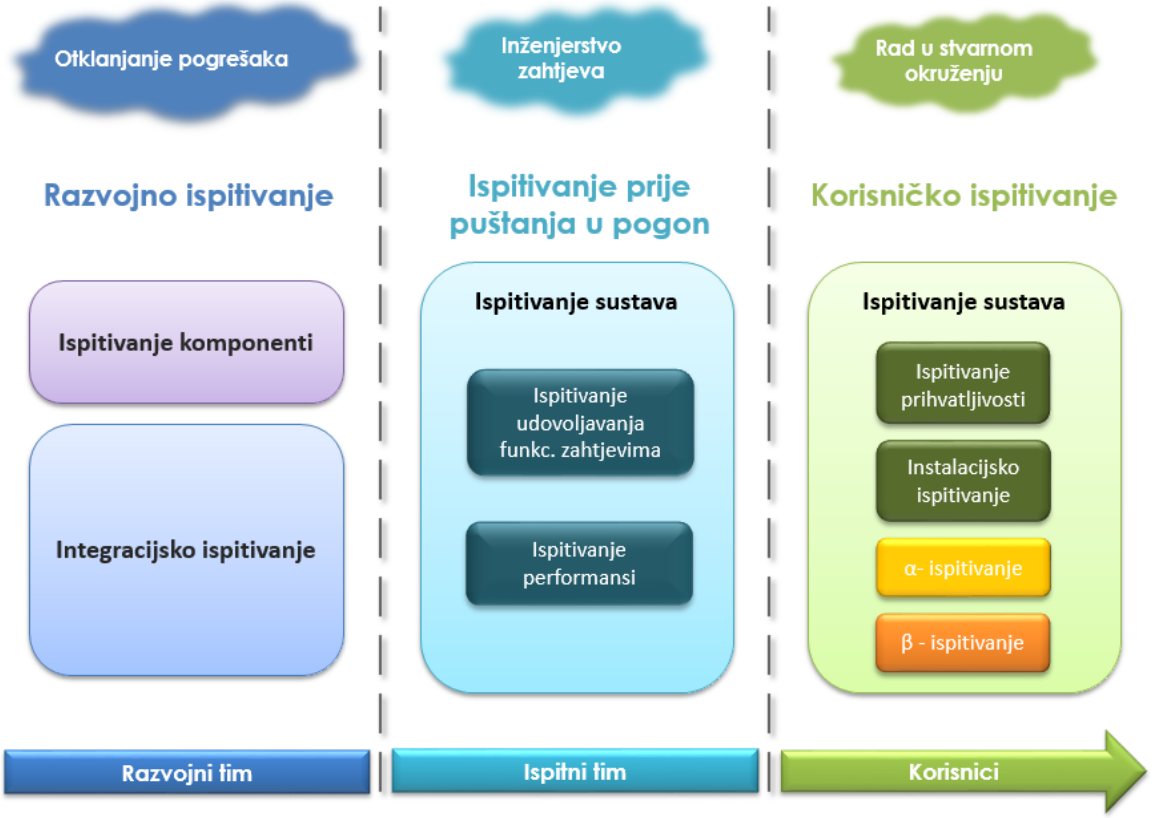
1. razvojno ispitivanje
2. ispitivanje prije puštanja u pogon
3. korisničko ispitivanje.

Razvojno ispitivanje (engl. *development testing*) prva je faza ispitivanja koja se izvodi tijekom razvoja i oblikovanja samog sustava. Ispitivanje se provodi na razini pojedinačne komponente te integracije više komponenata u cjelini. Sve ispitne aktivnosti izvodi tim koji razvija sustav. Najčešći ispitivači su programski inženjeri koji sustav i razvijaju, no u zahtjevnijim slučajevima može se razvojnom timu u sastav dodati i jedan ili više ispitivača koji će biti zaduženi za osmišljavanje i provođenje testova. Za sustave od kojih se zahtijeva visoka razina pouzdanosti često se koriste formalne metode verifikacije programske potpore pri čemu se najčešće u tom slučaju ispituju modeli takvih sustava. Cilj razvojnog ispitivanja je prvenstveno otkrivanje i ispravljanje pogrešaka koje nastaju prilikom razvoja.

U drugoj fazi – **ispitivanju prije puštanja u pogon**, poseban ispitni tim ispituje cjeloviti sustav. Fokus ispitivanja se pomiče s otkrivanja i uklanjanja pogrešaka na provjeru zadovoljavanja funkcionalnih i nefunkcionalnih zahtjeva – što je važan zadatak inženjerstva zahtjeva.

Nakon što je sustav uspješno prošao drugu, ulazi u treću, završnu fazu ispitivanja. **Sustav ispituje najprije manja, a zatim i veća skupina korisnika** koja daje konačno mišljenje o prihvatljivosti programskog proizvoda koji može biti prihvaćen uz (najčešće) manje izmjene i dorade ili pak (rjeđe) zahtijevati znatniju doradu.

Iako je uloga korisnika u ispitivanju najistaknutija u zadnjoj fazi, korisnik može (i trebao bi) biti uključen u proces razvoja programske potpore od samog početka i time povećati izglede za uspješan dovršetak izrade programskog proizvoda unutar zadanih vremenskih rokova i budžeta. Slika 7.3 prikazuje redoslijed faza ispitivanja programske potpore.



Slika 7.3 Faze ispitivanja programske potpore.

Uz iznimku vrlo malih programa, programski proizvodi ne mogu se ispitivati kao cjelovite, monolitne jedinice. Stoga se ispitivanje programske potpore u praksi provodi na razini: **komponenti, integracije** te **sustava u cjelini.**

7.2.1. Ispitivanje komponenti

Komponente su osnovni građevni blokovi svakog sustava. Pojam komponente u kontekstu razvoja programske potpore ima šire značenje te može označavati pojedinačne funkcije ili metode unutar objekta, razrede (u OO paradigmi) ili pak složene komponente s definiranim sučeljima za pristup njihovim funkcijama, kao što je opisano u ranijim poglavljima. Proces ispitivanja komponenti najčešće se dijeli u dvije podfaze:

1. **ispitivanje jedinica** (engl. *unit testing*) – odnosi se na najmanje moguće zasebne dijelove programa (razredi i metode)
2. **ispitivanje komponenti** (engl. *component testing*) – odnosi se na pojam komponente s definiranim sučeljima.

Ispitivanje jedinica

Ispitivanje jedinica (engl. *unit testing*) podrazumijeva ispitivanje funkcionalnosti najmanjih dijelova programa: razreda i pripadajućih metoda i atributa. Ovo je najniža razina ispitivanja s obzirom na granularnost i stupanj integracije te stoga treba biti najiscrpnija. Potrebno je ispitati svaki atribut i metodu nekog objekta te dovesti objekt u sva moguća stanja (simulirati sve događaje koji mogu dovesti do promjene stanja objekta).

Razlikuju se dvije vrste testova:

- normalan način rada – oponaša se tipičan način korištenja
- neispravni unosi i rubni uvjeti – provjerava se obrađuje li jedinica takve unose na ispravan način ili oni dovode do nestabilnosti, rušenja ili davanja netočnih izlaza.

Primjer ispitivanja jednostavnog razreda koji predstavlja račun u banci dan je u nastavku.

Primjer 7.1. Razred BankovniRacun ima jedan atribut; privatnu varijablu _stanje te tri metode: `Uplati()` za uplatu iznosa, `Isplati()` za isplatu iznosa i `Stanje()` za provjeru stanja računa. Metode za uplatu i isplatu sredstava na račun su implementirane tako da ne dozvoljavanju uplatu/isplatu negativnog iznosa te u tom slučaju javljaju grešku `NegativniBrojException`(). Također, metoda za isplatu računa osigurava da se ne može isplatiti veći iznos od trenutnog stanja na računu javljanjem greške `NemaDovoljnoSredstavaException()`.

```
public class BankovniRacun{
	private float _stanje;
	public BankovniRacun(){
		_stanje=0.0;
	}

	public void Uplati (float iznos){
		//provjera da iznos nije negativni broj
		if (iznos<0.0) throw new NegativniBrojException();
		_stanje+=iznos;
	}

	public void Isplati (float iznos){
		//provjera da iznos nije negativni broj
		if (iznos<0.0) throw new NegativniBrojException();
		if (_stanje<iznos) throw new
NemaDovoljnoSredstavaException(),
		_stanje-=iznos;
	}

	public float Stanje(){
		return _stanje;
	}
}
```

Ispitivanje jedinica (u ovom slučaju razreda `BankovniRacun`) najčešće se provodi korištenjem nekog od postojećih radnih okvira koji automatiziraju i ubrzavaju proces ispitivanja (više o ovim računalnim alatima može se pročitati u poglavlju 7.6). Najpoznatiji primjeri su radni okviri **JUnit** i **NUnit**. Primjer 7.2, dan u nastavku, prikazuje jedan ispitni razred (`TestClass`) u kojem su grupirane sve ispitne metode (`TestMethod`) za provedbu ispitivanja jedinice (razreda) `BankovniRacun`.

Najprije se provjerava normalan (tipičan) način korištenja, npr. uplata određenog iznosa na račun i zatim provjera stanja na računu nakon uplate. Vrlo je sličan postupak provjere isplate s računa. Radni okviri za provođenje ispitivanja jedinica imaju metode `Assert` za usporedbu očekivanog i dobivenog izlaza.

Primjer 7.2. Ispitni razred s ispitnim metodama kojima se ispituje normalan rad.

```
[TestClass]
public class BankovniRacunTests{

	public BankovniRacunTests(){
	}
}

[TestMethod]
public void UplatiIznosNaRacun(){
	BankovniRacun br= new BankovniRacun();
	br.Uplati(1000.5);

	//stanje racuna mora biti 1000.5
	Assert.AreEqual(1000.5, br.Stanje());
}

[TestMethod]
public void IsplatiIznosSaRacuna(){
	BankovniRacun br= new BankovniRacun();
	br.Uplati(1000.5);
	br.Isplati(0.5);

	//stanje racuna mora biti 1000.0
	Assert.AreEqual(1000.0, br.Stanje());
}
```

Nakon što je ispitan normalan način rada, ispituju se abnormalni unosi, npr. pokušaj uplate, a zatim i isplate negativnog iznosa na račun slučaj pokušaja isplate većeg iznosa sredstava od stanja računa, kao što je prikazano u primjeru 7.3. Kod ovakvog načina ispitivanja, kada se očekuje dojava pogreške od strane metode koja se ispituje, radni okviri najčešće zahtijevaju stavljanje atributa tipa `[ExpectedException(typeof(Exception)]` ispred same ispitne metode.

Primjer 7.3. Ispitne metode razred BankovniRacunTests kojima se ispituju rubni unosi, tj. pokušavaju se izazvati pogreške.

```
[TestMethod]
[ExpectedException(typeof (NegativniBrojException))] //pokusaj uplate
negativnog iznosa
public void UplatiNegativno(){
	BankovniRacun br= new BankovniRacun();
	br.Uplati(-1);
}

[TestMethod]
[ExpectedException(typeof (NedovoljnoSredstavaException))]
public void NedovoljnoNovaca(){
	BankovniRacun br= new BankovniRacun();
	br.Uplati(1000);
	br.Isplati(2000);
}
```

Važno je uočiti da u ovom primjeru atribut `_stanje` razreda `BankovniRacun` nije bilo moguće direktno ispitati budući da je označen kao privatan; umjesto toga, on je ispitan indirektno kroz poziv metode koja vraća stanje računa sadržano u atributu `_stanje`.

Ispitivanje u slučaju nasljeđivanja nešto je složenije, osobito u slučaju nadjačavanja tj. reimplementacije metoda. Tada je nužno ispitati rad naslijeđenih metoda na svakom mjestu u hijerarhiji gdje su implementirane i to uzimajući u obzir karakteristike svakog pojedinog razreda koji (re)implementira takvu metodu.

Ispitivanje komponenti

Programske komponente predstavljaju prvi stupanj integracije i sastavljene su od nekoliko jedinica s intenzivnom međusobnom interakcijom kojima se pristupa preko zadanih **sučelja.** Postoji nekoliko tipova sučelja:

1. **parametarsko sučelje** (engl. *parameter interface*) – najčešća vrsta sučelja; koriste ga metode za prijenos parametara putem argumenata i povratnih vrijednosti.

2. **sučelje zasnovano na dijeljenoj memoriji** (engl. *shared memory interface*) – dvije ili više komponenata komunicira koristeći zajednički odsječak memorije; najčešće korišteno u komunikaciji između dvije dretve.
3. **proceduralno sučelje** (engl. *procedural interface*) – jedna komponenta enkapsulira skup funkcionalnosti (metoda i procedura) koje mogu pozivati ostale komponente; karakteristično za objektnu paradigmu.
4. **sučelje zasnovano na porukama** (engl. *message passing interface*) – komponente pružaju jedna drugoj usluge tako da međusobno razmjenjuju poruke, što je tipično za paralelne i raspodijeljene sustave s niskom razinom kohezije.

Ispitivanje komponenti (engl. *component testing*) svodi se na ispitivanje njihovih sučelja i otkrivanje mogućih kvarova. Mogući tipovi kvarova sučelja su:

1. **pogrešna uporaba sučelja** (engl. *interface misuse*) – komponenta koristi sučelje druge komponente na pogrešan način, što je tipično za parametarska sučelja (pogrešan broj i/ili tip parametara).
2. **nerazumijevanje sučelja** (engl. *interface misunderstanding*) – postoji pogrešna pretpostavka o usluzi/funkcionalnosti komponente čije se sučelje koristi (npr. korištenje binarne pretrage nad nesortiranim poljem).
3. **vremenske pogreške** (engl. *timing errors*) – komponente koje komuniciraju nisu vremenski sinkronizirane. Podaci nisu spremni za korištenje u zadanom vremenskom trenutku (karakteristično za sustave za rad u stvarnom vremenu). Vremenske pogreške su najčešće prisutne u slučajevima kada se koriste sučelja dijeljene memorije ili razmjene poruka.

Otkrivanje i ispravljanje kvarova sučelja znatno je teže i složenije nego je to slučaj kod ispitivanja jedinica. Jedan od osnovnih izvora kvarova i pogrešaka je nepotpuno poznavanje načina na koji funkcionira komponenta čije se sučelje koristi. To je osobito slučaj kada se koriste komponente koje je razvio drugi tim ili čak druga tvrtka. Također, jedan od mogućih izvora kvarova je nedovoljna ispitanost jedinica od kojih se komponenta sastoji.

U praksi se pokazalo da se kvarovi sučelja najčešće očituju pri abnormalnim unosima te dovođenjem objekata u stanja koja su malo vjerojatna i samim time se rjeđe pojavljuju. Stoga se pri ispitivanju komponenti preporučuje:

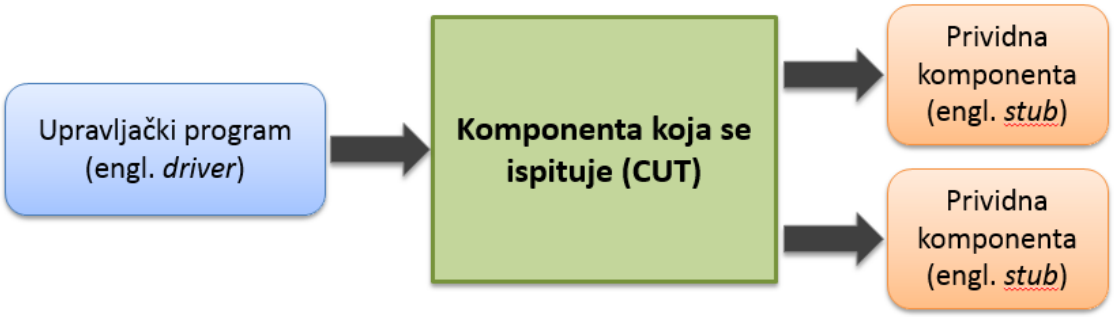
1. obavezno ispitati ponašanje u rubnim uvjetima i malo vjerojatnim situacijama
2. ispitati ponašanje u slučaju pojave NULL pokazivača

3. u paralelnim sustavima ispitati ponašanje pri različitom redoslijedu izvođenja komponenti (npr. sustav s proizvođačima i potrošačima)
4. u vremenski kritičnim sustavima dovesti komponentu pod maksimalno moguće opterećenje.

Pri ispitivanju jedinica ili komponenti čest je slučaj da je komponenta koja se ispituje (engl. *component under test* – CUT, *unit under test* – UUT) ovisna o radu drugih komponenti. Ovisnost može ići u dva smjera: komponenta koja se ispituje može pozivati neku drugu komponentu ili može sama biti pozvana od neke druge komponente. U trenutku ispitivanja komponente moguće je da komponente o kojima je ovisna nisu još implementirane ili pak ne mogu biti na jednostavan i brz način uključene u ispitivanje jedinica (tipičan primjer je baza podataka). Osim toga, ispitivanje je potrebno provesti i za slučaj izvanrednih situacija i situacija s malom vjerojatnošću pojave (npr. osjetljivost na datum/vrijeme).

U tim slučajevima nužno je simulirati rad komponenti koje poziva komponenta koja se ispituje ili koje nju pozivaju, slika 7.4. Takve zamjenske komponente se dijele u dvije skupine:

1. **prividna komponenta** (engl. ***stub***) – simulira rad komponente koja se poziva pomoću objekta koji mora imati isto sučelje te ponašanje kao i komponenta koju simulira, no njegova konkretna implementacija može se značajno razlikovati. U praksi postoje dvije vrste simulacije komponenti: **prividna (krnja) komponenta** (engl. ***stub***) te **imitacijska komponenta** (engl. ***mock***). Prividne komponente vraćaju unaprijed programirane (engl. *hard-coded*) vrijednosti za zadane upite. Imitacijske komponente su „inteligentna“ vrsta prividnih komponenti, podržane u većini razvojnih okruženja, te omogućuju konfiguraciju pri izvođenju ispitnog slučaja. Više o razlici i uporabi prividnih i imitacijskih komponenti moguće je pročitati u [18];
2. **upravljački program** (engl. ***driver***) – simulira objekt koji poziva komponentu koja se ispituje te upravlja njenim izvođenjem.



Slika 7.4 Odnos upravljačkog programa, komponente koja se ispituje i prividne komponente.

7.2.2. Integracijsko ispitivanje

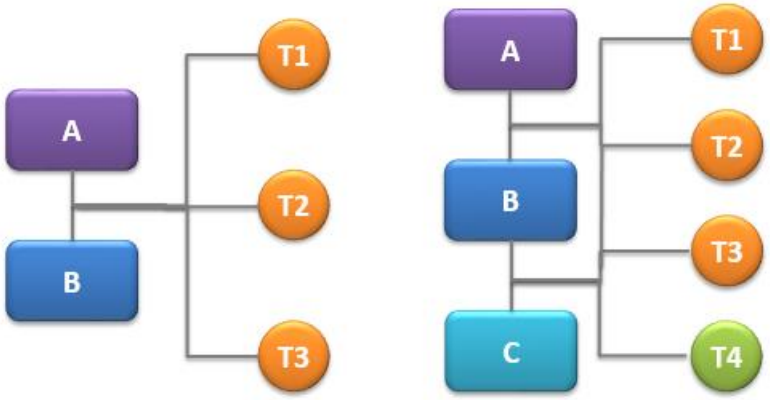
Izgradnja sustava podrazumijeva integraciju svih komponenti od kojih je taj sustav sastavljen. Stoga je iznimno važna njihova međusobna kompatibilnost (vremenski i podatkovno točna interakcija). Ispitivanje sustava započinje ispitivanjem interakcije komponenti temeljem **korisničkih scenarija** (engl. *use-case based testing*). Cilj integracijskog ispitivanja je osigurati zajednički rad grupe komponenti prema specifikaciji zahtjeva. Gotovo svaki specificirani korisnički scenarij implementiran je kroz interakciju dviju ili više komponenti, a sekvencijski dijagrami korišteni pri definiranju funkcionalnih zahtjeva korisni su u ovoj fazi razvoja programske potpore, jer dobro ilustriraju koji su sve objekti i komponente u interakciji. 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.

Ispitivanje sustava temeljeno na integraciji komponenti i njihovom ispitivanju naziva se **integracijsko ispitivanje** (engl. *integration testing*). Moguća su tri pristupa integraciji komponenata:

- **veliki prasak** (engl. *big bang*) – sve komponente se integriraju bez prethodnog ispitivanja. U slučaju pogrešaka problem predstavlja otkrivanje mjesta pogreške. Ne bi se trebalo koristiti u praksi.
- **poboljšani veliki prasak** (engl. *enhanced big bang*) – integracija svih komponenti nakon provedenog pojedinačnog ispitivanja. I dalje ostaje prisutan problem otkrivanja pogreške ako se ona dogodi. U praksi se koristi kod manjka vremena.

- **inkrementalni pristup** (engl. *incremental integration testing*) – integracija i ispitivanje sustava dio po dio. Ovo je uobičajen pristup u praksi koji karakterizira visoka učinkovitost u lokalizaciji mjesta pogreške. Postoje tri podinačice inkrementalnog pristupa:
    - Odozgo na dolje (engl. *top-down testing*) – najprije se razvije kostur sustava koji se onda popunjava komponentama, pri čemu stvarne komponente postupno zamjenjuju prividne komponente koje se koriste umjesto njih. Nije potrebno razvijati upravljačke programe za pojedinačne komponente, već se sustav najčešće ispituje počevši od ulazne lokacije u sustav.
    - Odozdo prema gore (engl. *bottom-up testing*) – najprije se integriraju komponente koje predstavljaju najvažnije i najčešće funkcionalnosti poput pristupa mreži i bazi podataka, a zatim se postupno spajaju i komponente koje izvršavaju ostale funkcionalnosti. Prednost ovog pristupa je što se najvažnije funkcionalnosti puno puta ispituju i što najčešće nije potrebno razvijati prividne komponente.
    - Funkcijska integracija (engl. *sandwich testing*) – komponente se integriraju u konzistentne funkcije bez obzira na hijerarhijsku strukturu. Ovaj pristup je kombinacija pristupa odozgo na dolje i odozdo prema gore i najčešći je postupak u praksi.

  Slika 7.5 ilustrira primjer integracijskog ispitivanja korištenjem skupa ispitnih slučajeva T1 do T4 nad komponentama A, B i C. Ispitni slučajevi T1, T2 i T3 provode se prvi nad minimalnim sustavom koji se sastoji samo od komponenata A i B. Kada se uklone sve pogreške koje otkrivaju ti ispitni slučajevi, u sustav se integrira i komponenta C. Ispitni slučajevi T1 do T3 se ponovno provode kako bi se provjerilo da zbog integracije komponente C nije došlo do neočekivanih interakcija između A i B. Ukoliko ti ispitni slučajevi sada otkriju problem, velika je vjerojatnost da je on uzrokovan dodavanjem nove komponente u sustav. Ukoliko je sve u redu, nad sustavom se provodi i ispitni slučaj T4 kako bi se pokrio novi skup funkcionalnosti i interakcija koje je u sustav donijela komponenta C.



Slika 7.5 Integracijsko ispitivanje komponenti.

Pri ispitivanju sustava tijekom procesa razvoja često se postavlja pitanje „koliko ispitivati?“ Za razliku od ispitivanja jedinica, postupak ispitivanja sustava je vrlo teško automatizirati. Također, nemoguće je ispitati apsolutno sve moguće slučajeve, no treba koristiti zaključke i naputke dobre prakse:

- ispitati sve funkcionalnosti kojima se pristupa kroz izbornike
- ispitati kombinacije funkcija iz istog izbornika (općenito su najčešće problem kombinacije u kojima se kombiniraju rjeđe korištene funcionalnosti sustava)
- ispitati funkcije s ispravnim i neispravnim korisničkim inputom (gdje je primjenjivo).

7.2.3. Ispitivanje sustava prije puštanja u pogon

Ispitivanje sustava prije puštanja u pogon (engl. *release testing*) podrazumijeva ispitivanje konkretne inačice namijenjene distribuciji korisniku. U ovoj drugoj fazi ispitivanja sustava cilj ispitivanja jest provjera ispunjava li sustav postavljene zahtjeve te zadovoljava li očekivane performanse. Za razliku od prethodne razine ispitivanja sustava (integracije), ispitni tim više nema pristup izvornom kodu niti mora biti upoznat s detaljima implementacije, već se sustav promatra kao crnu kutiju (engl. *black box*). Time se povećava vjerojatnost pronalaženja kvarova koje nisu uočili programeri u razvojnom timu.

Ispitivanje udovoljavanja funkcijskim zahtjevima se još naziva i ispitivanje scenarija korištenja (engl. *scenario testing*) i temelji se na sličnom pristupu kao modeliranje obrascima uporabe. Razlika u odnosu na scenarije korištene pri definiranju obrazaca uporabe je u tome što su ispitni scenariji uglavnom mnogo složeniji te obuhvaćaju više zahtjeva (funkcionalnih, nefunkcionalnih te domene primjene) unutar jednog

scenarija. Također, za pisanje ovakvih ispitnih scenarija karakteristično je stvaranje zamišljenih korisnika s različitim ulogama (dobronamjerni, zlonamjerni, stručnjaci u domeni primjene i sl.) te primjena različitih strategija ispitivanja (vidjeti poglavlje 7.4). Ispitivanje na temelju scenarija provodi se prateći sve korisničke scenarije za uporabu sustava koji bi svi trebali uspješno proći. Ponekad prilikom ispitivanja sustava u cjelini izranjaju iznenađujuća svojstva komponenata koja nisu bila vidljiva na razini podsustava. Stoga je ispitivanje cijelog sustava vrlo bitna stavka u ispitivanju prije nego što se sustav pusti u pogon i da na ispitivanje klijentima i krajnjim korisnicima.

Potrebno je naglasiti da pojedina svojstva sustava poput performansi te stupnja pouzdanosti proizlaze tek iz potpuno integriranog sustava. Ona se ispituju nad inačicom sustava koji će biti puštena u pogon u fazi ispitivanja pod nazivom ispitivanje performansi (engl. *performance testing*).

Ispitivanje performansi jest ispitivanje nefunkcionalnih zahtjeva te u praksi obuhvaća:

- Standardne *benchmark* testove brzine sustava i zauzeća memorije pri normalnom radu.
- Specifične ispitne slučajeve koji provjeravaju zadovoljava li sustav određenu predviđenu **razinu opterećenja** (engl. *load testing*) ili opterećenost podacima (engl. *volume testing*). Pritom je iznimno važno stvoriti odgovarajući operacijski profil koji će oponašati tipično stvarno opterećenje (udio poslova/zadataka s određenom vrstom opterećenja).
- Ispitivanje ponašanja sustava pod maksimalnim opterećenjem kada je „gurnut do krajnjih granica“ – tzv. ***stress test*.** To je ispitivanje u kojem se opterećenje na sustav postupno povećava sve dok ne počne izazivati zatajenja sustava. Time se provjera kako se sustav ponaša u stanju zatajenja te postoje li neki defekti koji se pri normalnom radu (opterećenju) ne iskazuju.
- Ispitivanje oporavka sustava (engl. *recovery testing*) – vrsta ispitivanja blisko povezana sa *stress testom* kojom se utvrđuje koliko se učinkovito sustav oporavlja od programskih ili sklopovskih pogrešaka koje su srušile sustav.

7.2.4. Korisničko ispitivanje

Korisničko ispitivanje (engl. *user testing, comparative testing*) posljednja je faza ispitivanja kada ulogu ispitivača sustava preuzimaju krajnji korisnici. U ovoj fazi, određena skupina korisnika ispituje rad sustava u okolini koja odgovara stvarnim

radnim uvjetima te na temelju rezultata daje kritike i prijedloge za poboljšanje sustava. Ispitivanje sustava od strane korisnika neophodno je za uspješan dovršetak razvoja programskog proizvoda, budući da je svaka ispitna okolina osim korisničke zapravo umjetna i ne mora potpuno ocrtavati stvarne radne uvjete, što može imati značajan utjecaj na performanse, pouzdanost, uporabivost i robusnost sustava. Ako korisnici zajedno s ispitnim timom provode ispitivanje i mogu se pratiti njihove reakcije na još nedorađeni proizvod (pri čemu se obično prati koliko su korisnici zadovoljni sa sustavom i koliko ga uspješno koriste), onda se dodatno govori o ispitivanju iskoristivosti (engl. *usability testing*).

Ispitivanje prihvatljivosti (engl. *acceptance testing*) formalna je provjera udovoljava li sustav traženim zahtjevima prije službenog preuzimanja (i plaćanja). Provodi se u slučaju kada se radi o programskom proizvodu koji je jedna tvrtka naručila od druge. Za proizvode koji se slobodno puštaju na tržište u pravilu se ovakvo ispitivanje ne provodi, osim kad je riječ o proizvodima koji moraju zadovoljiti propisane norme i standarde (sigurnost i sl).

Ispitivanje prihvatljivosti složen je proces koji započinje definiranjem kriterija prihvaćanja (engl. *acceptance criteria*) u ugovoru o izradi programskog proizvodu i u privitku toga ugovora. 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. Ispitivanje prihvatljivosti je potrebno pomno isplanirati te predvidjeti potrebne resurse, vrijeme, budžet i konačno raspored ispitivanja. Testovi prihvatljivosti izrađuju se od strane stručnjaka iz domene primjene te u idealnom slučaju trebaju obuhvatiti sve zahtjeve na sustav. Samo izvođenje sustava odvija se najčešće u posebno pripremljenoj korisničkoj okolini te ponekad zahtijeva prethodni mini-trening korisnika koji će ispitivati sustav.

Konačni rezultati ispitivanja uglavnom pokažu određeni broj nedostataka u sustavu. Međutim, zbog uloženog vremena i novca od obje strane (naručitelja i isporučitelja) te mogućnosti poboljšanja u poslovanju koje razvijeni programski proizvod nudi, vrlo rijetko dolazi do potpunog odbacivanja proizvoda. Najčešći rezultat ispitivanja prihvatljivosti jest uvjetno prihvaćanje sustava pri čemu se može definirati minimalni skup funkcionalnosti koji se mora isporučiti odmah, a dodatne „zakrpe“ i održavanje se isporučuju u kasnijim iteracijama.

Instalacijsko ispitivanje (engl. *installation testing*, *configuration testing*) podrazumijeva ispitivanje instalacije programskog sustava u radnoj okolini, a provodi se nakon što je dovršeno ispitivanje prihvatljivosti.

Alfa ispitivanje provodi odabrana skupina korisnika koja usko surađuje s razvojnim timom. Ispitivanje se uglavnom provodi na lokaciji razvoja, no za razliku od ranijih faza ispitivanja, korisnici sudjeluju u stvaranju ispitnih slučajeva, čime oni postaju realističniji. Ovakvo ispitivanje karakteristično je za agilne metodologije (posebno XP). Alfa ispitivanje najčešće uključuje ispitivanje programske potpore na manjem broju predviđenih klijentskih konfiguracija.

Beta ispitivanje najčešće znači da se programski proizvod (sustav) pušta u javnost kako bi se ispitao u stvarnim uvjetima u kojima će se koristiti. Ova vrsta korisničkog ispitivanja česta je u slučajevima kada će se sustav koristi u mnogo različitih okolina (programski proizvodi opće namjene). Naglasak ispitivanja je na interakciji između programskog proizvoda i okoline (sklopovlja i ostale programske potpora) u kojoj se koristi. Nakon što je utvrđeno da je programski proizvod uspješno prošao ispitivanje instalacije, može ga se bez ograničenja dalje koristiti. Ponekad se beta ispitivanje smatra i oblikom marketinga, budući da se beta inačica komercijalnog proizvoda (koji se plaća) u pravilu pušta u javnost besplatno.

7.3. Organizacija ispitivanja

Organizacija procesa ispitivanja ovisi o svojstvima programske potpore i o modelu procesa programskog inženjerstva koji je primijenjen u razvoju programske potpore. Odluka o tome u kojim fazama razvoja se ispitivanje provodi kroz povijest se rješavala na razne načine.

7.3.1. Evolucija koncepta ispitivanja programske potpore

Prema vrsti i načinu ispitivanja te trenutku u procesu razvoja programske potpore moguće je identificirati šest povijesnih razdoblja tijekom kojim je evoluirao koncept ispitivanja:

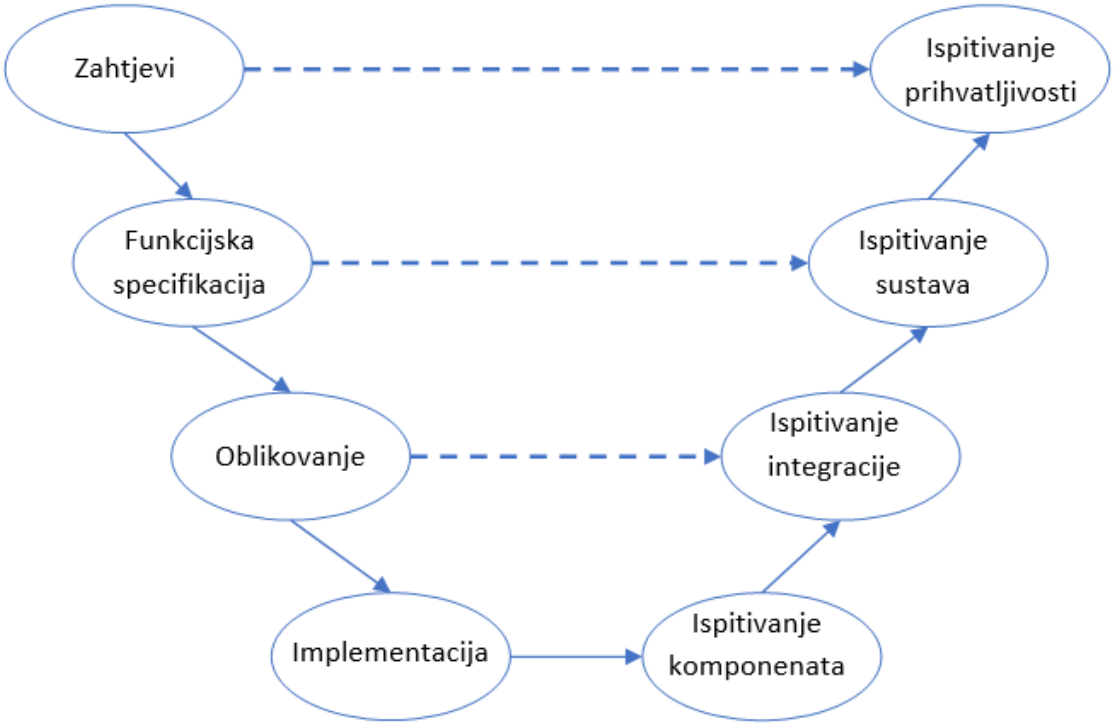
- **Do 1956. – ispravljanje pogrešaka** (engl. *debugging*). Ispitivanje se svodilo na provjeru rada programa u izvođenju i ispravljanje pogrešaka koje se uočilo (engl. *check-out & debugging*) bez sustavnog planiranja procesa ispitivanja.
- **1957.–1978. – demonstracija ispravnog rada programa** (engl. *demonstration*). Ispitivanje je trebalo pokazati da program radi ispravno za sve tipične ulaze u skladu sa specifikacijom. U ovoj fazi prvi puta se pojavljuje razdvajanje faze razvoja programa od faze ispravljanja pogrešaka, koje je slijedilo nakon razvoja programa.
- **1979.–1982. – namjerno izazivanje zatajenja** sa svrhom otkrivanja pogrešaka kao nadogradnja druge faze usmjerena na razaranje (engl. *destructive*).

Programska potpora se ispitivala s ispravnim i s neispravnim ulaznim podacima i korištenjem te se pratilo što se događa. Ispitivanje se smatralo uspješnim ako se otkrilo zatajenje.

- **1983.–1987. – evaluacija** (engl. *evaluation*) svih dijelova projekta. Ispitivanje se po prvi put smatralo dijelom generičke aktivnosti validacije i verifikacije. Zadatak je bio otkriti pogreške u zahtjevima, oblikovanju i implementaciji.
- **1988.–2000. – prevencija pojave pogrešaka u projektu** (engl. *prevention*). Ispitivanje se smatralo jednim od načina kako pokušati izbjeći pogreške. Prevencija pojava pogrešaka pokušala se ostvariti u zahtjevima, oblikovanju i implementaciji.
- **2000.–danas – razvoj programske potpore temeljen na ispitivanju** i razvoj vođen ispitivanjem (engl. *test-driven development*). Ovakav razvoj rezultira programskim kodom koji je podoban za ispitivanje.

7.3.2. V-model ispitivanja

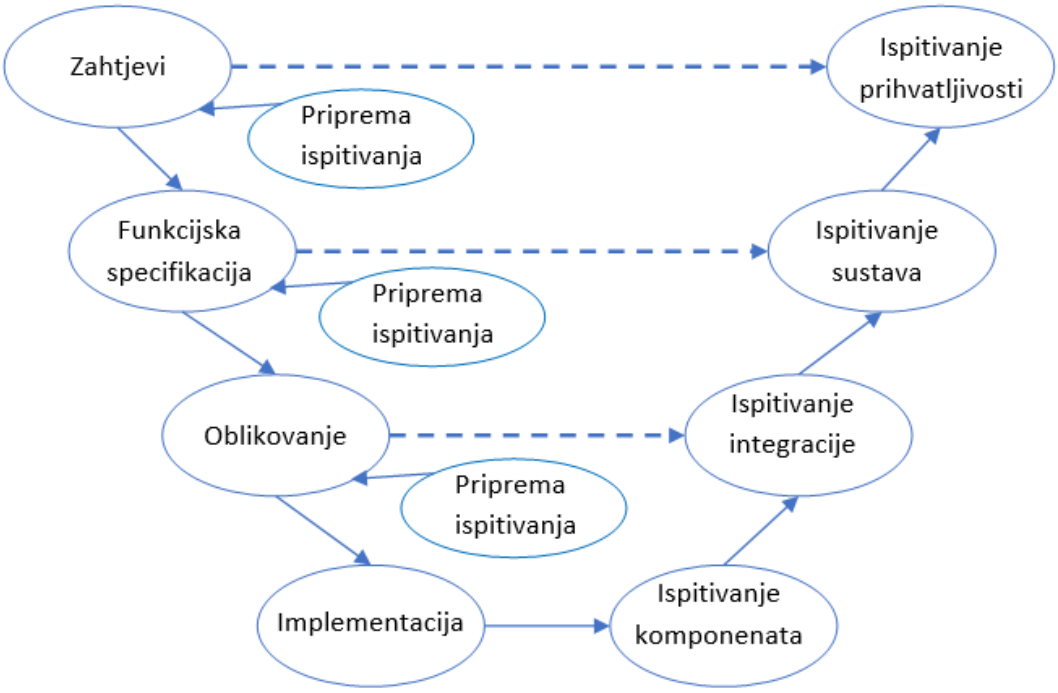
Na slici 7.6 prikazan je klasični **V-model ispitivanja** koji podrazumijeva izradu sustava od specifikacije do krajnjeg koda nakon čega slijedi ispitivanje od razine komponenata do ispitivanja prihvatljivosti. U povijesnom smislu, V-model bi se mogao poistovjetiti s prva četiri razdoblja ispitivanja budući da u V-modelu nema predviđenog planiranja ispitivanja ili razvoja programske potpore s namjenom da se kasnije lakše ispita.



Slika 7.6 V-model ispitivanja programske potpore.

7.3.3. V-model ispitivanja s ranom pripremom

Na slici 7.7 prikazan je **V-model ispitivanja s ranom pripremom**. Za razliku od klasičnog V-modela, model s ranom pripremom predviđa razvoj ispitnih slučajeva od samog početka razvoja projekta. Taj proces uključuje detaljni opis načina na koji će se zahtjevi ispitati, načina ispitivanja sustava u cjelini na temelju funkcionalne specifikacije te načina ispitivanja arhitekture sustava i integracijskog ispitivanja. Iako se suštinski ne razlikuje od klasičnog V-modela, V-model s ranom pripremom ipak donosi poboljšanja u razvoj programske potpore u smislu da ispitni tim treba unaprijed predvidjeti što se sve treba ispitati kako bi se osiguralo poboljšanje kvalitete proizvoda.

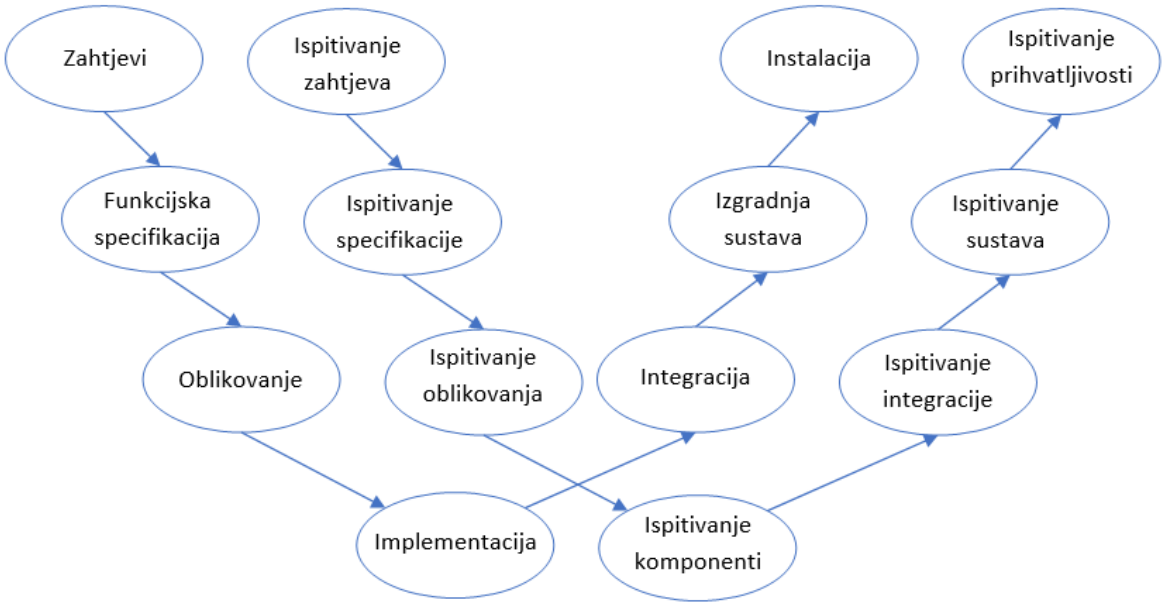


Slika 7.7 V-model ispitivanja programske potpore s ranom pripremom.

7.3.4. W-model ispitivanja

Na slici 7.8 prikazan je W-model ispitivanja programske potpore koji u novije vrijeme sve češće u praksi zamjenjuje V-model. Osnovna ideja W-modela jest provedba ispitivanja u paraleli sa svakom aktivnosti tijekom životnog ciklusa programske potpore: od specifikacije do izgradnje čitavog sustava. To znači da se sve što se napiše ili implementira ujedno i što prije provjeri, koliko god je to moguće iscrpno.

Može se primijetiti da u ovom modelu postoje dva paralelna V-modela, od kojih se jedan bavi samo s ispitivanjem. Tako se ispituju i propituju korisnički zahtjevi, razvijaju se rani prototipovi kako bi se ispitala funkcijska specifikacija, ispituju se modeli arhitektura sustava i zatim se prelazi na ispitivanje koda: od ispitivanja komponenti pa sve do ispitivanja prihvatljivosti te alfa i beta ispitivanja. W-model je blisko povezan s razvojem vođenim ispitivanjem i predstavlja poveznicu između klasičnog V-modela i V-modela s ranom pripremom sa šestim razdobljem razvoja ispitivanja, a to je razvoj vođen ispitivanjem.



Slika 7.8 W-model ispitivanja programske potpore.

7.3.5. Razvoj vođen ispitivanjem

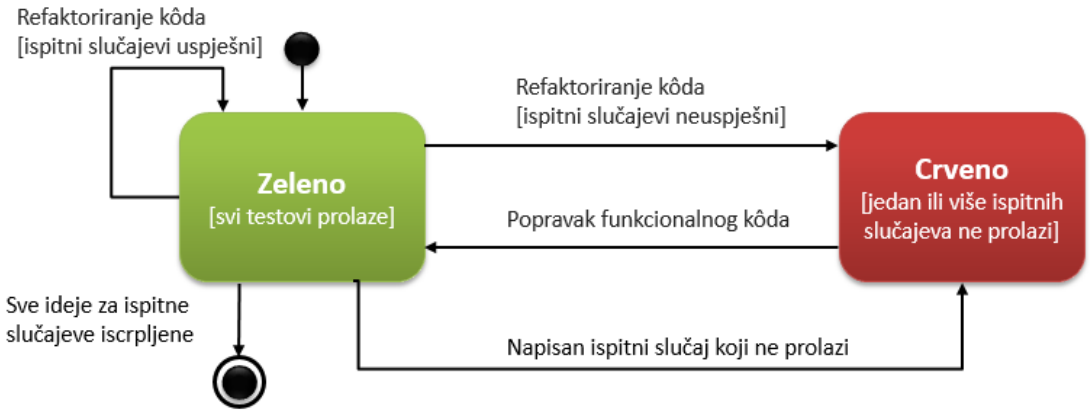
Razvoj vođen ispitivanjem (engl. *Test-driven Development,* TDD) podrazumijeva pristup razvoju programske potpore u kojem se ispreplićу faze razvoja (kodiranja) i ispitivanja. To znači da se za svaki inkrement koda koji se napiše, mora napisati i skup ispitnih slučajeva koji će ispitati ispravan rad tog inkrementa. Sve dok razvijeni kôd ne prođe ispitivanje, ne razvija se novi kôd. TDD se koristi najviše u agilnim metodama razvoja programske potpore (npr. ekstremno programiranje). Slika 7.9 ilustrira proces TDD-a.

Kod procesa TDD-a iznimno je važno postojanje automatiziranog ispitnog okruženja poput radnih okvira JUnit i NUnit, budući da se kôd razvija u vrlo malim inkrementima i svaki put kad se razvije novi inkrement moraju se provesti svi postojeći testovi. Automatizacija ispitivanja omogućava provođenje stotina ispitnih slučajeva u svega nekoliko sekundi.

Ovakav pristup razvoju programske potpore ima svoje prednosti i nedostatke. Najveće prednosti su:

1. TDD olakšava programerima razumijevanje načina na koji bi razvijeni komad kôda trebao funkcionirati
2. dobra pokrivenost kôda ispitnim slučajevima

3. olakšano otkrivanje kvarova
4. dokumentiranje funkcionalnosti sustava – ispitni slučajevi jasno prikazuju što koji dio sustava radi te služe kao oblik dokumentacije
5. olakšano (i ubrzano) regresijsko ispitivanje nakon što su učinjene promjene u sustavu.



Slika 7.9 Razvoj vođen ispitivanjem.

Mane TDD-a iskazuju se pri nadogradnji postojećih sustava te ponovnog korištenja postojećih komponenti za koje ne postoje ovakve vrste ispitnih slučajeva. U tom slučaju moraju se naknadno pisati ispitni slučajevi i ne postoji jamstvo da će pokriti sve postojeće funkcionalnosti. Također, TDD je vrlo problematičan kod višedretvenog izvođenja aplikacija kada se dretve drukčije isprepliću pri svakom novom izvođenju te svako izvođenje može proizvesti drukčije rezultate.

Važno je naglasiti da TDD ne može zamijeniti ispitivanje sustava u klasičnom smislu kada se provjerava udovoljavanje sustava funkcionalnim i nefunkcionalnim zahtjevima, no u ranim fazama razvoja programske potpore može smanjiti broj sitnih pogrešaka u malim dijelovima koda koje u kasnijim fazama integracije mogu uzrokovati velike pogreške u sustavu, a tada ih je i mnogo teže pronaći.

7.4. Strategije i pristupi ispitivanju

Prilikom planiranja ispitivanja, ispitivač treba razmotriti veći broj različitih strategija koje može primijeniti, u ovisnosti od opsežnosti i zahtjeva projekta, vremena koje mu je na raspolaganju i prijašnjeg iskustva. U nastavku ovog poglavlja navodi se deset strategija i pristupa koji se uobičajeno primjenjuju u praksi.

7.4.1. Ispitivanje ad hoc

Kod **ispitivanja ad hoc** (engl. *ad-hoc testing*) nema unaprijed definiranih scenarija uporabe sustava koji se ispituje niti unaprijed određenih ispitnih slučajeva. To je najmanje formalna strategija ispitivanja programske potpore, a temelji se na vještinama, iskustvu i intuiciji ispitivača koji pri ispitivanju ima potpunu slobodu. Nema posebnog planiranja i dokumentiranja, a ispitni slučajevi nisu predviđeni za ponavljanje ako se dobije očekivani odgovor od strane ispitivanog sustava. S obzirom da ovakav pristup nije ni u kojem obliku strukturiran, nema temelja za reprodukciju primijećenih zatajenja, ali prednost pristupa može biti u tome da se važnije pogreške mogu brzo otkriti.

Iako se uvelike temelji na improvizaciji, mogu se navesti neke pretpostavke i niti vodilje ovog pristupa. Može se npr. relativno brzo identificirati dio sustava koji je više podložan zatajenjima i koji bi mogao biti izvor većeg broja zatajenja i nestabilnosti sustava. Tada bi fokusiranje na njega bilo isplativo i za buduća ispitivanja.

Sposobnost uživljavanja razvojnog inženjera u funkcioniranje sustava je prednost kod ovakvog tipa ispitivanja kako bi se domislilo koje bi bile slabe točke sustava. Osim toga, dobro poznavanje sustava koji se ispituje je prednost kod smišljanja ispitnih scenarija ad hoc. Iskustvo ispitivanja i onih različitih ali i onih sličnih sustava (po strukturi i domeni) na strani ispitivača je također prednost kod načina razmišljanja o mogućim slabostima ispitivanog sustava. To se odnosi i na iskustvo poznavanja različitih pomoćnih alata s kojima se ispitivanje provodi. S boljim poznavanjem mogućnosti alata, njihov izbor i uporaba su brži i smisleniji. Ispitivanjem ad hoc daje prvi uvid u sustav i potencijalna mjesta zatajenja što poslije pomaže u određivanju rasporeda i prioriteta razvoja i ispitivanja te opsega i trajanja ispitivanja pojedinih dijelova sustava. Nakon popravaka prvotno pronađenih pogrešaka, ispitivanja ad hoc se mogu primjenjivati upravo na mjestima tih popravaka.

Nedostatci su u očitoj neorganiziranosti ispitivanja, riziku promašaja temeljnih funkcionalnosti kod ispitivanja, nemogućnosti predviđanja pokrivenosti ispitivanja kao i pokušaja reprodukcije pogrešaka.

7.4.2. Pogađanje pogrešaka ili ispitivanje zasnovano na pogreškama

Za strategiju **pogađanja pogrešaka** (engl. *error guessing*) ili pristup **ispitivanju zasnovanom na pogreškama** (engl. *error-based testing*) može se ponoviti većina toga rečenog za ispitivanje ad hoc, ali u osnovi ovo je nešto studiozniji pristup. Također se u principu temelji na vještini, intuiciji i iskustvu ispitivača, ali s postojanjem nekog plana ispitivanja. To se prije svega odnosi na smišljanje i pisanje

ispitnih slučajeva koji se temelje na uvidu u povijest već pronađenih pogrešaka u sustavu i na provjeri najčešćih i tipičnih uzroka pogrešaka za pojedine tipove sustava.

U ovom pristupu podrazumijeva se izvrsno poznavanje sustava iznutra: izvornog koda, načina ostvarenja pojedinih dijelova sustava, konteksta i okoline rada sustava, posebnih uvjeta koji se mogu odnositi na funkcioniranje operacijskog sustava, sustava baze podataka i poslužitelja. Nadalje, više se podrazumijeva poznavanje koncepata i situacija koji su najčešći izvori tipičnih pogrešaka te se ispituju potencijalne lokacije kvarova koje dovode do preljeva međuspremnika (engl. *buffer overflow*), odstupanja od granica podatkovnog polja, neispravnog korištenja iteratora, dodjela neispravnih i nedozvoljenih vrijednosti pokazivačima (engl. *null pointer assignent*), dijeljenja s nulom (engl. *division by zero*), postavljanja nedozvoljenih vrijednosti parametara, itd.

7.4.3. Istraživačko ispitivanje

Istraživačko ispitivanje (engl. *exploratory testing*) predstavlja još jedan korak naprijed u smislu udubljivanja u domenu ispitivanog sustava u odnosu na ispitivanje ad hoc i ispitivanje pogađanjem pogrešaka. **Istraživačko ispitivanje** je simultano učenje o sustavu kroz izradu i provedbu ispitnih slučajeva i interpretaciju rezultata kao međusobno podupirajućih aktivnosti. Između tih aktivnosti postoji, uz slobodu ispitivača, i znatan dinamizam kojim se upotpunjuje znanje ispitivača o sustavu i time unapređuje kvalitetu ispitivanja. Ova strategija se često koristi u agilnom procesu razvoja programske potpore. Sam pojam "istraživanja" u sebi pretpostavlja otkrivanje nečeg novog pa se ovaj tip ispitivanja i ne smatra nužno "tehnikom ispitivanja" već "načinom razmišljanja" koje bi se moglo primijeniti na bilo koju tehniku ispitivanja. Ova strategija ispitivanje predstavlja suprotnost ispitivanju temeljenom na pisanju skripti, gdje se ispitni slučajevi pripremaju unaprijed s predefiniranim koracima ispitivanja i očekivanim rezultatima. Te skripte poslije može koristiti netko tko uopće nema znanja o domeni sustava te samo doslovno uspoređivati dobivene s očekivanim rezultatima (čime se dolazi na trag automatizacije ispitivanja). Kod istraživačkog ispitivanja, očekivanja su skoro potpuno otvorena, neki rezultati se mogu predvidjeti, a neki ne, a ispitni primjeri se smišljaju usput.

Prednost istraživačkog ispitivanja je u potrebi manjih priprema za pristupanje ispitivanju te je svakako dinamičnije i intelektualno izazovnije nego druge strategije ispitivanja. Veće greške se brzo otkrivaju, ali nema ozbiljnije dijagnostike te se gubi mogućnost ponavljanja načina i redoslijeda ispitivanja te detaljnijeg razlikovanja ispitnih slučajeva po zatajenjima koje pokrivaju. Pristup je stoga više prikladan kod

nepotpunih zahtjeva i specifikacija ili manjka vremena, što je česti slučaj kod agilnog pristupa razvoju programske potpore.

7.4.4. Sustavno ispitivanje

Sustavno ispitivanje (engl. *systematic testing*) je uređeno, plansko i izrazito metodičko ispitivanje. U suprotnosti je s dosada navedenim tipovima ispitivanja (ad hoc, pogađanje pogrešaka, istraživačko ispitivanje) jer pretpostavlja duboku i detaljnu analizu sustava koji se ispituje i njegovih komponenti na vrlo niskoj razini granularnosti. Uključuje analizu podjele sustava na particije, analizu rubnih vrijednosti, kombinacija vrijednosti, stanja i prijelaza automata koji opisuju rad sustava iznutra i svih puteva grafa tijeka izvođenja (vidjeti poglavlje 7.5), međutim, ono nije nužno vezano uz tehniku strukturnog ispitivanja. Radi se o strategiji sveobuhvatne i eksplicitne analize svih aspekata sustava s ciljem njihova detaljnog i savjesnog ispitivanja.

7.4.5. Mutacijsko ispitivanje

Mutacijsko ispitivanje (engl. *mutation testing*) temelji se na manjim promjenama programa koji se ispituje. Traže se izvedbe programa koje su krive, ali što sličnije njegovom izvornom obliku da bi se pronašle pogreške koje bi inače bilo teško otkriti uobičajenom analizom sintakse i semantike. Svaka promijenjena inačica programa se naziva mutantom koji ispitni slučajevi potvrđuju ili odbacuju, ovisno o tome razlikuje li se ponašanje promijenjene inačice od izvorne ili ne. Broj odbačenih (tzv. ubijenih) mutanata koji uzrokuju različito ponašanje programa u odnosu na njegovu izvornu inačicu je mjera vrednovanja pojedinih testova. Teži se tome da sve promjene budu otkrivene ispitnim slučajevima.

Tradicionalni pristup mutacijskom ispitivanju temelji se na teoriji koja sužava prostor generiranja mutanata na dostatnu mjeru da bi ispitivanje bilo i izvedivo i smisleno u isto vrijeme. Ova teorija počiva na dvije hipoteze:

- hipotezi kompetentnog programera (engl. *competent programmer hypothesis*)
- učinku međuovisnosti (engl. *coupling effect*).

Hipoteza kompetentnog programera tvrdi da su programeri kompetentni, što znači da teže razvoju programa koji su vrlo blizu ispravnim inačicama istih. To znači da, iako u razvijenim programima može biti pogrešaka, uzima se pretpostavka da su

to u pravilu sitne pogreške koje se mogu popraviti s nekolicinom sintaktičkih promjena. Stoga se pod ovom pretpostavkom u mutacijskom ispitivanju rade manje sintaktičke promjene koje predstavljaju pogreške koje radi "kompetentni programer".

Hipoteza učinka međuovisnosti bavi se vrstama i opsegom pogrešaka. U teoriji, jednostavna pogreška je predstavljena jednostavnim mutantom dobivenim s manjom sintaktičkom promjenom, dok je složena pogreška predstavljena složenim mutantom dobivenim s uvođenjem više od jedne sintaktičke promjene. Hipoteza tvrdi da postoji učinak međuovisnosti (povezanosti) između jednostavnijih i složenijih pogrešaka u programu i da ispitni slučajevi koji otkrivaju jednostavne pogreške mogu otkriti i visok postotak složenih. Zaključak ove hipoteze je da je dovoljno stvarati jednostavne mutante izvornih programa.

Neke od osnovnih vrsta mutacija su:

- mutacije vrijednosti koje mijenjaju vrijednosti konstanti i parametara, npr. granica petlji
- mutacije odluka koje mijenjaju uvjete odluka grananja kao npr. kod operatora "manje od" ('<') i "veće od" ('>')
- mutacije izraza koje izostavljaju pojedine linije, mijenjaju redoslijed linija izvornog koda ili mijenjaju operacije u aritmetičkim izrazima.

7.4.6. Ispitivanje zasnovano na kvarovima

Pristup **ispitivanju zasnovan na kvarovima** (engl. *fault-based testing*) pretpostavlja definiranje ispitnih slučajeva koji omogućavaju otkrivanje pogrešaka i to tako da se u programski kôd umjetno ubacuju kvarovi (engl. *error-seeding*). Nakon toga, provođenjem ispitnih slučajeva određuje se u kojoj mjeri ih ispitivanje otkriva. Ispitivanje zasnovano na kvarovima slično je mutacijskom ispitivanju, međutim mutacijsko ispitivanje se smatra sistematičnijim, budući da ono sintaktički i semantički mijenja točno određene dijelove programa (kao što je opisano u prethodnom poglavlju 7.4.5), dok ispitivanje zasnovano na kvarovima nema iza sebe razrađenu teoriju, već umjesto toga samo služi razvojnom inženjeru da istražuje u kojoj mjeri ispitni slučajevi otkrivaju umjetne kvarove te da time stiče bolji uvid u ispravno funkcioniranje programa.

7.4.7. Slučajno ispitivanje

Pojam "slučajnosti" je suprotan i podrazumijeva odsustvo bilo kakve "sustavnosti". Tako se kod strategije **slučajnog ispitivanja** (engl. *random testing*) u pravilu radi o generiranju slučajnih nezavisnih ulaznih vrijednosti kao temelja ispitivanja. Slučajno ispitivanje koristi se kod tehnike funkcijskog ispitivanja (ili ispitivanje "crne kutije") gdje se rezultati ispitivane funkcije uspoređuju s vrijednostima dobivenim iz specifikacije (ili ugovora) ispitivane funkcije.

Prednosti ovog pristupa ispitivanju su niska cijena provedbe ispitivanja i ne nužnost poznavanja funkcionalnosti koju se ispituje, što znači da je pristup manje vođen potencijalno krivim pretpostavkama ispitivača. Ispiti su u pravilu kratki te je takav pristup dobar u brzom lociranju mjesta pogreške, pogotovo kada je sustav dobro specificiran.

Podaci koji se dovode kao ulazi u ispitivanju mogu se generirani s različitim udjelima elementa "slučajnosti", sa ili bez pomoćne heuristike koja bi usmjeravala taj proces. Cilj je proizvesti valjane ispitne skupove bez zalihosti, korištenjem različitih tehnika. Slučajni izbor ulaznih podataka može se ostvariti i iz predefinirane baze podataka, a slučajan može biti i slijed poziva metoda.

7.4.8. *Fuzz* ispitivanje

***Fuzz* ispitivanje** je često korištena strategija za provjeru sigurnosti programske potpore. To je prije svega funkcijsko ispitivanje (vidjeti poglavlje 7.5) u kojem se sustav ispituje dovođenjem nevaljanih, neočekivanih i slučajnih ulaza, čime se pokušava dovesti u neispravno stanje. Cilj je otkriti ranjivosti sustava i često se ovaj pristup smatra kontrolom kvalitete sustava radije nego tehnikom ispitivanja u svrhu pronalaženja pogrešaka. U najjednostavnijoj varijanti, *fuzz* ispitivanje postupa jednako kao i slučajno ispitivanje – dovodi niz potpuno slučajno generiranih znakova na ulaz ispitivanog sustava bez znanja o strukturi sustava. U složenijim slučajevima, *fuzz* ispitivanje se radi tako da se valjani ulazni podaci selektivno mijenjaju ili se stvaraju blokovi potpuno novih podataka temeljenih na predefiniranom modelu sučelja između sustava.

7.4.9. Ispitivanje ekvivalentnosti particija

Strategija **ispitivanja ekvivalentnosti particija** (engl. *partition testing, equivalence partitioning*) odnosi se na tehniku u kojoj se domena ispitivanja ulaznog

podatka dijeli na dvije ili više poddomena. Ispitni slučajevi se tada uzimaju iz tih poddomena tako da je svaka zastupljena s barem jednim ispitnim slučajem. Naprimjer, ako je ulaz ispitivanog programa jedna cjelobrojna vrijednost, tada se domena cjelobrojnih vrijednosti može podijeliti na poddomene pozitivnih i negativnih cjelobrojnih vrijednosti. Minimalni skup ispitnih slučajeva koji pokrivaju cjelokupnu domenu ulaznih vrijednosti se tada može sastojati od jednog pozitivnog i jednog negativnog cijelog broja.

Kod ispitivanja particija, ideja je zajedno grupirati slične ispitne slučajeve te podijeliti prostor ulaznih vrijednosti u **ekvivalentne razrede** (particije) kako bi se smanjio broj ispitnih slučajeva kojima treba ispitati neki dio koda. Pritom bi trebale postojati jasne relacije između tih razreda i njihovih mjera učestalosti pogrešaka. Podjela bi trebala biti napravljena s idejom vodiljom o tome gdje i kako se pogreške mogu pojaviti da bi po tome, u idealnom slučaju, svi ispitni slučajevi koji su uzročnici pojave neke greške bili pravilno raspoređeni u odgovarajuće razrede. Tada bi izbor samo jednog predstavnika razreda prokazao određenu pogrešku.

U stvarnosti su, međutim, moguće vrlo različite varijante podjele. Podjele se mogu temeljiti na funkcionalnim zahtjevima pa bi se svaka funkcionalnost mogla promatrati kao posebna domena za ispitivanje. S druge strane, strukturni kriteriji podjele mogu se temeljiti na pokrivenosti izraza, grananja i putova pa bi podjela mogla razlikovati sve konstrukte koji se ispituju (npr. generirati barem jedan ispitni slučaj za pokriti svaku granu prilikom grananja programa).

7.4.10. Ispitivanje zasnovano na pokrivenosti

Pristup **ispitivanju zasnovan na pokrivenosti** (engl. *coverage-based testing*) temelji se na tome da su zahtjevi ispitivanja specificirani u odnosu na neku mjeru pokrivenosti ispitnog programa ispitnim slučajevima (engl. *code coverage*). Ispitni slučajevi osmišljavaju se tako da postignu što veću vrijednost zadane mjere (metrike), a barem onu koja je zadana specifikacijom. Ovaj pristup uglavnom se koristi kod tehnike strukturnog ispitivanja (poglavlje 7.5). Najčešće metrike su, redom prema učestalosti pojavljivanja u praksi:

1. pokrivenost linija koda
2. pokrivenost broja grana u programu
3. pokrivenost broja putova kroz program
4. postotak programskih elemenata (funkcija, atributa i dr.) koji su izvedeni/mijenjani.

Prednost ovog pristupa je što postoji mjera koja govori u kolikoj mjeri je ispitan sustav. S druge strane, glavni nedostatak je pojednostavljenost problema pronalaska pogreške ako se koristi ovaj pristup. Naime, sam prolazak ispitnog slučaja kroz kôd ne znači da će pogreška sigurno biti otkrivena. Primjerice, ispitni slučaj možda nije pronašao kvar iako je prošao određenom linijom koda, budući da vrijednost neke varijable nije bila ona koja bi izazvala pogrešku. Drugim riječima, npr. 90% pokrivenih linija koda ispitivanjem ne znači da je program gotovo potpuno ispitan. Dapače, čak niti 100% pokrivenih linija koda ne jamči odsutstvo pogrešaka.

Pojam koji je blisko povezan s pojmom pokrivenosti je pojam **potpunog ili iscrpnog ispitivanja** (engl. *complete testing, exhaustive testing*). Ako razvojni inženjer uspije provesti potpuno ispitivanje nekog dijela programa, onda on može opravdano tvrditi da u njemu nema pogrešaka. Međutim, potpuno ispitivanje uključuje:

- ispitivanje svih mogućih vrijednosti varijabli (ulazne/izlazne/međuvarijable)
- ispitivanje svih mogućih kombinacija vrijednosti više ulaznih varijabli
- ispitivanje svih mogućih sekvenci izvođenja programa (svih putova izvođenja)
- ispitivanje svih mogućih HW/SW konfiguracija (uključujući i one sustave koje nemamo na raspolaganju)
- ispitivanje svih mogućih načina korisničke uporabe programa (npr. brzina utipkavanja novih naredbi koje se prenose programu).

Iz tih razloga, osim za vrlo kratke i jednostavne funkcije, u praksi nije moguće potpuno ispitati program. Kao klasični primjer granica potpunog ispitivanja navodi se američki sustav MASPAR (*Massively Parallel Computer*, do 64k paralelnih procesora) [19] kojemu je ispitivan skup instrukcija. Od svih instrukcija, najteže je bilo ispitati radi li ispravno računanje korijena 32-bitnog cijelog broja, koji ima 4.294.967.296 vrijednosti, budući da je ta instrukcija imala najviše mikrokoraka. Test koji je osmišljen i proveden trajao je 6 minuta (i koristio je procesorsku moć svih procesora) te je utvrdio postojanje 2 pogreške pri računanju (od svih 4 milijarde vrijednosti). Kvar je bio opskuran, no uspješno je pronađen i uklonjen. Pokazuje se da bi za ustanovljavanje je li korijen 64-bitnog cijelog broja ispravno izračunat na istoj konfiguraciji ispit potrajao 49029 godina (a danas nešto kraće).

7.5. Tehnike ispitivanja

Tehnike ispitivanja odnose se na tri moguća načina na koje je moguće pristupiti ispitivanju, u ovisnosti od toga imamo li pristup kodu ili ne, te ako nemamo pristup,

imamo li saznanja o internom funkcioniranju komponente. Tako tehnike ispitivanja dijelimo na:

- funkcijsko ispitivanje – nema pristupa kodu
- strukturno ispitivanje – postoji pristup kodu
- ispitivanje sive kutije – nema pristupa kodu, ali imamo saznanja o internom funkcioniranju.

U nastavku su detaljnije opisane sve tri tehnike ispitivanja.

7.5.1. Funkcijsko ispitivanje

Funkcijsko ispitivanje (engl. *black-box testing, functional testing, specification-based testing*) je tehnika ispitivanja kod koje ispitivač nema dostupan programski kôd sustava koji ispituje, već na sustav gleda kao na crnu kutiju koju treba ispitati. Za konkretnu funkciju koju se treba provjeriti dostupan je samo njezin potpis – deklaracija naziva funkcije, povratnog tipa i tipova argumenata te očekivano ponašanje.

Ispitivač se koncentrira samo na ulazno-izlazno ponašanje komponente i ispituje ju samo prema zahtjevima i specifikaciji. Temeljna pretpostavka funkcijskog ispitivanja je da za ulazne podatke možemo predvidjeti izlaz koji je najčešće zadan specifikacijom. Ako se izlaz ne može predvidjeti, onda jedino ostaje ispitati da li komponenta za neke ulazne vrijednosti javlja pogrešku u izvođenju ili uzrokuje zatajenje sustava.

Ulazne vrijednosti mogu se generirati različitih strategijama i pristupima ispitivanja, kao što se može pročitati u poglavlju 7.4, primjerice slučajnim ispitivanjem, ispitivanjem ekvivalencije particija i sl. U praksi, kod funkcijskog ispitivanja cilj je smanjiti broj ispitnih slučajeva. To se najbolje postiže ekvivalentnom podjelom ulaznih vrijednosti i analizom graničnih vrijednosti. Ispitni slučajevi oblikuju se odabirom **ekvivalentnih particija** (vidjeti poglavlje 7.4.9) za ispitivanje, pri čemu se particije dijele na one koje sadržavaju valjanu vrijednost i one koje sadržavaju nevaljanu vrijednost. Ako su vrijednosti ulaza valjane u nekom intervalu, tada se odabiru tri ispitna slučaja, i to s:

- vrijednostima ispod intervala
- vrijednostima u intervalu
- vrijednostima iznad intervala.

U svakom slučaju, potrebno je ispitati i rubne vrijednosti, jer su one obično najkritičnije za pogreške u programima. Stoga se ispituju rubne vrijednosti s obje strane doljnje i gornje granice dozvoljenog intervala (dodatna četiri ispitna slučaja), vidi sljedeći primjer.

Primjer 7.4. Za dozvoljeni ulaz funkcije čija varijabla treba biti četveroznamenkasti broj (u rasponu od 1000 do 9999), potrebno je napisati ispitne slučajeve (minimalne) za funkcijsko ispitivanje.

Rješenje:

1) ulaz: neki broj manji od 1000, npr. 500 – očekivani izlaz: NE;

2) ulaz: 999 – očekivani izlaz: NE;

3) ulaz: 1000 – očekivani izlaz: DA;

4) ulaz: neki dozvoljeni broj u intervalu, npr. 5000 – očekivani izlaz: DA;

5) ulaz: 9999 – očekivani izlaz: DA;

6) ulaz: 10000 – očekivani izlaz: NE;

7) ulaz: neki broj veći od 10000, npr. 12000 – očekivani izlaz: NE.

Kod funkcijskog ispitivanja provode se sljedeći koraci:

1. odrediti particije za sve ulazne varijable
2. za sve particije odabrati vrijednosti ispitivanja (uključujući rubne slučajeve)
3. definirati ispitne slučajeve (ulaz i očekivani izlaz) koristeći odabrane vrijednosti
4. provesti ispitivanje i vrednovati rezultate.

Funkcijsko ispitivanje posjeduje svojstvo kombinatorne eksplozije ispitnih slučajeva koja se događa zato što je potrebno ispitati i valjane i nevaljane podatke. U slučaju kada na ulazu u funkciju imamo više varijabli čije intervale trebamo uzeti u obzir, stvari se pogoršavaju. Ovo možemo vidjeti u sljedećem primjeru koji demonstrira kako funkcijsko ispitivanje smanjuje problem klasičnog **kombinacijskog ispitivanja** (engl. *combinatorial testing*), ali mu je i dalje potreban velik broj ispitnih slučajeva.

Primjer 7.5. Funkcija prima na svom ulazu dva dvoznamenkasta broja (raspon od 10 do 99) i određuje koji je broj veći. Odredite minimalne ispitne slučajeve za funkcijsko ispitivanje ove kombinacije.

Rješenje:

1) Ulaz: prvi broj manji od 9, drugi broj manji od 9 – očekivani izlaz: POGREŠKA

2) Ulaz: prvi broj: 9, drugi broj bitno manji od 10 – očekivani izlaz: POGREŠKA

3) Ulaz: prvi broj: 10, drugi broj bitno manji od 10 – očekivani izlaz: POGREŠKA

...

49) Ulaz: prvi broj veći od 100, drugi broj veći od 100 – očekivani izlaz: POGREŠKA

Ukupno imamo najmanje 49 ispitnih slučajeva. Klasičnim kombinacijskim ispitivanjem svih dozvoljenih vrijednosti imali bismo 90 x 90 = 8100 ispitnih slučajeva. Možemo utvrditi da smo pomoću ekvivalentnih particija i uzimanjem u obzir rubnih vrijednosti značajno smanjili broj vrijednosti koje treba provjeriti.

7.5.2. Strukturno ispitivanje

Strukturno ispitivanje (engl. *white-box testing, structural testing, program-based testing*) je tehnika ispitivanja kod koje ispitivač ima dostupan programski kôd sustava koji ispituje. Ispituje se struktura koda tako što se izrađuju ispitni slučajevi koji pokrivaju određene dijelove programa. Strategija koja se pritom koristi jest ispitivanje pokrivanjem, pri čemu je cilj ispitivanja najčešće pokrivanje izvođenja svih mogućih naredbi i uvjeta programa s najmanje jednim ispitnim slučajem.

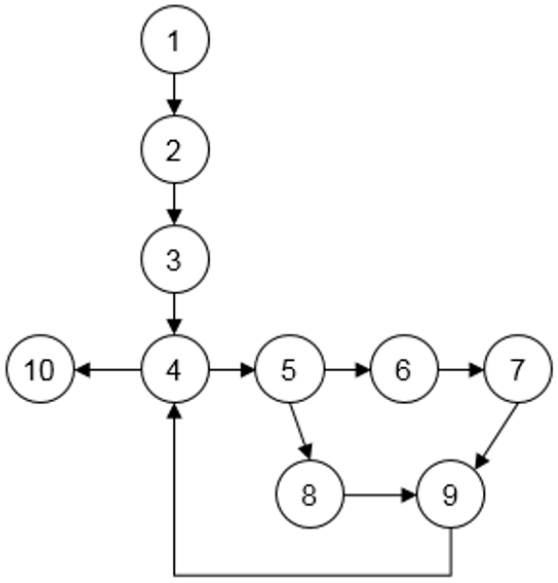
Kako bi se omogućilo učinkovito generiranje ispitnih slučajeva, program koji se ispituje apstrahira se **grafom tijeka programa**. To je graf koji pojednostavljeno grafički predstavlja tijek izvođenja programa. Graf tijeka programa sastoji se od **čvorova**, koji mogu biti obične instrukcije (procesi) ili odluke o grananju te od **lukova** koji povezuju čvorove. Čvorovi se uobičajeno označavaju s krugom unutar kojeg se nalazi redni broj instrukcije u ispitivanom programu. Radi daljnjeg

pojednostavljenja, za više instrukcija moguće je nacrtati samo jedan čvor u kojem se navode redni brojevi tih instrukcija, ako one slijede jedna iza druge i ako ne dovode do grananja programa.

Primjer grafa tijeka programa prikazan je na slici slici 7.10 desno za program koji se nalazi slijeva. Može se uočiti da su u programski kôd funkcije na slici 7.10 dodane brojke koje ukazuju na redni broj instrukcije tijekom izvođenja programa, kako bi se jednoznačno povezao program sa svojim grafom tijeka. Graf ovaj put iz edukativnih razloga nije skraćen u smislu da je više instrukcija zaredom prikazano jednim krugom.

```
public void TDM(A[], x , y , z , B[])
{
    int i = x ; (1)
    int j = y ; (2)

    for (int k = x; (3) k < z; (4) k++ (9)) {
     if (i < y && (j >= z  || A[i] <= A[j])) (5){
            B[k] = 7*A[i]; (6)
            i = i++;  (7)
     } else {
            B[k] = 4+A[j]/2; (8)
      }
    }
    return; (10)
}
```



Slika 7.10 Primjer grafa tijeka programa (desno) za određenu fukciju koja se ispituje (lijevo).

Ispitivanje temeljnih putova

Kod strukturnog ispitivanja provodi se analiza putova kroz program. Problem kod analize putova predstavlja postojanje programskih petlji, što dovodi do velikog broja mogućih putova. Iako su ti putovi zavisni jedni o drugima jer dijele iste instrukcije petlje, oni nisu posve isti, jer vrijednosti varijabli nisu iste na svakom putu. Primjerice, broj putova kroz `for` petlju programa na slici 7.10 ovisi o vrijednostima varijabli `A[], x, y,` i `z` te može ići od 0 do gotovo proizvoljno velikog broja, što značajno otežava razvoj ispitnih slučajeva. Iz tog razloga, strukturno ispitivanja se fokusira na prolazak barem jednom svim **linearno neovisnim putovima** kroz program.

Definicija 7.7. Putevi kroz program su linearno neovisni (engl. *linearly independent paths*) ako se međusobno razlikuju u barem jednoj čvoru (ili luku) grafa tijeka program, uz dodatni uvjet da jedan put nije u potpunosti sadržan u bilo kojem drugom putu.

Temeljni skup predstavlja nam osnovu za provođenje strukturnog ispitivanja. Kaže se da u tom slučaju provodimo **ispitivanje temeljnih putova** (engl. *basis path testing*) kroz program.

Definicija 7.8. Skup svih linearno neovisnih putova kroz program naziva se temeljni skup (engl. *basis set*). Taj skup minimalno jednom pokriva izvođenje svih naredbi i grananja.

Ako ispitni slučajevi prođu svim putovima u temeljnom skupu barem jednom, onda imamo garanciju da smo prošli sve instrukcije koda i da smo izvršili obje strane svih odluka o grananju. Temeljnih skupova može biti više, budući da linearno nezavisni putovi nisu jednoznačni za neki graf tijeka programa, međutim dovoljno je ispitati samo jedan takav skup.

Za izračun broja linearno nezavisnih putova koristi se teorija grafova.

Definicija 7.9. Ciklomatska složenost (engl. *cyclomatic complexity*) nekog grafa, u oznaci CV(G) je broj linearno neovisnih putova u temeljnom skupu tj. njegova kardinalnost.

Primjenu ciklomatske složenosti na mjerenje logike odlučivanja u programskom modulu prvi je proveo McCabe 1974. Ciklomatska složenost može se izračunati za bilo koji graf tijeka programa kao:

CV(G) = br. lukova – br. čvorova + 2*P

Ovdje je *P* broj povezanih komponenti (podgrafova) u čitavom grafu. U uobičajenom slučaju, ako ispitujemo jednu funkciju i imamo samo jedan takav podgraf u cijelom grafu, *P* iznosi 1, te je formula pojednostavljena na:

CV(G) = br. lukova – br. čvorova + 2

U primjeru sa slike 7.10, ciklomatska složenost iznosila bi CV(G) = 11 – 10 + 2 = 3. Primjerice, to bi bili linearno neovisni putovi:

- 1 -> 2 -> 3 -> 4 -> 10 ;
- 1 -> 2 -> 3 -> 4 -> 5 -> 8 -> 9 -> 4 -> 10 ;
- 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 9 -> 4 -> 10 ;

Potrebno je primijetiti da nijedan put nije u potpunosti sadržan u nekom drugom putu.

Osim ispitivanja temeljnih putova, strukturno ispitivanje zasebno razmatra i sljedeća ispitivanja:

- ispitivanje uvjeta (engl. *condition testing*)
- ispitivanje petlji (engl. *loop testing*)
- ispitivanje protoka podataka (engl. *dataflow testing*).

Ispitivanje uvjeta

Ispitivanje uvjeta poklanja posebnu pažnju tome da se razviju ispitni slučajevi koji bi pokrili prolaz svih granama svakog Booleovog uvjetnog (`if`) grananja. Općenito, složeni Booleov logički izraz koji se sastoji od n binarnih logičkih uvjeta koji se ispituju imat će 2^n potrebnih ispitnih slučajeva. Primjerice, izraz `if((a=b)&(c>d))` zahtijevat će dva osnovna ispitna slučaja: onaj za koji složeni izraz vrijedi i onaj za koji on ne vrijedi. Međutim, ako želimo pokriti sve mogućnosti odnosa između jednostavnih logičkih izraza, tada imamo složeniju hijerarhiju `if-else` grananja:

```
if ((a=b) & (c>d)) {...}

else if ((a!=b) & (c>d)) {...}

else if ((a=b) & (c<=d)) {...}

else {...}  // ((a!=b) & (c<=d))
```

Time nam je potrebno četiri ispitna slučaja za pokrivanje složenog izraza koji se sastoji od dva jednostavna izraza.

Ispitivanje petlji

Ispitivanje petlji fokusira se na to da se na ispravan način, s adekvatnim brojem ispitnih slučajeva, ispita svaka petlja. Adekvatni broj je uvijek onaj koji je bitno manji od broja svih mogućih putova kroz petlju, ali je obično veći od onog koji bi pokrio samo linearno neovisne putove kroz petlju. Pristupi se razlikuju ovisno o tome kakvu

petlju ispitujemo. Tako se za jednostavne petlje (koje nemaju ugniježđene druge petlje) odredi najveći mogući broj prolazaka kroz petlju n (npr. za primjer sa slike 7.10, $n = z - 1 - x$) i obično se razviju ispitni slučajevi za:

- preskakanje petlje
- jedan prolaz kroz petlju
- dva prolaza kroz petlju
- m prolaza kroz petlju ($m < n$)
- $n - 1$ prolaz kroz petlju
- n prolaza kroz petlju
- $n + 1$ prolaz kroz petlju.

Za ugniježđene petlje, unutarnja petlja se ispituje kao jednostavna petlja (vanjske petlje se postavljaju na početnu vrijednost), a zatim se vanjske petlje mijenjaju na sličan način kao što je navedeno za jednostavnu petlju te se postupak ponavlja. Adekvatno ispitivanje ugniježđenih petlji je dosta zahtjevno.

Ispitivanje protoka podataka

Ispitivanje protoka podataka jest ispitivanje upravljačkog toka programa koje uzima u obzir način korištenja varijabli i služi za uočavanje kvarova provjerom uzorka njihove uporabe. Za svaku cjelinu koja se ispituje određuje se ispravan način uporabe varijabli. Varijable se mogu u programu koristiti na neke od ovdje navedenih načina:

- definiranje (D) – deklaracija varijable (objekta), konstruktor, pridjeljivanje vrijednosti
- uporaba (U) – bez mijenjanja vrijednosti
- brisanje (K) – varijabla postaje nedefinirana, oslobađanje memorije
- nebitno (X – prijašnje akcije i X~ nebitno nakon).

Analizira se slijed akcija nad varijablama i određuju se ispravni slijedovi za svaku varijablu, npr. da je varijabla najprije definirana (D) a potom korištena (U), dakle: DU. Ili primjerice DUK. Moguće je otkrivati i moguće putove između pojedinih korištenja varijabli, npr. između D i U za pojedinu varijablu te tako podešavati ispitne slučajeve. Ako se detektira neki od neispravnih slijedova (primjerice najprije U, a potom D), dojavljuje se pogreška čija se lokacija potom treba provjeriti.

7.5.3. Ispitivanje sive kutije

Funkcijsko i strukturno ispitivanje predstavljaju ekstremne slučajeve. **Ispitivanje sive kutije** (engl. *gray-box testing*) pretpostavlja kompromis u kojem ispitivač ima ograničeno znanje o internom funkcioniranju neke metode ili međukomunikaciji internih dijelova sustava koji ispituje.

Ispitivač koji provodi ispitivanje sive kutije najčešće nije razvojni inženjer koji je sustav napravio, nego je dio ispitnog tima, moguće iste tvrtke, te poznaje kako načelno sustav iznutra funkcionira (čak i koje strukture podataka sustav koristi kao i algoritme koje ostvaruje), ali ne poznaje detalje koda.

Cilj ispitivanja sive kutije jest utvrđivanje prisutnosti pogrešaka u ispitivanom sustavu, a to se utvrđuje razvojem ispitnih slučajeva slično kao i kod funkcijskog ispitivanja (vidi poglavlje 7.5.2). Međutim, bolje poznavanje sustava omogućuje pametnije osmišljavanje ispitnih slučajeva, koji ne samo da prate dobru praksu funkcijskog ispitivanja već nastoje ciljati i uobičajene načine na koje bi se sustav mogao zlouporabiti. U tom smislu, ispitivanje sive kutije se često koristi kod tzv. **penetracijskih ispitivanja** (engl. *penetration test*), kod kojih se želi ustanoviti može li „informirani korisnik“ na neki pametan način iskoristiti slabost programskog sučelja i uzorkovati pad ili zlouporabu sustava, kao što je primjerice ubacivanje neželjenog SQL upita u sustav (engl. *SQL injection*) preko sučelja i na taj način dobavljanja podataka iz baze podataka.

7.6. Automatizacija ispitivanja

U nekom tradicionalnom poimanju procesa ispitivanja, osoba ili grupa zadužena za ispitivanje smišlja ispitne planove, ispitne slučajeve i ispitne scenarije u svrhu postizanja sveobuhvatnosti i potpunosti ispitivanja. Kada se pri tome počnu pisati pomoćne skripte prelazi se u domenu automatizacije ispitivanja što u konačnici vodi sve do uporabe programske potpore specijalizirane za ispitivanje. Dobitak automatizacije je prvenstveno u ubrzanju provjere zadanih ispitnih scenarija.

Od ostalih prednosti automatizacije, osim brzine, potrebno je spomenuti i poboljšanje uspješnosti otkrivanja pogrešaka, stabilnije ispitivanje kvalitete, automatizirano dokumentiranje prijava pogrešaka i izvješćivanje te smanjenje ljudskog rada. Nedostatci automatizacije su cijena uvođenja automatizacije i vrijeme potrebno da se automatizacija uvede.

U povijesnom pregledu glavnih konceptualnih pristupa ispitivanju, potpuna automatizacija ispitivanja još nije ostvarena i dolazi na kraju lanca koji počinje sa

začetkom potrebe pravog ispitivanja sustava 70-ih godina 20. stoljeća, negdje tijekom druge faze (vidjeti poglavlje 7.3.1). Nakon toga, u 80-im i 90-im godinama 20. stoljeća dolazi na napretka u pogledu bilježenja i ponavljanja smislenih ispitnih procesa (engl. *capture/replay testing*) te razvoja ispitnih skripti. Daljnja poboljšanja se početkom 21. stoljeća sastoje u sveobuhvatnijem razvoju ispitnih metodologija u pogledu dokumentiranja ispitnih podataka i uputa te procesa stvaranja ispitnih slučajeva gdje se kao aktualan pristup nameće **ispitivanja zasnovano na modelima** (engl. *model-based testing*). U tom pristupu, posebni modeli se koriste za generiranje samih ispita. Potpuna automatizacija ispitivanja kao (ne)dostižan cilj se očekuje tek u budućim fazama razvoja ispitnih metodologija.

Moguće je razlikovati tri općenita pristupa kod automatiziranog ispitivanja:

1. **ispitivanje na razini izvornog koda** (engl. *code-driven testing*) – ponašanje pojedinih dijelova izvornog koda se provjerava uporabom posebnih radnih okvira. Najpoznatiji su radni okviri tzv. **xUnit** za provođenje jediničnog ispitivanja po dijelovima ispitivanog koda, a za koje već postoje inačice za dobar broj programskih jezika. Automatizacija ispitivanja na razini izvornog koda je temeljna ideja TDD-a te agilnog pristupa razvoja programske potpore. U ovakvom pristupu, jedinični ispitni slučajevi se pišu prije same implementacije, odnosno pisanja izvornog koda koji ostvaruje funkcionalnost prvotno predviđenu za ispitivanje. Podatci koji se u konačnici nalaze u ispitnim slučajevima mogu biti ukodirani u sam kôd programa ili se mogu čitati iz datoteke ili baze podataka. Odvajanje podataka u zasebnu datoteku ili bazu podataka naziva se podatkovno vođena metodologija ispitivanja (engl. *data-driven methodology*).
2. **ispitivanje na razini korisničkog sučelja** (engl. *GUI testing*) – ovdje se radi o metodologiji snimanja i reprodukcije (engl. *record-and-playback methodology* ili *capture-and-replay methodology*) testova, gdje je ispitivaču omogućena interaktivna pohrana vlastitih akcija te njihovo uzastopno ponavljanje uz provjeru dobivenih s očekivanim izlazima. Ovaj pristup ne zahtijeva ozbiljniji razvoj programske potpore za proces ispitivanja, a jedina čvrsta pretpostavka je postojanje grafičkog sučelja. Isti koncept se primjenjuje i kod ispitivanja web stranica jer je "sučelje" tu implicitno uključeno, a zahtjevi razvoja dodatne programske potpore ispitivanju su također minimalni. Bitna konceptualna razlika je u tehnikama promatranja i snimanja akcija korisnika jer se kod web stranica radi o čitanju HTML koda, a kod GUI aplikacija o promatranju događaja unutar prozora. Drugi pristup ispitivanju aplikacija s grafičkim sučeljem koji ne zahtijeva razvoj prateće programske potpore ispitivanju (na strani ispitivača) temelji se na stvaranju modela sustava koji se

podvrgava ispitivanju. Bitna pretpostavka modela je da promjena različitih ispitnih parametara i uvjeta koji su ugrađeni u model omogućava ispitivaču jednostavnije stvaranje ispitnih slučajeva. Na taj način se u pogledu stvaranja i održavanja ispitnih primjera ostvaruje snaga i fleksibilnost skriptnog pristupa ispitivanju bez stvarnog pisanja skripti na strani ispitivača. S druge strane, većina napora je prenešena na samo stvaranje modela sustava koji se mora unaprijeđivati usporedno s daljnjim razvojem sustava.

3. **ispitivanje na razini API programskog sučelja** (engl. *API driven testing*) – provođenje i održavanje ispitivanja na razini grafičkog sučelja u slučaju češćih izmjena unutar grafičkog sučelja povećava udio "perifernih" aktivnosti koje nisu bitno vezane uz temeljne funkcionalnosti koje se ispituju. Ispitivanje na razini programskog sučelja nije toliko ovisno o promjenama grafičkog sučelja, već se veže uz tzv. programska sučelja aplikacije (engl. *API - Application Programming Interface*) koje sama ispitivana aplikacija pruža prema van u obliku metoda, protokola i/ili alata. Sučelja mogu biti manje ili više standardizirana u vidu za njih uporabive programske tehnologije te ih je moguće ispitati korištenjem posebne programske potpore ili u komandno-linijskom načinu rada. Podrazumijeva se postojanje odgovarajuće dokumentacije za uporabu API sučelja pa se na tom temelju mogu graditi skripte i radni okviri za automatizaciju ispitivanja.

Jedan cjeloviti ciklus automatiziranog ispitivanja bi u principu trebao uključivati sljedeće korake:

1. određivanje dijelova programske potpore koji će biti podvrgnuti automatiziranom ispitivanju
2. izbor odgovarajućeg alata za automatizaciju ispitivanja
3. pisanje ispitnih skripti
4. razvoj ispitnih skupova ili slučajeva
5. izvođenje skripti
6. izrada izvješća s rezultatima
7. pronalaženje pogrešaka i nedostataka.

7.6.1. Računalni alati za potporu automatizaciji ispitivanja

Da bi se postigla optimalnost između postavljenih ciljeva u razvoju nekog programskog proizvoda, ključna su pitanja kod pristupa automatizaciji ispitivanja:

što automatizirati, kada automatizirati i je li automatizacija uopće potrebna. Više je kriterija izbora alata za automatizirano ispitivanje:

- lakoća integracije i kompatibilnost
- performanse
- vrste podržanih testova
- održivost
- pristupačnost (financijska).

Više je skupina alata i radnih okvira programske potpore za automatizaciju ispitivanja. Najčešće korišteni su radni okviri xUnit. To je zapravo kolektivni naziv za skupinu radnih okvira čija struktura i funkcionalnost izvorno potječe od radnog okvira SUnit napravljenog za programski jezik Smalltalk. S vremenom se to proširilo i na druge programske jezike pa tako postoji JUnit za programski jezik Java, QUnit za JavaScript, PHPUnit za programski jezik PHP, NUnit za Microsoft .NET kao i mnoštvo drugih koji ne slijede nužno tu konvenciju imenovanja. U radnim okvirima te vrste, ispitni slučajevi se pišu na razini jezika izvornog koda, izvršavaju se na automatizirani način uz potvrđivanje uspješnosti ispitnih primjera (za primjer ispitivanja jedinica putem JUnita vidjeti poglavlje 7.2.1).

Posebnu grupu alata čine alati snimanja i reprodukcije koji bilježe akcije ispitivača, spremaju ih u ispitne skripte te ih potom automatizirano ponavljaju. Pritom se ovakvi alati u pravilu odnose na ispitivanje grafičkih sučelja, a prednost im je što zahtijevaju vrlo malo razvojnog posla za ispitivača. Primjeri takvih alata su Selenium, SmartBearTestComplete, IBM Rational Functional Tester i dr.

Dok ove prve dvije skupine alata kreću od fragmetarnog ispitivanja pojedinih funkcionalnosti, alati za upravljanje ispitivanjem na razini projekta (engl. *test project management*) kreću od puno šire slike funkcionalnosti ispitivanja. Ovi alati su namijenjeni za okoline raspodijeljenog razvoja programske potpore (engl. *distributed software development*) u kojima sudjeluje više razvojnih timova na različitim lokacijama ili za povezivanje s vanjskim ispitnim timovima. Smišljeni su s namjerom davanja konsolidacije i strukturiranja razvojnog procesa unutar jedinstvenog okvira alata, a neki od njih uključuju sposobnosti upravljanja sve do razine korisničkih zahtjeva. Za razliku od prve dvije grupe alata, (*xUnit* i *capture-and-replay* radnih okvira) koji češće potpadaju pod licencu otvorenog koda ili su besplatni, ovi sveobuhvatniji alati su najčešće komercijalnog tipa. Primjeri ovakvih alata su: Enterprise Tester (Catch Software), Rational Quality Manager (IBM) i TestLink (Teamtest).

Osim ovih, kao posebna skupina alata postoje i tzv. kombinacijski ispitni alati nad podacima (engl. c*ombinatorial test data tools*) koji automatiziraju stvaranje kombinacija nad ispitnim podacima, npr. Testcover, AETG, IBM Focus.

8. Računalni 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 8.1. 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 povećanje kvalitete konačnog proizvoda kroz normiranje načina prikaza i dijeljenja informacija te 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 ako se koriste CASE-alati).

8.1. 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.

8.1.1. Funkcionalna perspektiva

S obzirom na zadaću koju obavljaju CASE-alati, postoje različite vrste/tipovi alata, tablica 8.1.

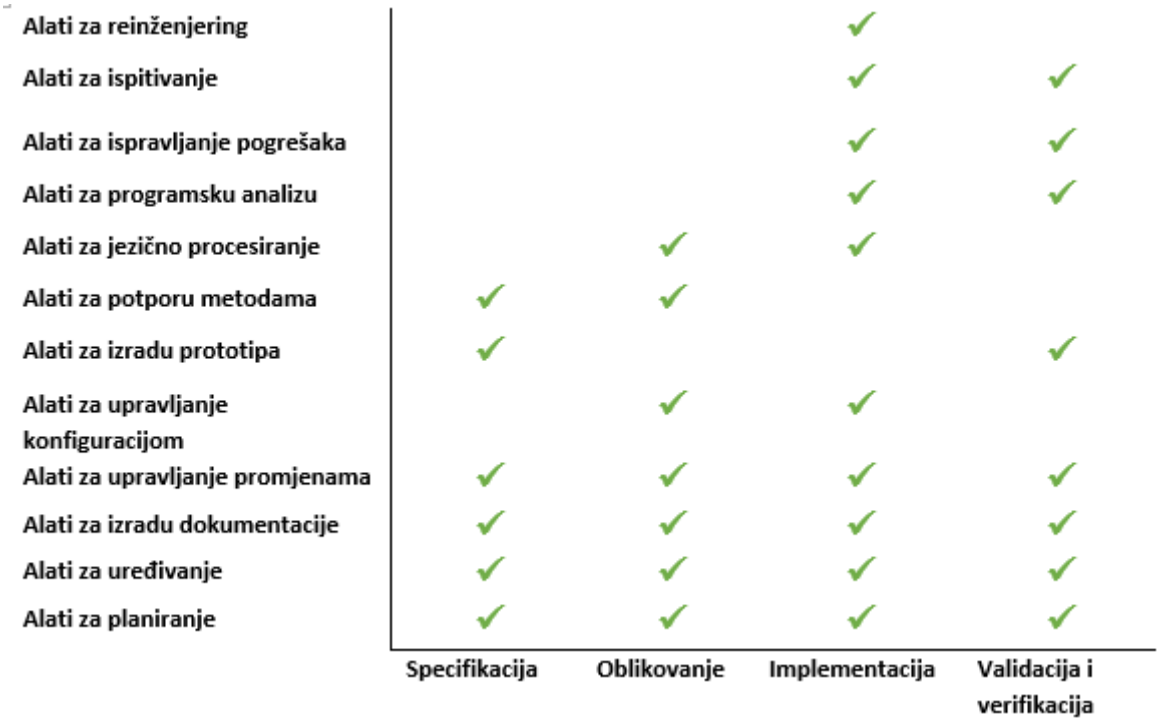
Tablica 8.1. Tipovi CASE-alata prema funkcionalnoj perspektivi.

Tip alata	**Primjer**
Alati za planiranje (engl. *planning tools*)	PERT alati, tablični kalkulatori (npr. Excel)
Alati za uređivanje (engl. *editing tools*)	Uređivači teksta, dijagrama (npr. Notepad, gedit, Word, Writer, Visio)
Alati za upravljanje promjenama (engl. *change management tools*)	Sustavi za praćenje promjena u zahtjevima i proizvodu (npr. IBM Rational DOORS, Borland Caliber)
Alati za upravljanje konfiguracijom (engl. *configuration management tools*)	Sustavi za kontrolu i upravljanje verzijama (npr. Git, Subversion)
Alati za izradu prototipa (engl. *prototyping tools*)	Jezici vrlo visoke razine apstrakcije (npr. UML)
Alati za potporu metodama (engl. *method-support tools*)	Različiti podatkovni rječnici, generatori koda

Alati za obradu jezika (engl. *language-processing tools*)	Prevoditelji, interpreteri
Alati za programsku analizu (engl. *program analysis tools*)	Različiti alati za analizu statičkih i dinamičkih performansi programa
Alati za ispitivanje (engl. *testing tools*)	Generatori ispitnih skupova
Alati za ispravljanje pogrešaka (engl. *debugging tools*)	Ugrađeni u razvojne okoline (npr. Visual Studio, Eclipse…)
Alati za izradu dokumentacije (engl. *documentation tools*)	Različiti alati za prijelom i uređivanje slika
Alati za reinženjering (engl. *re-engineering tools*)	Posebni alati za unakrsnu usporedbu dijelova sustava i restrukturiranje programa

8.1.2. Procesna perspektiva

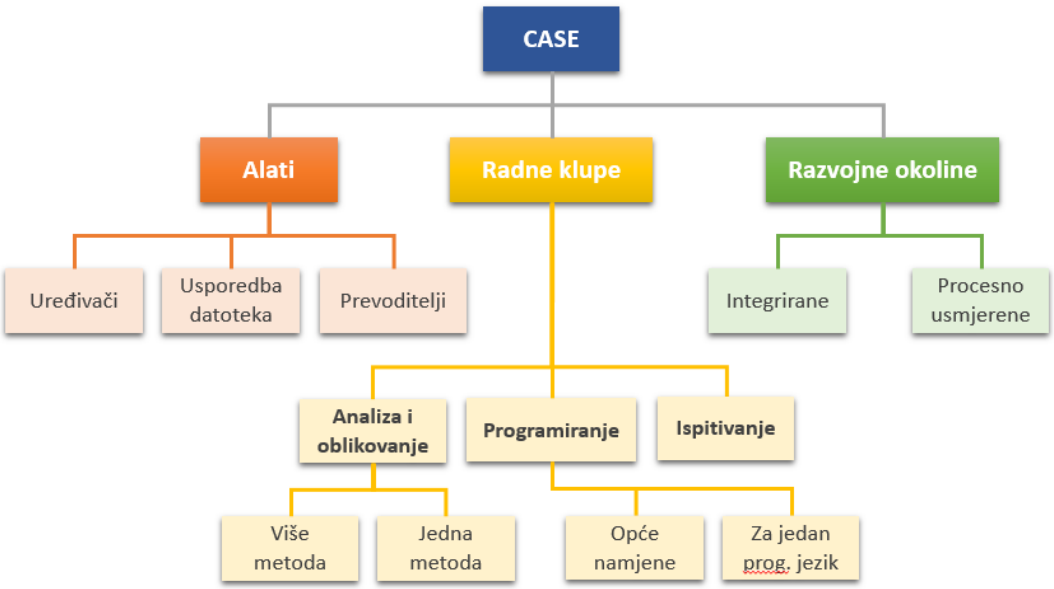
S obzirom na generičke aktivnosti procesa programskog inženjerstva koje podupire određena skupina alata može se napraviti shema prema slici 8.1.



Slika 8.1 Podjela CASE-alata prema procesnoj perspektivi.

8.1.3. Integracijska perspektiva

Ova perspektiva promatra alate s obzirom na stupanj integracije alata u cjelinu. Na slici 8.2 prikazan je dijagram integracijske klasifikacije.



Slika 8.2 Integracijska klasifikacija CASE-alata: alati, radne klupe i razvojne okoline.

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

- *Upper-CASE* alati (*front-end*), koji se koriste u ranijim fazama kao što su izlučivanje zahtjeva, analiza i modeliranje.
- *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.

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

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

- Git – upravljanje konfiguracijama programske potpore.
- Astah – 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 i IntelliJ Idea – implementacija programske potpore vezana uz Javine tehnologije (i druge).

8.2. 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 istom projektu. Stoga su kontrola inačica datoteka unutar projekta i vođenje evidencije o promjenama koje su nastale neizostavni.

Definicija 8.2. 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. Osnovni pojam je **repozitorij.**

Definicija 8.3. Repozitorij je mjesto (na udaljenom poslužitelju ili na lokalnom računalu) u kojem se nalaze sve datoteke i popratni meta-podaci.

Revizija je pojam kojim se opisuje slijed razvoja. To je jedna inačica (stanje) repozitorija, a svaki skup promjena nad repozitorijem dovodi do stvaranje nove revizije. Jedinstveni slijed razvoja, tj. revizija, u kojem nema grananja naziva se osnovna razvojna linija (engl. *trunk* ili ***master*). 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***).

Definicija 8.3. Grana (engl. *branch*) je skup datoteka obuhvaćen sustavom kontrole inačica koji se u određenom trenutku odvaja od osnovne razvojne linije i dalje se razvija zasebno i neovisno o ostalim pomoćnim linijama razvoja.

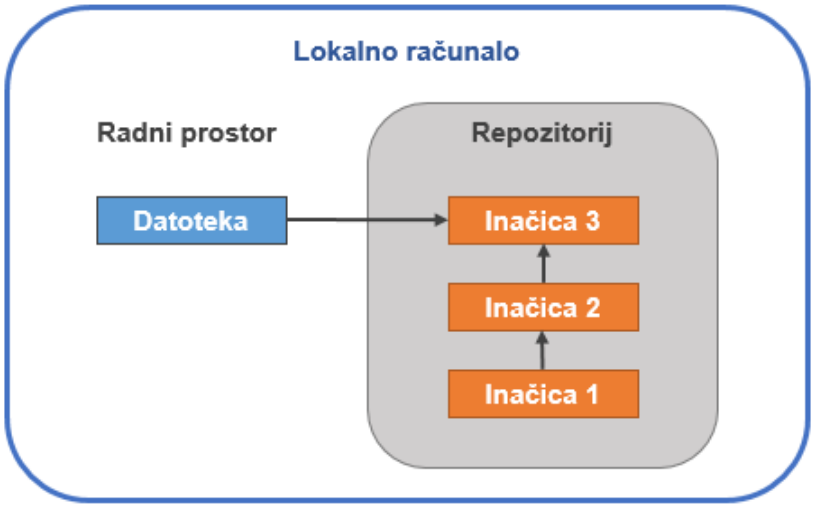
U najjednostavnijem slučaju, kada nema grananja, svaka revizija temelji se isključivo na jednoj jedinoj reviziji koja joj neposredno prethodi te sve revizije čine jednu liniju. U takvom nizu postoji jedinstvena zadnja verzija koja se često naziva **vršna ili *head* revizija** (s oznakom HEAD). Ukoliko 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 (grana) u jedinstvenu reviziju naziva se **spajanje linija** (engl. ***merge***). U praksi je proces spajanja linija težak za izvesti i predstavlja jedan od najsloženijih aspekata kontrole inačica, budući da je moguća situacija da dvoje ili više korisnika napravi promjene u istom dokumentu te sustav ne može automatski objediniti promjene novu verziju (npr. linije mogu imati iste dijelove programskog koda ali se izmijenjeni dijelovi ne moraju podudarati). U tom slučaju govorimo o **sporu (konfliktu) revizija** koji se treba razriješiti (engl. *resolve*) tako što će jedan od korisnika sam uklopiti različite promjene u jedinstvenu novu inačicu ili odbaciti sve osim jedne izabrane inačice. Na slici 8.3 dan je primjer grafa kontrole inačica s osnovnom linijom i grananjima.

Slika 8.3 Graf kontrole inačica.

8.2.1. Vrste sustava za kontrolu inačica programske potpore

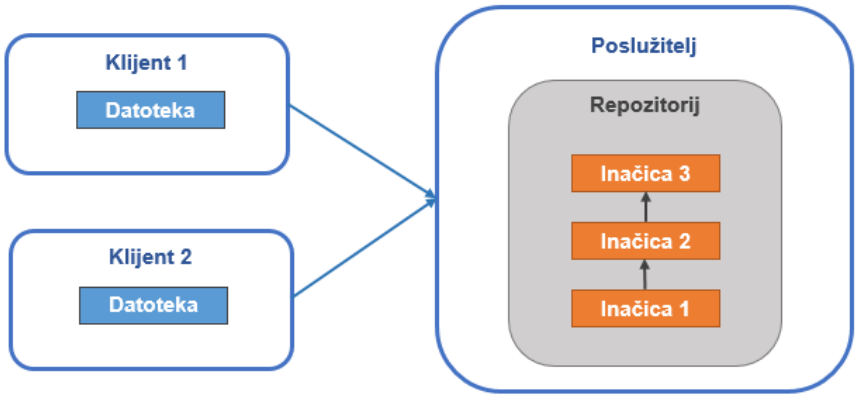
Danas postoji mnogo različitih sustava za kontrolu inačica programske potpore. Osim što se razlikuju po tipu licence – komercijalni i sustavi otvorenog koda, prije svega se razlikuju po lokalnosti pristupa te centraliziranosti, tj. postojanju središnjeg repozitorija.

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 podaci o ranijim promjenama u obliku repzitorija (baza promjena). Promjene se prate za svaku datoteku zasebno, što znači da nije moguće odjednom raditi s cijelim skupom datoteka (npr. projektom), slika 8.4. Kao dio GNU projekta razvijen je sustav RCS (engl. *Revision Control System*) koji je kasnije poslužio kao okosnica za razvoj naprednijih sustava poput CVS-a (engl. *Concurrent Versioning System*) i SVN-a (engl. *Apache Subversion*), o čemu će više riječi biti u nastavku.



Slika 8.4 Shema komponenti lokaliziranog modela.

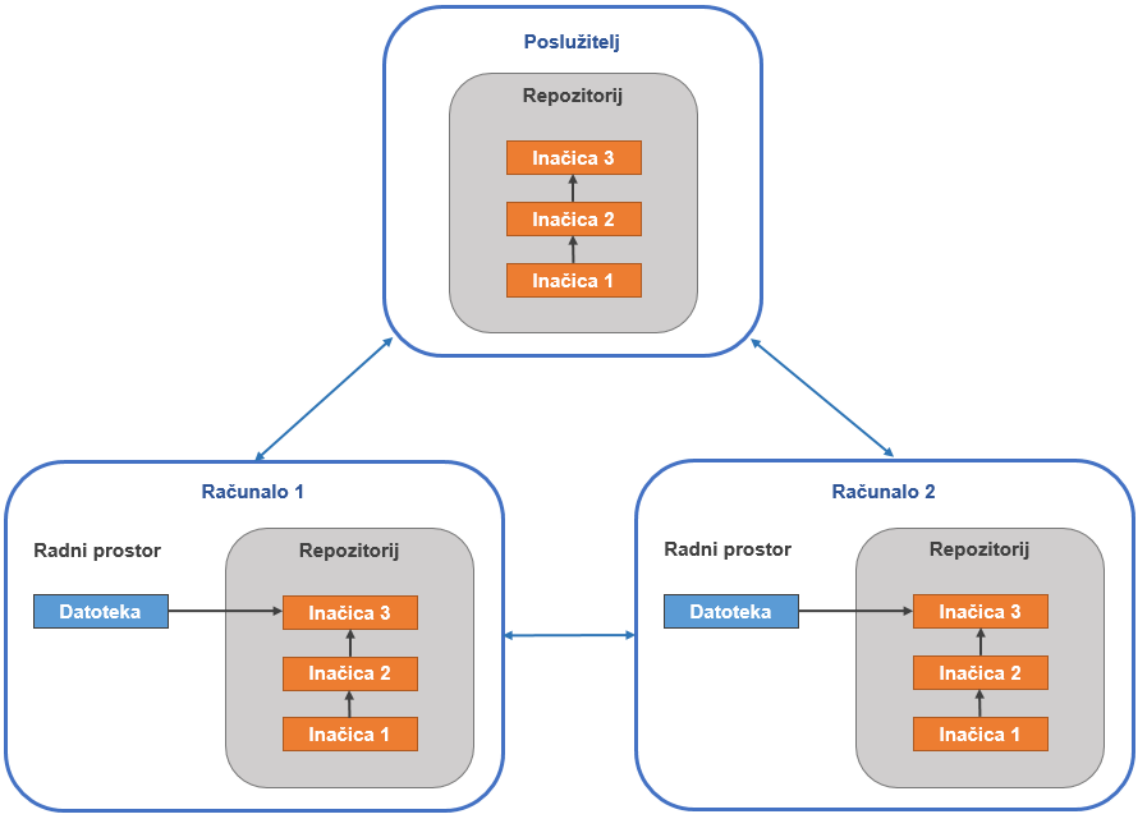
Model klijent-poslužitelj podrazumijeva postojanje središnjeg repozitorija na poslužiteljskom računalu čije radne inačice korisnici pohranjuju na svojim računalima (radni prostor) i u određenom trenutku sinkroniziraju sa središnjim repozitorijem, slika 8.5. Jedan od prvih sustava otvorenog koda koji je implementirao ovaj model bio je CVS. Na temelju njega kasnije se razvio Apache SVN koji se i danas koristi.



Slika 8.5 Shema komponenti modela klijent - poslužitelj.

Raspodijeljeni model (engl. *distributed model*) podrazumijeva da svi klijenti koji komuniciraju pri razvoju sustava imaju lokalni repozitorij koji ima kompletnu povijest revizija koda na kojem se radi. Klijent povlači (engl. *pull*) reviziju iz jednog središnjeg (udaljenog) repozitorija u lokalni repozitorij, radi na reviziji tako što stvara radnu inačicu lokalnog repozitorija i mijenja kôd te nakon što je zadovoljan s poslom izmijeni lokalni repozitorij i potom gurne (engl. *push*) reviziju nazad u središnji repozitorij. Ostali klijenti se potom na sličan način sinkroniziraju s udaljenim repozitorijem, tako da se kopija svih revizija nalazi na svim računalima klijenata, slika 8.6. U načelu, svi postojeći lokalni repozitoriji su ravnopravni, a korisnici mogu međusobno usklađivati repozitorije putem centralnog repozitorija prema načelu *peer-to-peer*. Glavni predstavnici ovakvog modela su sustavi otvorenog koda Git i Mercurial. Primjeri javno dostupnih usluga koje omogućavaju stvaranje središnjeg Git poslužitelja su GitHub[1] i GitLab[2].

[1] https://github.com/
[2] https://gitlab.com/



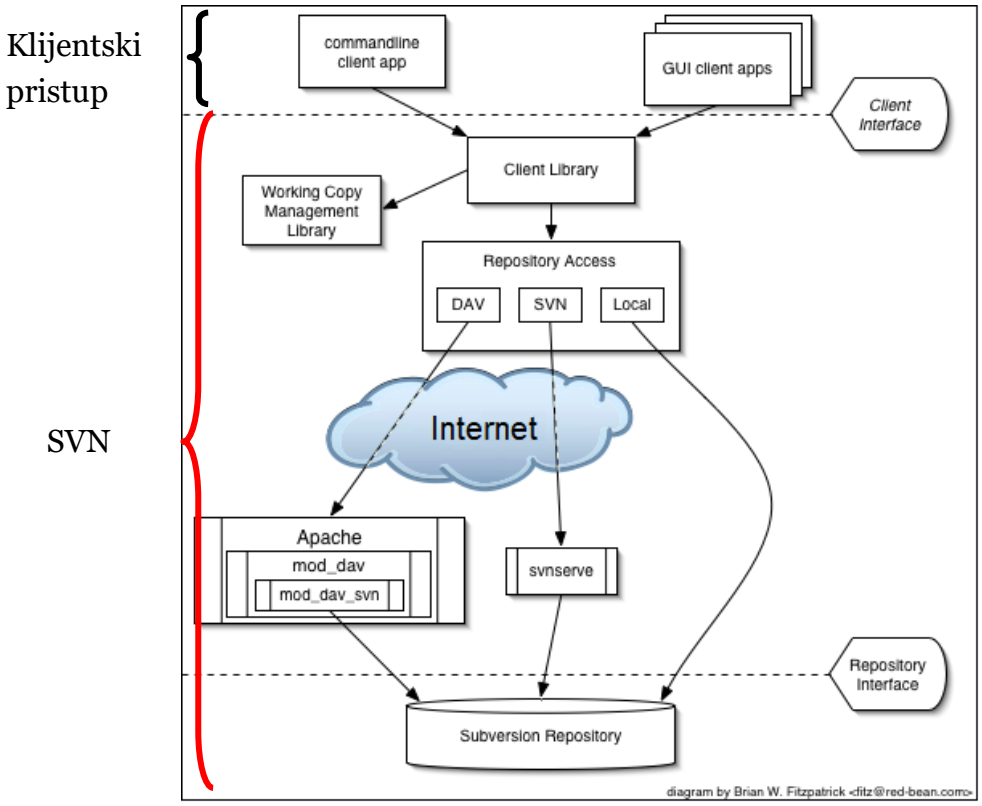
Slika 8.6 Shema komponenti raspodijeljenog modela.

U nastavku će se ukratko opisati sustavi SVN i Git, kao danas najčešći sustavi za kontrolu inačica koda.

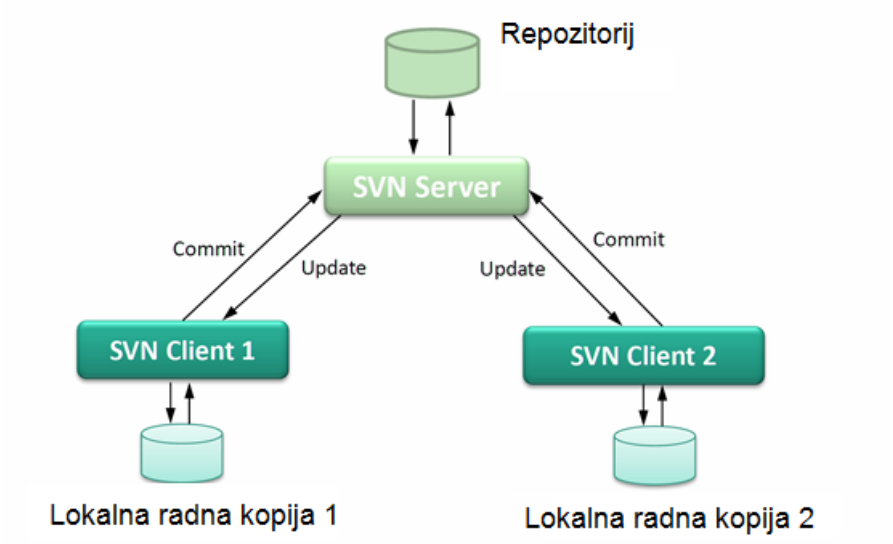
8.2.2. Apache Subversion (SVN)

SVN [20] je sustav za kontrolu inačica programske potpore distribuiran pod Apacheovom 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 8.7 prikazana je arhitektura SVN-sustava, a dijagram na slici 8.8 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`**.



Slika 8.7 Arhitektura sustava subversion (SVN).



Slika 8.8 Rad sa SVN-om.

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*). Budući da Apache SVN podržava isključivo komandno-linijski način rada,

razvijeni su posebni klijenti poput Tortoise SVN-a koji imaju integrirano grafičko korisničko sučelje i time olakšavaju korištenje SVN-a.

8.2.3. Git

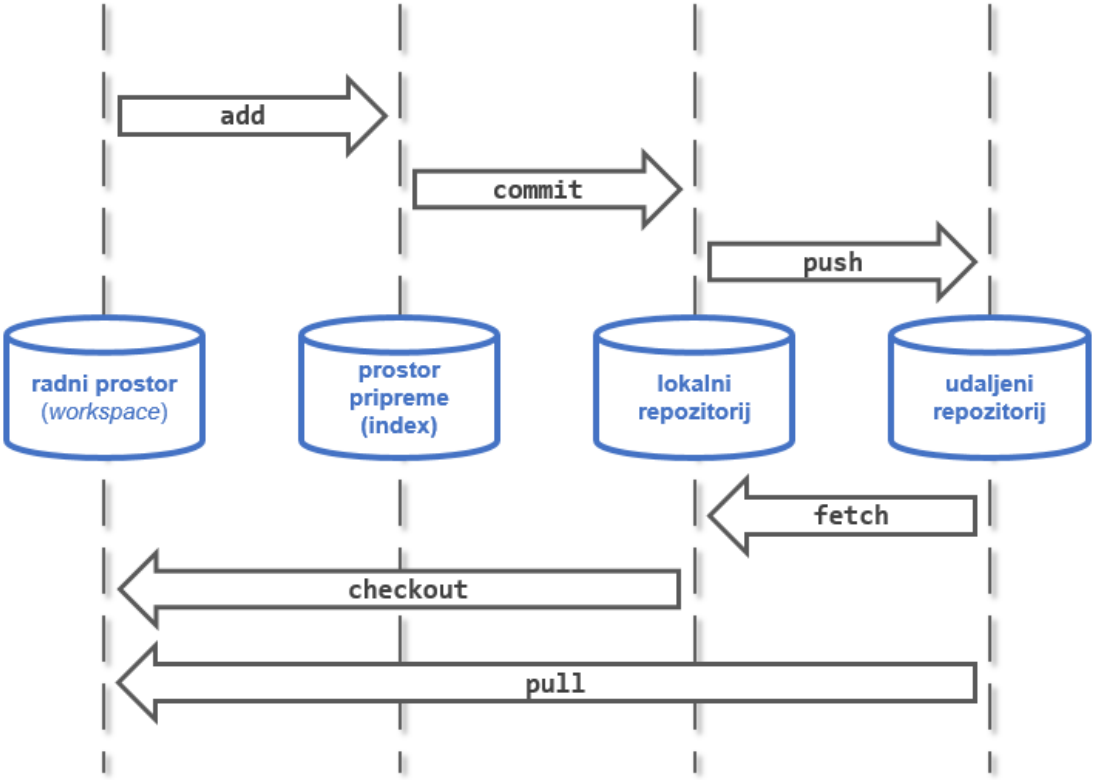
Git [21] je danas vjerojatno najraširenij sustav otvorenog koda za kontrolu inačica programske potpore. U odnosu na SVN znatno je složeniji za savladavanje i korištenje, no zato nudi dodatne funkcionalnosti, a njegova raspodijeljena arhitektura omogućava istovremeno postojanje više smjerova razvoja programske potpore koji se mogu, ali i ne moraju objediniti.

Arhitektura sustava Git podrazumijeva rad s dva repozitorija – lokalnim i udaljenim. Tijek rada s lokalnim repozitorijem je sljedeći: sve datoteke u kojima se rade promjene se nalaze u tzv. **radnom prostoru** (engl. *workspace*). Da bi se trenutačno stanje radnog prostora pohranilo u lokalni repozitorij kao nova inačica, nužno je promjene najprije prebaciti u tzv. **prostor pripreme** (engl. *staging area, index*) – lokalni kvazi-repozitorij u koji se pohranjuju nove i izmijenjene datoteke prije sinkronizacije s pravim lokalnim repozitorijem. Operacija **`add`** dodaje promjene u prostor pripreme. Promjene koje se nalaze u prostoru pripreme je moguće odbaciti - operacija **`reset`** ili pohraniti u lokalni repozitorij - operacija **`commit`**. Datoteke iz lokalnog repozitorija se u radni prostor dohvaćaju naredbom **`checkout`**.

Lokalni se repozitorij sinkronizira s udaljenim pomoću operacija:

- **`push`** – slanje promjena s lokalnog u udaljeni repozitorij
- **`pull`** (ili **`fetch`** + **`merge`**) – dohvaćanje promjena s udaljenog u lokalni repozitorij.

Prilikom prvog dohvaćanja sadržaja udaljenog direktorija kojim se stvara ekvivalentna struktura lokalnog repozitorija koristi se operacija **`clone`**. Sve navedene operacije u sustavu Git shematski su prikazane na slici 8.9.



Slika 8.9 Sustav Git - dijagram tijeka osnovnih operacija

Najčešće korištene operacije u sustavu Git su navedene u tablici 8.2, a pregled svih mogućnosti sustava Git dostupan je u [22].

Tablica 8.2. Najčešće korištene operacije u sustavu Git.

Naredba	**Opis**
`help`	Opis i parametri pojedine naredbe
`config`	konfiguracija parametara na razini repozitorija ili globalno
`init`	inicijalizacija lokalnog git repozitorija
`add`	spremanje trenutnog stanja datoteka iz radne inačice u „index”
`commit`	spremanje trenutnog stanja datoteka iz radne inačice u lokalni repozitorij
`branch`	stvaranje nove grane u lokalnom repozitoriju
`push`	Prebacivanje izmjena lokalnog repozitorija na udaljeni repozitorij

`pull`	Preuzimanje izmjena s udaljenog repozitorija u radni prostor
`fetch`	Preuzimanje izmjena s udaljenog u lokalni repozitorij
`checkout`	Osvježavanje datoteka u radnom prostoru sa sadržajem lokalnog repozitorija
`status`	Status repozitorija (promijenjene datoteke…)
`log`	Lista *commitova*
`diff`	Prikaz razlike između verzija

Literatura

[1] Sommerville, Software Engineering, 8th ed., Addison-Wesley, Harlow, England, 2007.

[2] ACM/IEEE Code of Ethics v5.2, 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] M. G. Pillai, Concept of Platform agnostic ATM application Development Framework, Virtusa, 2015. https://www.linkedin.com/pulse/concept-platform-agnostic-atm-application-development-ganesa-pillai/ , pristupljeno: 09/2019.

[10] T. Karhela, A Software Architecture for Configuration and Usage of Process Simulation Models: Software Component Technology and XML-Based Approach, doktorat na Helsinki University of Technology, 2002. https://www.researchgate.net/publication/27515831_A_Software_Architecture_for_Configuration_and_Usage_of_Process_Simulation_Models_Software_Component_Technology_and_XML-Based_Approach , pristupljeno 09/2019.

[11] Sparx Systems Pty Ltd., Deployment Diagram, https://sparxsystems.com/enterprise_architect_user_guide/14.0/guidebooks/tools_ba_deployment_diagram.html , pristupljeno 09/2019.

[12] P. Kruchen, Architectural Blueprints —The "4+1" View Model of Software Architecture, IEEE Software, 1995.

[13] Robert C. Martin (2000). "Design Principles and Design Patterns". objectmentor.com. https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf. pristupljeno 09/2019.

[14] SourceMaking, Design Patterns, 2019. https://sourcemaking.com/design_patterns , pristupljeno 09/2019.

[15] Fenton, Steve (2017). Pro TypeScript: Application-Scale JavaScript Development. p. 108. ISBN 9781484232491

[16] P. Bourque, R. E. Fairley (urednici), SWEBOK v3.0 – Guide to the Software Engineering Body of Knowledge, IEEE Computer Society, 2013.

[17] SoftwareTestingStandard, ISO/IEC/IEEE 29119 Software Testin http://www.softwaretestingstandard.org/ , pristupljeno 08/2019.

[18] University of Wisconsing Madison, Fuzz Testing of Application Reliability, 2006. http://pages.cs.wisc.edu/~bart/fuzz/ , pristupljeno 08/2019.

[19] D. Hoffman, Exhausting Your Test Options, STQE, 2003. http://www.testingeducation.org/BBST/foundations/Hoffman_Exhaust_Options.pdf , pristupljeno 08/2019.

[20] Apache, Apache Subversion, http://subversion.apache.org/, pristupljeno 09/2014.

[21] Git, http://git-scm.com/, pristupljeno 09/2019.

[22] Git, Git - Book, https://git-scm.com/book/en/v2 pristupljeno 09/2019.