# Projektovanje softvera

Projektovanje softvera je daleko osetljivija i važnija aktivnost u životnom ciklusu softvera od same izrade softvera. Objektno orijentisani pristup uvodi koncepte koji olakšavaju modelovanje ralnog sveta i omogućavaju ponovnuupotrebu koda, ali samo pod uslovom da je softverski sistem dobro isprojektovan.

Dobar projekat softvera znači ne samo upotpunosti zadovoljenje trenutnih zahteva nego i sposobnost softvera da se lako prilagodi novim budućim zahtevima korisnika. Zato su fleksibilnost i lakoća održavanja softvera jedan od ključnih principa kojima se teži u objektno orijentisanom programiranju.

Proces projektovanja softvera se može posmatrati kao proces rešavanja problema. Cilj projektovanja je da se pronađe rešenje problema. Projektovanje nije analitički proces, tako da ga treba jasno odvojiti od druge faze u životnom ciklusu softvera, koja se odnosi na analizu korisničkih zahteva.

Sam proces projektovanja zahteva da se najpre pronađe više mogućih rešenja, a da se onda ta rešenja procene i izabere najbolje. Osnovni faktori koji utiču na izbor rešenja su veličina rešenja, brzina rada, lakoća upotrebe, cena itd. Izbor rešenja se više može posmatrati kao umetnost, nego kao nauka. Za izbor dobrog rešenja treba imati pre svega dobar osećaj, dok matematički pokazatelji mogu da pomognu, ali nisu presudni. Pri tome treba voditi računa da problem ima puno aspekata sa kojih se posmatra. Ako pronađete rešenje za jedan aspekt, može se desiti da to uvede nova ograničenja u nekom drugom aspektu problema, ili da stvori potpuno novi problem.

Ata ograničenja sužavaju ukupan prostor mogućih rešenja, tako što se smanjuje broj mogućnosti između kojih programer može da bira. Ograničenja mogu biti vezana za problem koji se rešava, ali i za konkretno rešenje koje se bira. U toku faze projektovanja softvera treba proučiti uticaj tih ograničenja i najbolje je da se više puta ponovo pregleda izabrano rešenja i pronađu eventualna ograničenja.

U procesu projektovanja mogu se koristiti različiti metodi projektovanja, kao i pripremljeni obrasci za projektovanje (engl. - design patterns).

## Kvalitet softvera i softverskog projekta

Kako oceniti neki softverski projekat. Da li je sam projekat dobar ili loš? Naravno da kvalitet softvera u velikoj meri zavisi od toga da li je taj softver dobro projektovan. Sa jedne strane, dobar projekat ne mora da obavezno dovede i do dobrog proizvoda (softvera). Koliko god projekat bio dobar, ako je realizacija loša, tu nema pomoći. Sa druge strane, ako projekat nije dobar, ne postoji ništa što se tokom izrade softvera može uraditi da se od toga napravi dobar proizvod. U krajnjoj liniji se ipak uspeh projekta softvera može meriti samo uspehom finalnog proizvoda, odnosno softvera.

Kvalitet softvera se najbolje može objasniti kao kombinacija nekoliko faktora.

### *Spoljašnji i unutrašnji faktori*

Svi želimo da naš softver bude brz, pouzdan, jednostavan za upotrebu, čitljiv, modularan, struktuiran itd. Ova svojstva opisuju dve različite vrste kvaliteta.

Na jednoj strani govorimo o kvalitetima kao što su brzina i jednostavnost korišćenja. Prisustvo ovog u sistemu mogu da otkriju krajnji korisnici. U tom smislu ovo su spoljašnji faktori kvaliteta.

Drugi kvaliteti softvera, kao što su modularnost ili čitljivost predstavljaju unutrašnje faktore. Njih mogu da prepoznaju samo profesionalci koji imaju pristup do izvornog teksta sotfvera.

Na kraju presudni uticaj imaju spoljašnji faktori. Ako koristim web pretraživač iopšte me ne zanima da li je izvorni program čitljiv i modularan, ako je grafički prikaz spor kao puž. Ključa za postizanje

spoljašnjhi faktora leži u unutrašnjim. Da bi korisnici mogli da uživaju u vidljivim kvalitetima moraju da prethodno primene unutrašnje tehnologije koje će obezbediti skrivene kvalitete.

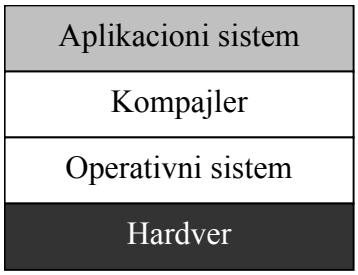
### *Spoljašnji faktori*

### *Korektnost*

Definicija: Korektnost je mogućnost softverskog proizvoda da tačno izvršava svoje zadatke koji su definsiani njihovom specifikacijom.

Korektnost je glavni kvalitet. Ako neki sistem ne radi ono što je predviđeno da radi, sve ostalo u vezi sa njim, da li je brz, da li ima dobar korisnički interfejs, nije mnogo bitno. Ovo nije lako obezbediti. Čak i prvi korak do korektnosti je težak. Zahteve sistema moramo da navedemo u preciznom obliku, što je samo po sebi veliki izazov.

Metodi koji obezbeđuju korektnost su obično uslovni. Softverski sistem, čak i onaj koji je veoma mali se dodiruje sa toliko mnogo oblasti da bi bilo nemoguće garantovati korektnost ukoliko bi se sve komponente posmatrali na jednom nivou. Zbog toga je neophodan slojevit pristup, u kome se svaki sloj oslanja na niže slojeve. Na primer:

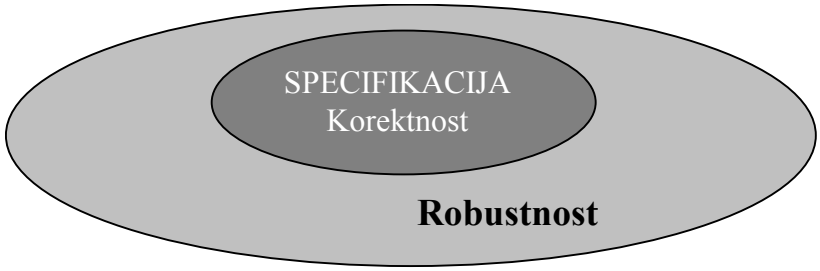



Kod uslovnog pristupa korektnosti nas jedino zanima da svaki sloj bude korektan pod pretpostavkom da su niži slojevi korektni. Praktično ne možete proveriti da li je nekii program u programskom jeziku X korektan, ako ne pretpostavite da je kompajler koji imate za taj programski jezik korektan. Ovo ne znači da treba slepo verovati kompajleru, već da treba razdvojiti dve komponente problema: korektnost kompajlera i korektnost samog programa.

### *Robustnost*

Definicija: *Robusnost je mogućnost softvera da na odgovarajući način reaguje na nenormalne situacije*.

Robusnost se posmatra uajedno sa korektnošću. Korektnost se odnosi na ponašanje sistema u slučajevima koji su opisani u specifikaciji, a robustnost karkateriše šta se dešava izvan specifikacije.




Robustnost je po prirodi stvari manje jasan pojam od korektnosti. Pošto se radi o slučajevima koji nisu pokriveni specifikacijom, nije moguće reći da sistem treba da izvrši svoj zadatak. Da su ti zadaci unapred poznati, nenormalni slučajevi bi postali deo specifikacije i bili bismo u oblasti korektnosti.

Uvek će postojati slučajevi koje specifikacija ne pokriva eksplicitno. Uloga robustnosti je da obezbedi da ukoliko se takvi slučajevi dese, ne dođe do katastrofalnih posledica. Sistem treba da proizvede poruku o grešci i da redovno završi rad.

### *Proširivost*

Definicija: *Proširivost je lakoća prilagođavanja softvera promenama specifikacije*.

Softver se generalno lako menja. Potrebno je samo da imate izvorni kod. Treba samo uzeti editor teksta i promeniti šta je potrebno.

Problem proširivosti se tiče veličine. Za male programe promena obično nije veliki problem, ali kako softver raste, sve ga je teže menjati. Veliki softverski sistem često izgleda kao kula od karata u kojoj izvlačenje bilo kog elementa može izazvati rušenje cele građevine.

Zašto je potrebna proširivost? Pogledajmo na primer, neki poslovni program. Novi zakon ili prodaja preduzeća mogu da iznenada poništе pretpostavke na kojima je sistem počivao.

Mada se tehnike koje unapređuju proširivost mogu pokazati na malim primerima, njihova opravdanost postaje jasna tek kod velikih projekata. Za poboljšanje proširivosti najvažnija su dva principa:

Jednostavnost dizajna: jednostavna arhitektura se uvek lakše prilagođava promena od složene.

Decentralizacija: što su moduli samostalniji, veća je veorvatnoća da će se mala promena odnositi samo na jedan modul, ili na mali broj modula, i da neće izazavati lanac promena u celom sistemu.

### *Višekratna upotreba*

Definicija: *Višekratna upotreba je mogućnost softverskih elemenata da sliže za konstruisanje više različitih aplikacija*.

Potreba za višekratnom upotrebom proizilazi iz činjenice da softverski sistemi često slede iste obrasce. Ovu zajedničku osobinu treba nekako iskoristiti i tako izbeći ponovno otkrivanje rešenja za probleme čije smo rešenje već imali.

Višekratna upotreba u suštini znači i da treba da se napiše manje softvera i da se time više pažnje može posvetiti drugim faktorima.

### *Kompatibilnost*

Definicija: *Kompatibilnost je lakoća kombinovanja softverskih elemenata jednih sa drugim*.

Kompatibilnost je važna, jer softverske elemente ne pravimo u vakuumu. Oni moraju da međusobno sarađuju. Često se dešava da je saradnja otežana, jer elementi imaju različite pretpostavke o ostatku sveta. Neki program može da kao svoj ulaz iskoristi pdoatke iz drugog sistema, samo ako su formati datoteka kompatibilni.

Evo jednog primera kako nekompatibilnost može da dovede do katatrofe u razvoju softvera.

*Kompanija AMR, vlasnik American Expresa, pokušala je da razvije najmoderniji industrijski sistem koji bi se primenjivao za auto i hotelske rezervacije.*

*Kompanija je prekinula razvoj novog sistema samo nekoliko nedelja pre puštanja u rad. Obustavljanje projekta vrednosg 125 miliona dolara dovelo je do smanjenja dohotka kompanije za 165 miliona dolar i uništavanja njihove reputacije lidera u tehnologiji turističkih putovanja.*

*Osnovni razlog neuspeha softvera je bio taj što su glavni delovi ogromnog projekta (opis sadrži 47 000 strana) razvijani odvojeno, različitim metodima. Kada su bili objedinjeni nisu mogli da rade jedan sa*

*drugim. Različiti moduli nisu mogli da izdvoje potrebne podatke koji dolaze sa druge strane. Otpušteno je osam viših članova projekta, uključujući i rukovodioca projekta.*

Ključ za kompatibilnost leži u homogenosti dizajna i dogovoru oko standardnih konvencija za komunikaciju. Tu spadaju standardizovni formati datoteka, standardizovane strukture podataka, standardizovni korisnički interfejsi i sl.

### *Efikasnost*

Definicija: *Efikasnost je mogućnost softvera da što manje zahteva hardverske resurse, kao što su vreme procesora, zauzeti prostor u memoriji, propusna moć u uređajima za komunikaciju*.

Performansa predstavlja sinonim za efikasnost. U vezi efikasnosti postoje dva tipična stava:

- Neki programeri imaju opsesiju u vezi performansi i ulažu mnogo napora da bi ostvarili navodne optimizacije.
- Postoji opšta tendencija da se ne ističu pitanja efikasnosti, što potvrđuju maksime tipa najpre ga napravi da radi, a zatim da radi brzo" ili "računari će sledeće godine ionako biti 50% brži".

Istina je kao i obično negde na sredini. Efikasnost ne znači mnogo ako softver nije korektan. U opštem slučaju pitanja efikasnosti se moraju uravnotežiti sa drugim ciljevima kao što su proširivost i višekratna upotreba. Ekstremna optimizacija može učiniti da softver postane toliko specijalizovan da ga više nije moguće lako menjati ili višekratno upotrebljavati. Osim toga stalno rastuća snaga hardvera nam ipak dozvoljava da ne brinemo o uštedi svakog bajta ili mikrosekunde.

Sve ovo ne umanjuje značaj efikasnosti. Ako je završna verzija sistema spora i glomazna, oni koji su govorili da "brzina nije toliko važna" biće prvi koji će se žaliti.

### *Prenosivost*

Definicija*: Prenosivost je lakoća prenošenja softvera u različita hardverska i softverska okruženja.*

Prenosivost se ne odnosi samo na varijacije fizičkog hardvera, nego opštije na kombinaciju hardvera i softvera, gde spada kombinacija procesora i operativnosg sistema.

Mnoge postojeće nekompatibilnosti nisu opravdane i za naivnog posmatrača je objašnjenje da je u pitanju zavera da bi se mučilo čovečanstvo, odnosno programeri. Bez obzira na uzrok, ova šarolikost uslovljava da prenosivost bude glavna briga onih koji softvera razvijaju, ali i onih koji ga koriste.

### *Jednostavnost upotrebe*

Definicija: *Jednostavnost upotrebe je lakoća sa kojom osobe različitog obrazovanja i kvalifikacija mogu da nauče da koriste softverske proizvode i primene ih u rešavanju problema. Ona podrazumeva i jednostavnost instalacije, održavanja i nadgledanja.*

Ova definicija insistira na različitim nivoima znanja korisnika. Taj zahtev je jedan od osnovnih izazova za projektante softvera koji se bave jednostavnošću upotrebe. Kako obezbedeiti detaljna uputstva za početnike, a pri tome ne opteretiti eksperte koji žele da odmah počnu sa radom.

Jedan od ključeva za jednostavnu upotrebu je strukturna jednostavnost. Dobar sistem, napravljen na osnovu jasne i dobro osmišljene strukture obično je jednostavniji za upotrebu nego neki zbrkan. Ovaj uslov nije dovoljan, jer ono što je projektantu jasno, korisnicima ne mora biti, ali može da pomogne.

Ova karakteristika se pre svega odnosi na korisnički interfejs. Neki projektanti softvera predlažu da pre izrade korisničkog interfejsa treba dobro upoznati korisnika. Sa druge strane, neki zastupaju stanovište, da oni koji žele da naprave dobar korisnički interfejs treba da o svojim korisnicima da pretpostavljaju

što manje. Treba imati neka osnovna očekivanja (da ljudi znaju da čitaju i pomeraju miša) ali ne mnogo više od toga.

U skladu sa prethodnim može se dati jedan savet vezan za dizajn korisničkog interfejsa:

*Nemojte misliti da poznajete korisnika. Ne znate ga.*

### *Funkcionalnost*

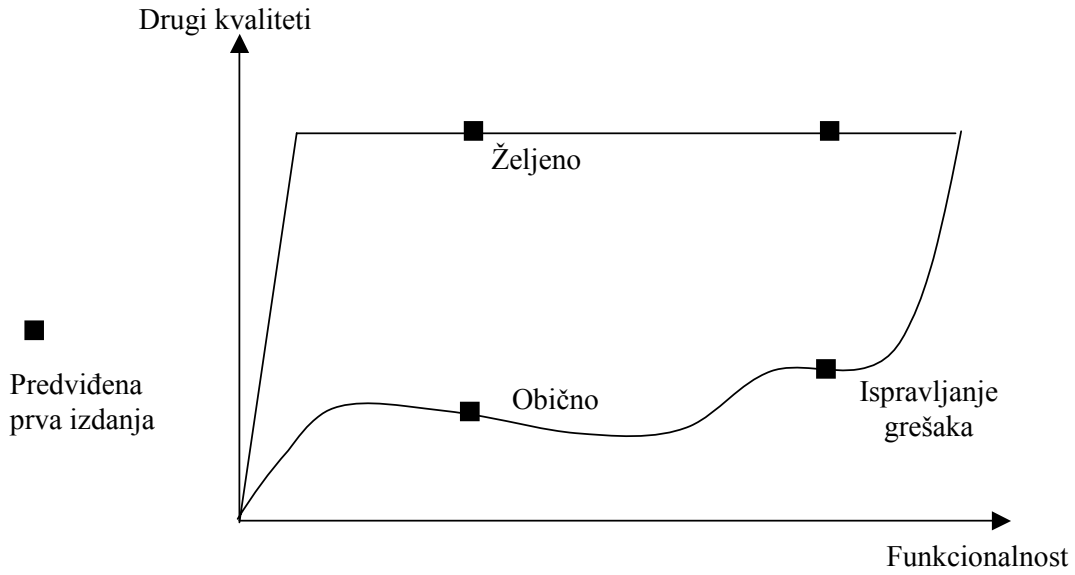
Definicija: *Funkcionalnost je dijapazon mogućnosti koje pruža sistem*.

Jedan od najtežih problema s akojima se susreće rukovodilac softvera je da zna koliko je funkcionalnosti dovoljno. Uvek postoji konstantan pritisak za dodavanje novih mogućnosti, bilo da je projekat interni kada pritisak dolazi iz istog preduzeća, ili su u pitanju komercijalni proizvodi, kada treba ugraditi sve što imaju slični proizvodi.

Ovaj problem predstavlja u stvari kombinaciju dva problema. Lakši od njih je gubitak konzistentnosti usled dodavanja novih mogućnosti, što utiče na jednostavnost korišćenja. Poznato je da se korisnici žale da nove verzije proizvoda čine njegovu uptorebu složenom, mada to treba uzeti sa rezervom, jer te nove mogućnosti proizilaze iz činjenice da su ih tražili neki drugi korisnici.

Rešenje za ovaj problem je stalan rad na konzistentnosti ukupnog proizvoda. Dobar proizvod se zasniva na malom broju dobrih ideja. Čak i ukoliko ima puno specijalnih mogućnosti, one treba da budu posledica opštih koncepata. Slika glavnog plana mora da se uvek vidi i u njemu sve mora da ima svoje mesto u njemu.

Drugi problem je da se zaokupljenost mogućnostima ne pretvori u zanemarivanje drugih kvaliteta. Projekti često upadaju u greške koje Rodžer Ozmond predstavlja grafički u obliku dva moguća puta do završetka projekta:




Donja kriva je vrlo česta. U stalnoj trci za dodavanjem novih mogućnosti, razvoj ispada iz koloseka ukupnog kvaliteta. Završna faza, u kojoj najzad sve treba sklopiti i dovesti u ispravno stanje, može da bude duga i stresna. Ako pod pritiskom korisnika ili konkurencije razvoj morate da završite ranije, u trenucima koji su označeni crnim kvadratićima, onda se može desiti da rezultat ukalja vašu čast.

Ozmond predlaže da se kvalitet stalno održava na konstantnom nivou, u celom projektu. To važi za sve aspekte osim za funkcionalnost. Ne smete da pravite kompromise za pouzdanost, proširivost i slične kvalitete. Odbijte da dodajete nove mogućnosti ukoliko niste zadovoljni postojećim.

Ovakav savet je mnogo lakše dati nego ga primeniti u praksi. Ipak svaki projekat treba da teži da prati pristup koji je predstavljen gornjom Ozbondovom krivom.

### *Blagovremenost*

Definicija: *Blagovemenost je mogućnost softvera da bude završen kada ga korisnici žele, ili pre toga*.

Blagovremenost je jedna od frustracija softverske industrije. Odličan softverski proizvod koji se pojavi suviše kasno može da promaši cilj. Ovo važi i u drugim industrijama, ali je malo onih koje se tako brzo razvijaju.

Blagovemenost, je posebno kod velikih projekata, redak fenomen.

### *Ostali kvaliteti*

Pored pomenutih kvaliteta postoje i drugi, koji se tiču korisnika sistema. Tu se pre svega misli na:

**Proverljivost** je jednostavnost pripreme postupaka primopredaje, naročito probnih podataka i postupaka za otkrivanje grešaka u toku završne faze.

**Integritet** je mogućnost softverskih sistema da zaštite svoje komponente (programe i podatke) od nedozvoljenog pristupa i modifikacija.

**Popravljivost** je mogućnost jednostavne ispravke grešaka.

**Ekonomičnost** ima veze sa blagovremenošću, predstavlja mogućnost sistema da bude završen u okviru dodeljenog budžeta.

### *Dokumentacija*

Možda bi neko očekivao da u listi faktora kvaliteta softvera naiđe i na dokumentaciju. Dokumentacija se pre može posmatrati kao posledica drugih pomenutih faktora kvaliteta, a ne kao poseban faktor. Možemo razlikovati tri vrste dokumentacije:

Potreba za spoljašnjom dokumentacijom, koja korisnicima omogućava da razumeju snagu sistema i pravilno ga koriste, je posledica jednostavnosti korišćenja.

Potreba za unutrašnjom dokumentacijom, koja programerima omogućava da razumeju strukturu sistema, je posledica faktora proširivosti.

Potreba za dokumentacijom interfejsa modula, koja programerima omogućava da razumeju module, a da pri tome ne poznavati implementaciju tih modula, je posledica višekratne upotrebe.

Umesto da dokumentaciju tretiramo odvojeno, softver treba napraviti tako da se sam dokumentuje. To se odnosi na sve tri vrste dokumentacije:

Dodavanjem ugrađene pomoći i primenom jasnih i odgovarajućih konvencija vezanih za korisnički interfejs, olakšavate zadatak autorima uputstva za korisnike.

Dobar programski jezik koji se koristi za implementaciju će u dobroj meri smanjiti potrebu za internom dokumentacijom, ako favorizuje jasnoću i strukturu.

Ataj programski jezik takođe, treba da obezbedi i razdvajanje modula od implementacije. Kasnije je moguće koristiti alate koji automatski proizvode dokumentaciju vezanu za interfejs modula, na osnovu teksta modula.

### *Kompromisi*

Prilikom pregleda spoljašnjih faktora kvaliteta softvera naišli smo na zahteve koji mogu biti međusobno sukobljeni.

Kako neko može da dobije integritet bez uvođenja zaštita raznih vrsta, što će neizostavno pogoršati jednostavnost upotrebe? Ekonomičnost se često sukobljava sa funkcionalnošću. Optimalna efikasnost zahteva potpuno prilagođavanje hardverskom i softverskom okruženju, što je u suprotnosti prenosivosti. Pritisak blagovremenosti može da nas navede da ubrzamo razvoj i dobijemo rezultat koji nije lako proširiv.

Mada je u mnogim slučajevima moguće pronaći rešenje koje će pomiriti suprotstavljene faktore, ponekad se mora praviti kompromis. Oni koji razvijaju softver često prave kompromise implicitno, be mnogo razmišljanja o alternativama. Efikasnost je obično dominantan faktor u takvim slučajevima. Pravi pristup zahteva da se jasno formulišu kriterijumu i svesno izabere alternativa.

Ma koliko bilo potrebno da s eprave kompromisi između faktora kvaliteta, jedan se jasno izdvaja. To je korektnost. Nikad ne postoji opravdanje za dovođenje u pitanje korektnosti na račun drugih faktora, na primer efikasnosti. Ako softver ne obavlja svoju funkciju sve ostalo je beskorisno.

## Metodi projektovanja softvera

Metodi projektovanja softvera sadrže osnovna upustva o izborima koji se prave tokom procesa projektovanja. Postoje različiti metodi, a mi ćemo ovde objasniti prednosti i nedostatke metoda funkcionalne dekompozicije i objektno orijentisane dekompozicije.

Metodi projektovanja softvera treba da pored ostalog obezbede mehanizam transfera znanja, da pruže neke standardne tehnike, kriterijume i ciljeve koji se koriste u timu, da omoguće zapisivanje odluka i razloga njihovog donošenja na sistematičan način, da osiguraju da su svi faktori koji su uključeni u problem uzeti u obzir (svi elementi koji postoje u projektu se moraju povezati sa nekim delom zahteva), da identifikuju važne trenutke u projektu.

### *Funkcionalna dekompozicija*

Pre pojave objektno orijentisanih tehnologija softver se razvijao uglavnom na bazi funkcionalne dekompozicije. I tada se pokušavalo da se poštuje princip modularnosti, ali su se moduli pravili oko neke akcije.

Ključni element za odgovor na pitanje "da li sisteme treba da oblikujemo oko funkcija ili podataka" je problem proširivosti, odnosno neprekidnosti. Metod projektovanja softvera zadovoljava princip neprekidnosti ako proizvodi stabilne arhitekture. Stabilna arhitektura je ona kod koje promena projekta ostaje proporcionalna veličini promene specifikacije. To znači da mala promena u specifikacija donosi relativno malu promenu projekta.

Ako se posmatra životni ciklus nekog softvera neprekidnost je jedan od glavnih problema. Većina sistema posle prve verzije doživljava veliki broj promena.

Da bi se ocenio kvalitet nekog metoda projektovanja, mora se uzeti u obzir ne samo koliko je bilo lako da se ta arhitektura početno dobije već i koliko će dobro arhitektura izdržati test promena.

Tradicionalni odgovor na pitanje modularizacije sistema je bila funkcionalna dekompozicija sistema, tipa odozgo nadole. Koliko dobro ovakav metod odgovara potrebama stvarnosti?

Princip razvoja odozgo nadole gradi sistem postepenim doterivanjem, počinjući od definicije njegove apstraktne funkcije. Ovaj proces počinje izražavanjem najopštijeg iskaza ove funkcije, na primer:

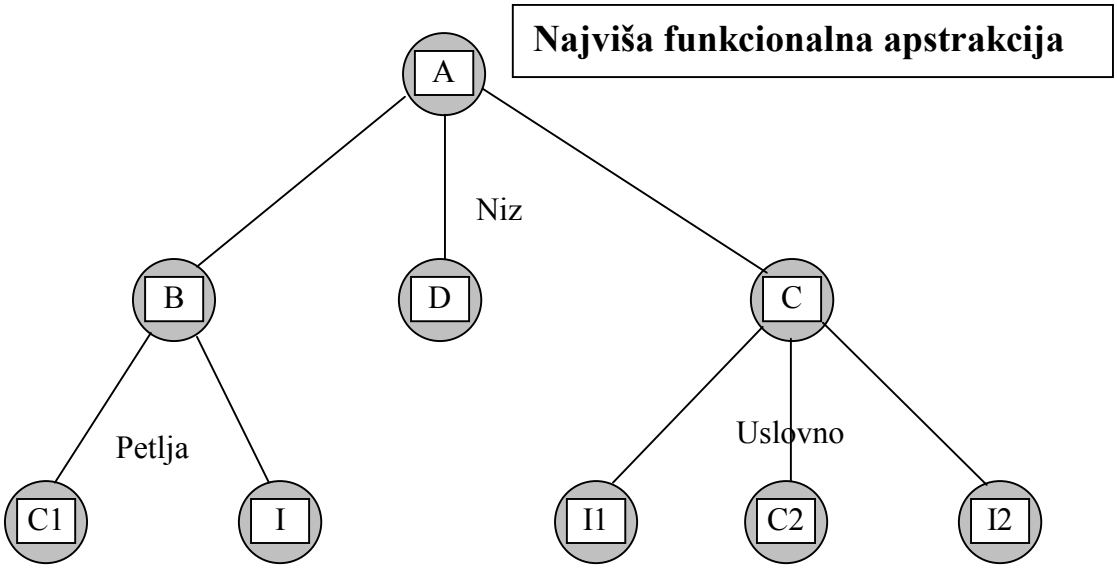
"Prevesti C program u izvršni kod"

i nastavlja se nizom koraka poboljšavanja. Svaki korak smanjuje nivo apstrakcije dobijenih elemenata, odnosno svaki korak razlaže operaciju na nekoliko prostijih operacija. Na primer, naredni korak u prethodnom primeru bi mogao biti:

- Pročitati program i proizvesti niz tokena
- Analizirati niz tokena i pretvoriti ga u stinaksno stablo
- Ubaciti u stablo sintaktičke informacije
- generisati kod na osnovu stabla

Programe u svakom koraku mora da ispita sve preostale nekompletne elemente i da razradi, sve dok ne bude na dovoljno niskom nivou apstrakcije da ih može direktno implementirati.

Proces poboljšavanja odozgo nadole se simbolički može prikazati u obliku stabla. Čvorovi predstavljaju elemente dekompozicije, a grane prikazuju odnos "B je deo poboljšanja čvora A".



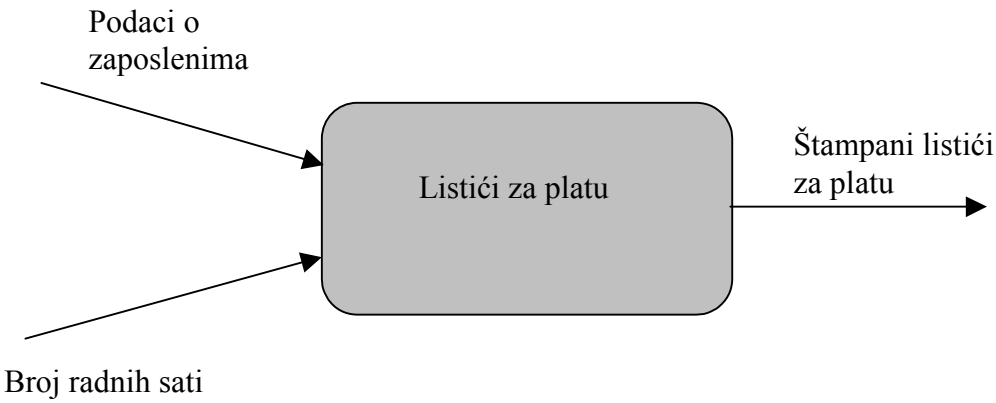

Ovakav pristup ima svoje prednosti. To je logična, dobro organizovana misaona disciplina, može se efikasno podučavati, ohrabruje uredan razvoj sistema, pomaže programeru da pronađe put kroz složenost koju sistemi pokazuju u ranim fazam projektovanja.

Pored ovih prednosti, ovakav pristup ima i neke nedostatke. To su pre svih:

- ideja da sistem može da se okarakteriše samo jednom funkcijom je vrlo sumnjiva
- pošto za osnovu modularne dekompozicije uzima svojstva koja se najviše menjaju, metod ne uspeva da objasni i prati evolucionu prirodu softverskih sistema.

### *Ne postoji samo jedna funkcija*

Prilikom evolucije sistema se često dešava da ono što je na početku izgledalo kao glavna funkcija sistema postane manje važno. Pogledajmo tipičan sistem za obračun plata. Kada je navodio svoj početni zahtev, kupac je možda naveo baš ono na šta i ime ukazuje, a to je da mu treba sistem koji na osnovu odgovarajućih podataka štampa listiće za platu. Njegov pogled na sistem bi se mogao predstaviti sledećim dijagramom:




Sistem uzima neke ulazne pdoatke (broj radnih sati i pdoatke o zaposlenima) i proizvodi neke izlazne rezultate (štampa listiće za platu). Ova specifikacija je strogo funkcionalna i sistem definiše kao izvršavanje jedne funkcije, a to je isplata zarade zaposlenima.

Pretpostavimo da je program doživeo uspeh, odnosno da radi ono za šta je predviđen. Razvoj programa se verovatno neće završiti na tome. Dobri sistemi svojim korisnicima daju prilike da im padaju na pamet ideje o drugim stvarima koje bi mogli da urade. Kao projektantu sistema, na početku vam je bilo rečeno da samo treba generisati listiće za platu i nekoliko pomoćnih rezultata. Ali sada stižu zahtevi za proširenje. Da li bi program mogao da usput izračuna i neke statističke podatke? Zar nije rečeno da ćemo ubuduće neke zaposlene isplaćivati nedeljno, a ne mesečno? Vi čuvate sve ove podatke o platama. Zar ne bi bilo lepo kada bi ljudi iz kadrovskog mogli da im pristupaju interaktivno.

Fenomen dodavanaj novih funkcija uspešnim sistemima se javlja u svim oblastima primene. Taaj proces se često odvija u malim koracima. Novi sistem je u mnogim aspektima i dalje isti kao stari. Ali originalna glavna funkcija koja na početku izgledala toliko važna, sada je samo jedna od mnogih funkcija.

Ako je prilikom projektovanja korišćena funkcionalna dekompozicija, onda struktura programa prati početnu glavnu funkciju sistema. Oni koji taj kod kasnije treba da menjaju, mogu da zažale zbog početne procene. Svako dodavanje nove funkcije, ma kako to kupcu izgledalo beznačajno, rizikuje poništavanje cele strukture.

Metodi razvoja odozgo nadole pretpostavljaju da je svaki sistem okarakterisan na najapstraktnijem nivou svojom glavnom funkcijom. Definisanje sistema jednom funkcijom je obično moguće, ali to daje veštačku sliku sistema. Mnogo je bolje sistem posmatrati kao skup određenih usluga.

Pogledajmo na primer kompajler. Ako ga posmatramo u najjednostavnijem obliku kompajler je implementacija jedne funkcije koja prevodi izvorni kod u mašinski kod. Moderni kompajleri se, međutim, ne mogu tako posmatrati. Usluge koji kompajleri nude su i otkrivanje grešaka, formatiranje programa, generisanje izveštaja itd.

Zaključak je da stvarni, realni sistemi nemaju vrh.

Ne samo da glavna funkcija nije najbolji kriterijum koji početno karakteriše sistem, već to može da, tokom evaluacije sistema, bude prvo svojstvo koje treba menjati.

Pristup razvoja odozgo nadole ima dve vrlo neprijatne posledice. Jedna je njegov naglasak na spoljašnji interfejs, a druga njegovo prerano vezivanje za vremenske relacije (redosled kojim će se aktivnosti izvršavati).

Arhitektura sistema treba da se zasniva na suštini, a ne na formi. Razvoj odozgo nadole kao osnovu strukture koristi aspekt sistema koji je najviše veštački, a to je spoljašnji interfejs.

Naglasak na spoljašnjim interfejsima je neizbežan u metodu čije je ključno pitanje "Šta će sistem uraditi za krajnjeg korisnika". Odgovor će obično isticati spoljašnje aspekte.

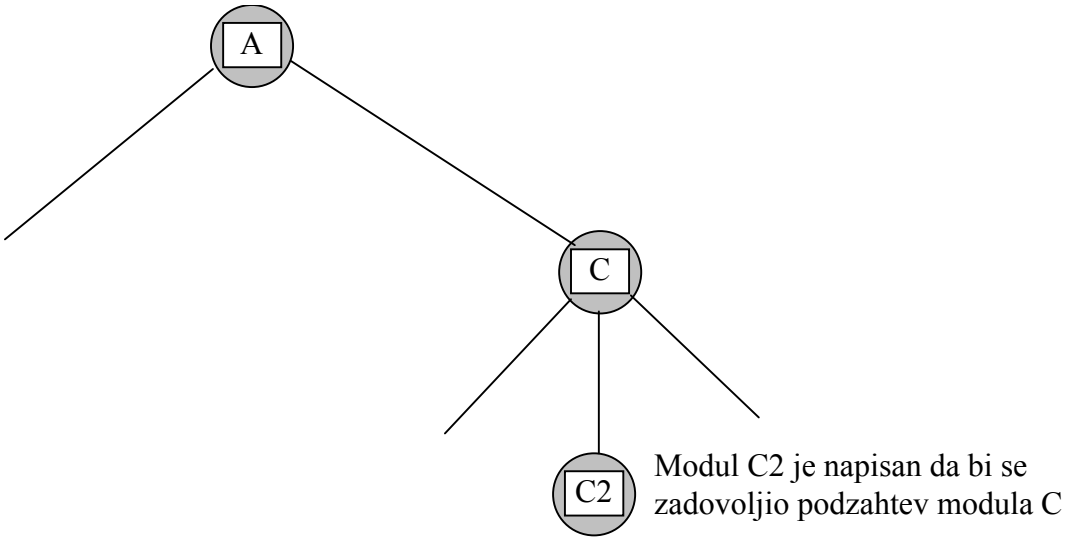
Korisnički intefejs je samo jedna komponenta sistema. To je često komponenta koja je najpodložnija promenama. Razlog je i u tome što je teško da se prvi uradi kako treba. Početne verzije mogu biti nezadovoljavajuće, jer dobro rešenje zahteva eksperimentisanje i povratne informacije od korisnika. Dobar metod projektovanja će uvek pokušati da razdvoji interfejs od ostatka sistema, jer tada korisnički interfejs postaje irelevantan za ukupni projekat sistema.

Drugi nedostatak je prerani naglasak na vremenska ograničenja. Svako poboljšavanje proširuje deo apstraktne strukture u detaljniju kontrolnu strukturu, određujući redosled u kojem razne funkcije (aktivnosti) treba da se izvrše.

Definisanje redosleda u najranijoj fazi projektovanja nije opravdano. Pitanja o redosledu imaju više mogućih odgovora i programerima treba omogućiti da se pozabave dizajnom komponenti mnogo pre nego što počnu da zalaze u strukturu redosleda.

### *Višekratna upotreba*

Kada radite odozgo nadole, to znači da softverskeelemente razvijate kao odgovor na određene podspecifikacije do kojih se dolazi tokom razvoja sistema koji ima strukturu stabla. U nekom trenutku, koji odgovaa poboljšanju određenog čvora stabla, otkrićete potrebu za specifičnom funkcijom, na primer analiza ulaznog reda, ako je reč o kompajleru i napisati njenu specifikaciju. Posle toga neko treba da je implementira.




Ova slika ilustruje ono o čemu smo govorili. Modul C2 je napisan da bi zadovoljio podzahtev modula C. Karakteristike tog novog modula su potpuno određene neposrednim kontekstom, odnosno potrebama modula C. C bi na primer mogao biti modul za analizu ulaznih podataka, a C2 modul za analizu jednog reda.

Ovakav pristup će obezbediti da projekat zadovolji početnu specifikaciju, ali ne promoviše višekratnu upotrebu. Moduli se prave kao odgovor na specifične potprobleme i obično nisu mnogo opštiji nego što je određeno njihovim kontekstom. U ovom slučaju je modul C namenjen analizi teksta određene ulazne vrste, pa je malo verovatno da će modul C2 biti primenljiv na bilo koju drugu izlaza.

U principu mi možemo kod funkcionalne dekompozicije programerima preporučiti da pišu module koji prevazilaze neposredne potrebe, ali ne postoji ništa u samom metodu što ohrabruje tu generalizaciju.

### *Kreiranje i opisivanje*

Jedan od razloga za prvobitnu privlačnost ideje dizajna odozgo nadole je u tome da je on pogodan za objašnjenje nekog projekta, nakon što je on već završen. Sa druge strane ono što je dobro za dokumentovanje postojećeg dizajna nije obavezno najbolji način za novi projekat.

Kada projektant sistema već ima jasnu ideju kompletnog rezultata, on može iskoristiti postupak odozgo nadole da na papiru opiše ono što je u njegovoj glavi. Zato ljudi mogu verovati da koriste pristup odozgo nadole i da su u tome uspešni, ali oni pri tome mešaju metod opisa sa metodom razvoja. Kada faza odozgo naniže počinje, problem je već rešen i ostaje samo da se specificiraju detalji.

### *Zaključak*

Prethodna diskusija pokazuje da metod nije pogodan za razvoj bitnih i većih sistema. On ostaje korisna paradigma za male programe, i pojedinačne algoritme. On je svakako korisna tehnika za opisivanje dobro poznatih algoritama, naročito na kursevima programiranja. Sam metod, međutim, ne može da se prilagodi velikom softveru. Ako se koristi ovakv pristup žrtvuje se dugoročna fleksibilnost, radi kratkoročne pogodnosti. Jednoj funkciji se daje prekomerna važnost u odnosu na druge, može se desiti da se interfejsu posveti preterana pažnja, na uštrb bitnijih osobina, gubi se iz vida aspekt podataka, a postoji i rizik žrtvovanja višekratne upotrebe.

## Obektno orijentisana dekompozicija

Mnogi argumenti koji govore protiv funkcionalne dekompozicije prirodno govore u korist objektno orijentisanog pristupa. Upotreba objektno orijentisanog pristupa međutim, ne znači da u potpunosti treba odbaciti funkcije. Nijedan pristup konstruisanju softvera ne može biti potpun ako ne uzima u obzir kako funkcionalne, tako i objektne delove. Stoga i kod objektno orijentisanog pristupa treba zadržati funkcije, iako one u ovom slučaju svoju ulogu podređuju objektima. Apstraktni tip podatka (klase) obezbeđuju definiciju objekta u kojoj ima mesta i za funkcije.

Objektno orijentisani pristup projektovanju softvera donosi različite prednosti. Najvažnije su one koje se odnose na proširivost, višekratnu upotrebu i kompatibilnost.

### *Proširivost*

Ako se funkciej sistema često menjaju, postavlja se pitanje da li se može naći nešto što će stabilnije okarakterisati suštinska svojstva sistema i voditi nas u izboru modula.

Taipovi objekata kojima sistem manipuliše mnogo više obećavaju. Ako pogledamo sistem za obračun plata koji smo koristili ranije, bilo šta da se desi sa njim on će i dalje manipulisati objektima koji predstavljaju zaposlene, koeficijente plata, propise preduzeća, radne sate itd. Bilo šta da se desi sa nekim programom za proračun metodom konačnih elemenata, on će i dalje manipulisati matricama, konačnim elementima i koordinatnim mrežama.

Ovaj argument važi samo ako objekte posmatramo iz perspektive dovoljno visokog nivoa. Ako objekte posmatramo samo u okviru njihove fizičke predstave, nećemo dobiti mnogo više od onog što imamo sa funkcijama, odnosno dobićemo čak i manje, pošto funkcionalna dekompozicija bar ohrabruje apstrakciju. Zato je pitanje pronalaženja odgovarajućeg apstraktnog opisa objekata od suštinske važnosti.

### *Višekratna upotreba*

Ako se kao jedinica višekratne upotrebe posmatra samo rutina (funkcija) može se izvesti zaključak da ona obično nije dovoljva. Pogledajmo primer pretraživanja tabele. Ako pođemo od naizgled prirodnog kandidata za višekratnu uptorebu rutine za pretraživanje, doći ćemo do zaključka da takvu rutinu nije lako više puta upotrebiti odvojeno od operacija, kao što su stvaranje, dodavanje i brisanje. Ove operacije se primenjuju na tabelu za pretraživanje. Otuda se prirodno nameće ideja da zadovoljavajući višekratno upotrebljiv modul, treba da bude kolekcija takvih operacija. Ako pokušamo da pronađemo konceptualnu nit koja objedinjuje sve te operacije, dolazimo do tipa objekta na koji se one primenjuju, a to je tabela.

Ovakav i slični primeri pokazuju da tipovi objekata, sa svim odgovarajućim operacijama, obezbeđuju stabilne jedinice višekratne upotrebe.

### *Kompatibilnost*

Kompatibilnost je još jedan faktor kvaliteta softvera. Može se definisati kao jednostavnost kojom se softverski proizvodi (moduli) mogu međusobno kombinovati.

Teško je kombinovati operacije, ako strukture podataka kojima one pristupaju nisu predviđene za tu svrhu. Zašto onda ne bismo kombinovali cele strukture podataka?

### *Objektno orijentisano projektovanje softvera*

Objektno orijentisano projektovanje softvera je metod za razvoj softvera koji arhitekturu softverskog sistema zasniva na modulima izvedenim iz tipova objekata kojima sistem manipuliše (umesto na funkcijama koje sistem treba da obezbedi).

Analizom ove definicije možemo doći do nečeg što može biti moto svakog programera koji koristi objektno orijentisani pristup:

***Ne pitajte se odmah šta sistem radi.***

***Pitajte se nad čime radi.***

Da biste na kraju dobili implementaciju koja funkcioniše, moraćete pre ili kasnije da saznate šta sistem radi. Otuda reč odmah u prethodnoj definicij. Bolje kasnije nego pre, tvrdi objektno orijentisani pristup. Izbor glavne funkcije, kod ovog pristupa je jedan od poslednjih koraka koje treba preduzeti u procesu projektovanja softvera.

Programeri koji koriste OO programiranje će do poslednjeg trenutka odložiti trenutak opisa i implementacije najviše funkcije sistema. Umesto toga oni će analizirati tipove objekata. Dizajn sistema će postepeno napredovati, kako se bude povećavao nivo shvatanja klasa objekata. To je proces građenja proširivih i pouzdanih rešenja za delove problema. Tom prilikom se ide odozdo naviše. Ta rešenja se zatim kombinuju u sve veće sklopove, sve do konačnog sklopa koji daje krajnje rešenje. Tu ostaje i nada da to rešenje nije jedino moguće, odnosno da se iste komponente, samo drugačije složene, i kombinovane sa drugim, novim komponentama, mogu upotrebiti kao rešenja za buduće probleme. Ovo je nusproizvod rada na rešenju prvobitnog problema.

### *Pronalaženje tipova objekata*

Iz prethodne priče se verovatno odmah nameće pitanje kako pronaći objekte (tipove objekata). Na ovo pitanje se može dati odgovor primenom sledećih razmišljanja:

- Mnogi objekti se nameću sami po sebi. Oni direktno modeliraju objekte fizičke realnosti na koju se program odnosi. Jedna od boljih strana objektno orijentisane metodologije je da ona

predstavlja veoma dobar alat za modeliranje, pri čemu se koriste softverski tipovi objekata (klase). Nije potrebna neka ogromna diskusija da biste programer, na primer, neke telekomunikacione aplikacije, ubedili da u svojoj aplikaciji treba da ima klase POZIV i LINIJA, ili da programera sistema za obradu dokumenata ubedite da treba da postoje klase DOKUMENT, PASUS i FONT.

- Izvor tipova objekata je višekratna uptoreba. Tu se misli na klase koje su drugi napravili. Iako se ovo ne ističe dovoljno u OO literaturi, ovo je tehnika koja se najviše koristi u praksi. Mora se odoleti nagonu da izmišljamo nešto, ako su drugi taj problem već rešili na zadovoljavajući način.
- Iskustvo i kopiranje imaju takođe svoju ulogu. Kako budete upoznavali uspešne primere objektno orijentisanog projektovanja i dizajnerske obrasce, bićete u stanju da na osnovu tih primera dobijete inspiraciju kako definisati objekte.

### *Opisivanje tipova objekata*

Pod pretpostavkom da nekako uspemo da dobijemo objekte, posavlja se pitanje kako te objekte opisati. Prilikom odgovora na ta dva pitanja moramo se držati dva kriterijuma:

- Potrebno je obezbediti opise koji su nezavisni od predstavljanja, kako ne bismo izgubili apstrakciju.
- Potrebno je iskoristiti i funkcije i dati im pravo mesto u strukturi sistema, čija je dekompozicija uglavnom zasnovana na tipovima objekata. Na kraju se ipak mora uzeti u obzir dualizam objekat-funkcija.

### *Opisivanje relacija i struktura softvera*

Još jedno pitanje je, koje vrste relacija između objekata treba dozvoilit. Pošto će moduli biti zasnovani na objektima, odgovor na ovo pitanje određuje i tehnike struktuiranja koje možemo koristiti za sklapanje sistema od komponenti.

U najčistijem obliku objektne tehnologije postoje samo dve relacije: klijenska i nasleđivanje. One odgovaraju različitim vrstama zavisnosti između dva tipa objekata A i B:

- B je klijent tipa A, ako svaki objekat tipa B može sadržati informacije o jednom ili više objekata tipa A.
- B je naslednik tipa A ako B označava specijalizovanu verziju tipa A.

Ako ste navikli na modeliranje preko ER (entity relationship) dijagrama, onda postojanje samo ovakve dve relacije, na prvi pogled može izgledati restriktivno. Ali taj utisak ne mora obavezno biti i tačan.:

- Klijentska relacija je dovoljno široka i obuhvata mnoge različite oblike zavisnosti. Atu spada i ono što se obično naziva agregacijom (prisustvo podobjekta tipa A u svakom objektu tipa B), referentna zavisnost i generička zavisnost.
- Relacija nasleđivanja obuhvata specijalizaciju u mnogim različitim oblicima.

### Primer

Da pogledamo kako teorija koju smo ovde izložili može da se primeni u praksi.

Problem je napraviti interaktivni sistem, uobičajen za obradu poslovnih podataka u kojima se korisnici vode korak po korak, pomoću prozora koji prekrivaju ceo ekran. Prelazi iz jednog u drugi prozor su

unapred definisani. Ovaj primer ćemo uraditi na više načina, tako da možemo da istaknemo dobra i loša svojstva svake od metodologija u razvoju softvera.

Problem je relativno jednostavan. Svaka sesija proalzi kroz određen broj stanja. U svakom stanju se prikazuje određeni prozor sa pitanjima za korisnika. Korisnik popunjava traženi odgovor, proverava se konzistentnost odgovora (pitanja se ponavljaju sve dok se unese dobar odgovor), zatim se odgovor obrađuje na neki način (na primer, ažurira se baza podataka). Deo krosinikovog odgovora omogućava izbor sledećeg koraka koji treba izvršiti, a koji sistem interpretira kao sledeće stanje u kome će se primeniti isti proces.

Tipičan sistem može biti rezrevacija letova. Stanja koja tu postoje su: identifikacija korisnika, upit o letovima (za određene destinacije i određene datume), upit o sedištima (za određeni let) i na kraju rezervacija.

Na primer, prozor za unos upita o letovima može izgledati kao na sledećoj slici. Ekran je prikazan na kraju unosa, stavke prikazane kurzivom su odgovori korisnika, dok su masnim slovima ono što sistem odgovara.

**Letovi**

Let iz: Beograd Za: Pariz

Polazak: 21 Nov

Željeni prevoznik:

Specijalni zahtevi:

Raspoloživi letovi: 1

**Let: AA 42 Polazak 12:00 Dolazak 13:45**

Izaberite sledeću akciju:

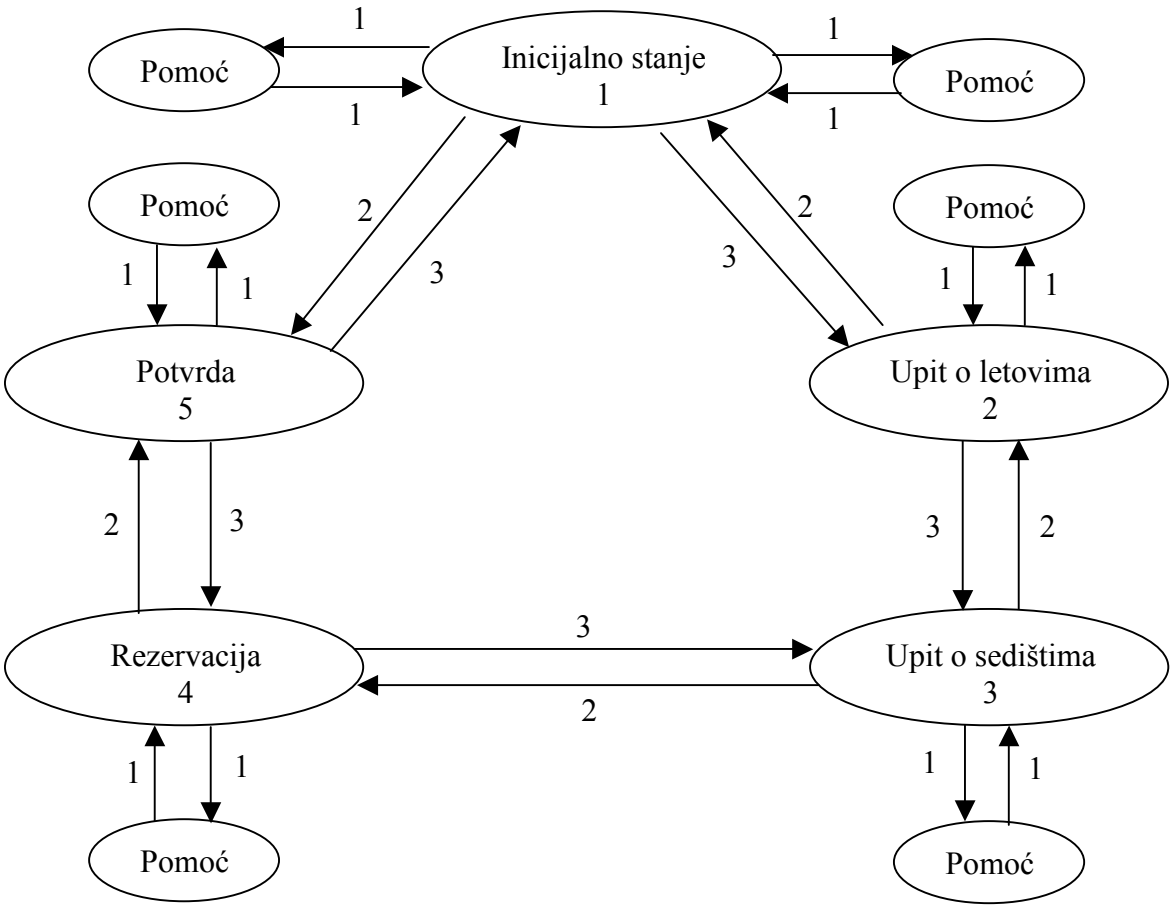
**0 - Izlaz**

**1 - Pomoć**

**2 - Dalje pretraživanje**

**3 - Rezervacija**

Globalnu strukturu možemo predstaviti dijagramom prelaza, koji prikazuje moguća stanja i prelaze između njih. Stanja su prikazana elipsama, koje su obeležene celim brojevima, koji odgovaraju mogućim odgovorima korisnika za sledeći korak na kraju stanja. Dijagram je prikazan na sledećoj slici:




Problem koji se ovde postavlja je obuhvatiti sve mogućnosti projekta i aplikacije uz što veću fleksibilnost. Na primer:

- Dijagram može biti velik. Nije neobično da postoje aplikacije sa nekoliko stotina stanja i prelaza.
- Struktura je podložna promeni. Projektanti ne mogu da predvide sva moguća stanja i prelaze. Kada korisnik započne testiranje sistema, pojaviće se zahtevi za promenama i dodacima.
- Ako je nekom potreban ovakav sistem, koji će se možda koristiti i na drugim mestima, bolje je napraviti opšti dizajn, ili još bolje skup modula koji se mogu upotrebljavati na više mesta, od aplikacije do aplikacije.

### *Prvi pokušaj*

Počećemo pravolinijskom programskom šemom. Ta verzija je napravljena od brojnih blokova. Postoji po jedan blok za svako stanje sisema. Blokove ćemo označiti sa $B_{upit}$, $B_{rezrevacija}$ itd. Tipičan blok može se predstaviti ovako:

```
Bupit:
   "Prikaži prozor Upit za letove"
   ponavljaj
          "Pročitaj odgovore korisnika i izbor C za sledeći korak"
          ako "odgovor pogrešan" onda "prikaži poruku"
   sve dok "odgovor nije pogrešan"
   "obradi odgovor"
   switch (C)
```

```
            C0: goto izlaz
            C1: goto B_pomoć
            C2: goto B_rezervacije
            ...
end
```

Ovo se može primeniti i na ostale korake.

Ovakva struktura ima svojih dobrih strana. Nije je teško napraviti i obaviće posao. Sa tačke softverskog inženjerstva ostaje još puno toga što bi se moglo poželeti.

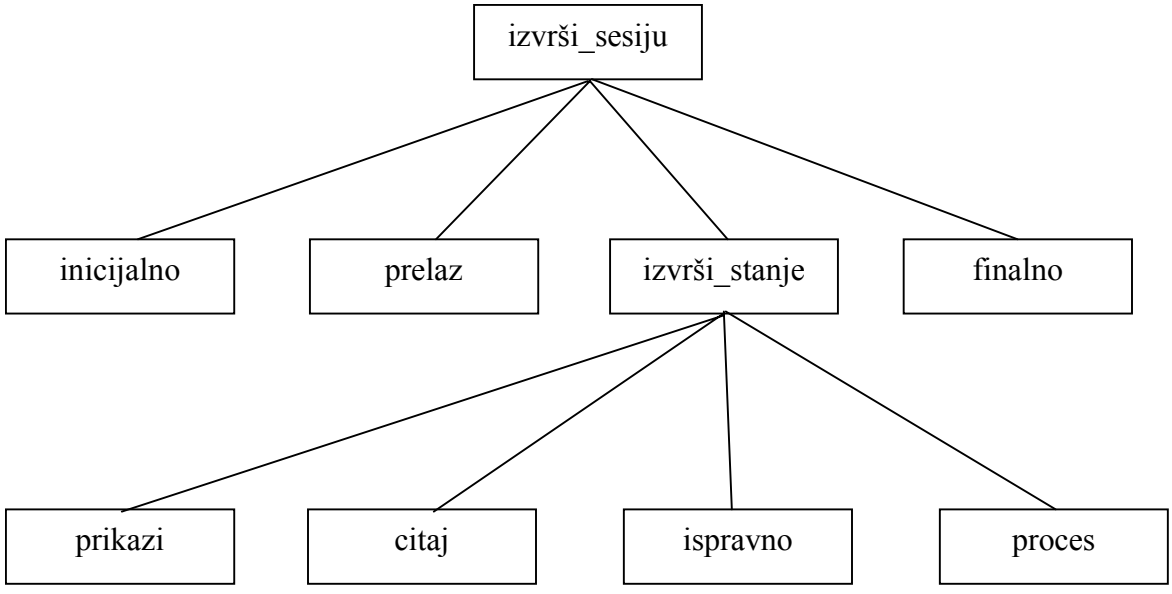
Najočiglednija kritika je prisustvo instrukcija tipa goto.

Naredbe goto nisu greška, već simptom. Uzeli smo u obzir samo površnu stranu problema, odnosno trenutni oblik dijagrama prelaza i ubacili ga u algoritam. Granajuća struktura programa je odraz strukture dijagrama prelaza. Ovo projekat softvera čini ranjivim na bilo koje jednostavne promene. Ako neko zatraži da dodamo stanje ili promenimo prelaz, moraćemo da menjamo centralnu kontrolnu strukturu sistema. Naravno da možemo da zaboravimo na višekratnu upotrebu jer kontrolna struktura mora da pokrije sve aplikacije.

## Funkcionalno rešenje odozgo nadole

Da bismo unapredili rešenje, prvo treba da se otarasimo centralne uloge algoritma obilaska dijagrama prelaza u strukturi. Šta je dijagram prelaza? Apstraktno se on može shvatiti kao funkcija prelaz(), koja ima dva argumenta, to su stanje i izbor korisnika. prelaz(s, c) predstavlja stanje u koje se dolazi kada korisnik izabere c dok napušta stanje s. U ovom trenutku ne moramo još uvek da odlučimo da li je funkcija prelaz, koju ovde pominjemo zaista funkcija, ili će se realizovati preko neke strukture podataka, kao što je na primer, niz. U ovom trenutku se odlaže izbor konačnog rešenja, a funkcija se posmatra u matematičkom, apstraktnom smislu.

Ako pratimo tradicionalne recepte dekompozicije odozgo nadole, onda biramo vrh, odnosno glavni program našeg sistema. To bi mogla biti rutina *izvrši_sesiju*, koja opisuje način na koji treba da se izvrši celokupna sesija sa korisnikom.




Odmah ispod na nivou 2, se nalaze operacije koje se odnose na stanja. To su definicije inicijalnih i završnih stanja, struktura prelaza i rutina *izvrši_stanje*, koja propisuje akcije koje se izvršavaju u svakom stanju.

Kao što se vidi ovakvo rešenje odslikava realni svet, odnosno struktura softvera savršeno odslikava strukturu aplikacije, koja uključuje stanja, a ona sadrže elementarne operacije. Ono što OO programiranje odvaja od drugih pristupa nije realnost, jer ona postoji i kod drugih, pa i ovog pristupa, već način na koji se svet modelira.

U svakoj fazi se nalazimo u određenom stanju. Na početku je to stanje inicijalno. Proces se zaustavlja kada dođemo u neko finalno stanje. Za stanje koje nije konačno se izvršava rutina izvrši_stanje, koja uzima trenutno stanje i vraća korisnikov izbor prelaza. Taj rezultat se šalje u rituni prelaz, koja određuje sledeće stanje.

Rutina izvrši_stanje opisuje akcije koje se izvode u svakom stanju. Ona dalje poziva rutine prikazi, koja prikazuje odgovarajući prozor u zavisnosti od stanja, citaj, koja učitava ono što je korisnik odgovorio, rutinu ispravno, koja vraća tačno samo je to što je korisnik odgovorio prihvatljiv odgovor za pitanje u stanju s. Ako je to u redu, onda rutina proces obrađuje odgovor (na primer ažurira bazu podataka).

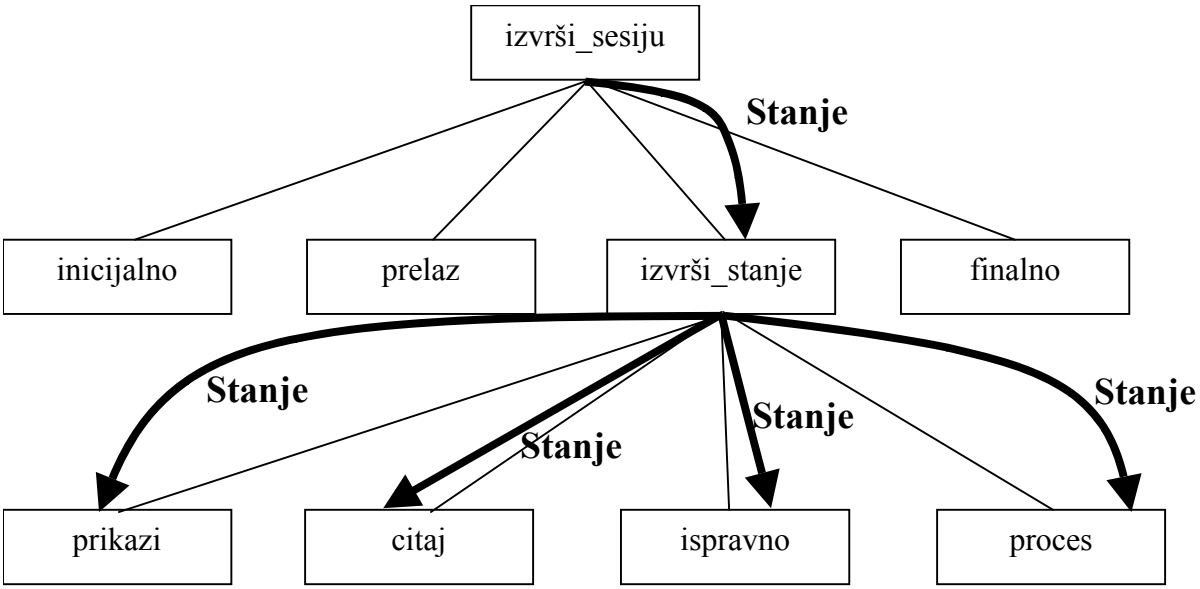
### *Kritika rešenja*

Da li smo dobili zadovoljavajuće rešenje? Ne baš. Rešenje je bolje od prve verzije, ali i dalje nisu postignuti ciljevi vezani za proširivost i višekratnu upotrebu. Deklaracija rutina koje smo pomenuli bi mogla da izgleda ovako:

```
izvrši_stanje(stanje s, ...)
prikazi (stanje s)
čitaj (stanje s, ..)
ispravno (stanje s, ..)
obradi (stanje s, ...)
```

Kako što se vidi uloga stanja je prenaglašena. Trenutno stanje se pojavljuje kao argument svih rutina, počev od prvog modula. Hijerarhijska struktura, prikazana na poslednjoj slici, koja izgleda jednostavna i podesna za rukovanje, predstavlja smao fasadu. Iza formalne elegancije funkcionalne dekompozicije leži zbrka u prenosu podataka.

Pozadina primene objektne tehnologije je borba između funkcijskih aspekata i aspekta podataka. U pristupima koji nisu OO orijentisani funkcije vladaju podacima, ali to nije dobro. Prava slika ovog sistema, sa prenosom potrebnih podataka izgleda ovako:



U ovom primeru nedostatak strukture se ogleda u potrebi da se pravi razlika zavisno od stanja. Sve rutine u nivou 1 preduzimaju različite akcije u zavisnosti od stanja (prikaz odgovarajućeg prozora, čitanje odgovarajućih podataka, provera ispravnosti unosa itd.). Realizacija provere stanja se odvija kroz neku kontrolnu strukturu tipa ako je stanje to i to onda uradi to i to.

Ovo pretpostavlja dugačke i složene kontrolne strukture i sistem koji nije stabilan. Svako dodavanje stanja zahteva izmene i u samoj kontrolnoj strukturi. To je tipičan primer nekontrolisane distribucije znanja. Previše modula u sistemu se oslanja na neki podatak, u ovom slučaju listu mogućih stanja, koji je podložan promeni.

Ako bismo želeli da ovo rešenje primenimo i na druge vrste rezervacija, a ne samo za avionska putovanja, onda u igru ulazi još jedan implicitni argument, a to je aplikacija o kojoj se radi. Da bismo rutine, kao što je prikaži, napravili zaista opštim moramo ih obavestiti o svim stanjima u svim aplikacijama. Funkcija prelaz u tom slučaju sadrži graf prelaza za sve aplikacije. Ovo je naravno, nerealno.

## Objektno orijentisana arhitektura

Nedostaci funkcionalne dekompozicije odozgo nadole ukazuju na to šta moramo da uradimo da bismo dobili dobru objektno orijentisanu verziju.

Šta je pogrešno. Previše prenosa podataka u softverskoj arhitekturi obično ukazuje na nedostatke u dizajnu. Ovo se može izraziti sledećim pravilom:

*Ako rutina razmenjuje previše podataka, postavite je u podatke.*

Umesto da module pravimo oko operacija (kao što su izvrši_sesiju ili izvrši_stanje) i da između rutina distribuiramo podatke, kod objektno orijentisanog projektovanja se koristi obrnuti pristup. Najvažniji tipovi podataka se koriste za module, a rutina se pridružuje onom tipu podatka sa kojim je najuže vezana.

Prethodno navedeno pravilo je ključno kod prelaza sa funkcionalne dekompozicije na objektno orijentisanu. Takva potreba se javlja kod reverznog inženjeringa postojećih sistema. Isto se međutim može desiti i ako kod objektno orijentisanog projektovanja ne razmišljate na pravi način.

Analiza rpenosa podataka predstavlja dobar način za detekciju i ispravljanje propusta u objektno orijentisanom projektovanju. Ako u strukturi koja je zamišljena kao objektno orijentisana uočite da je šema podataka slična sa onime što se događa u ovom primeru, to bi trebalo da vam privuče pažnju. Dalje ispitivanje vas u većini slučajeva vodi do otkrića neke apstrakcije podataka koju ste zanemarili.

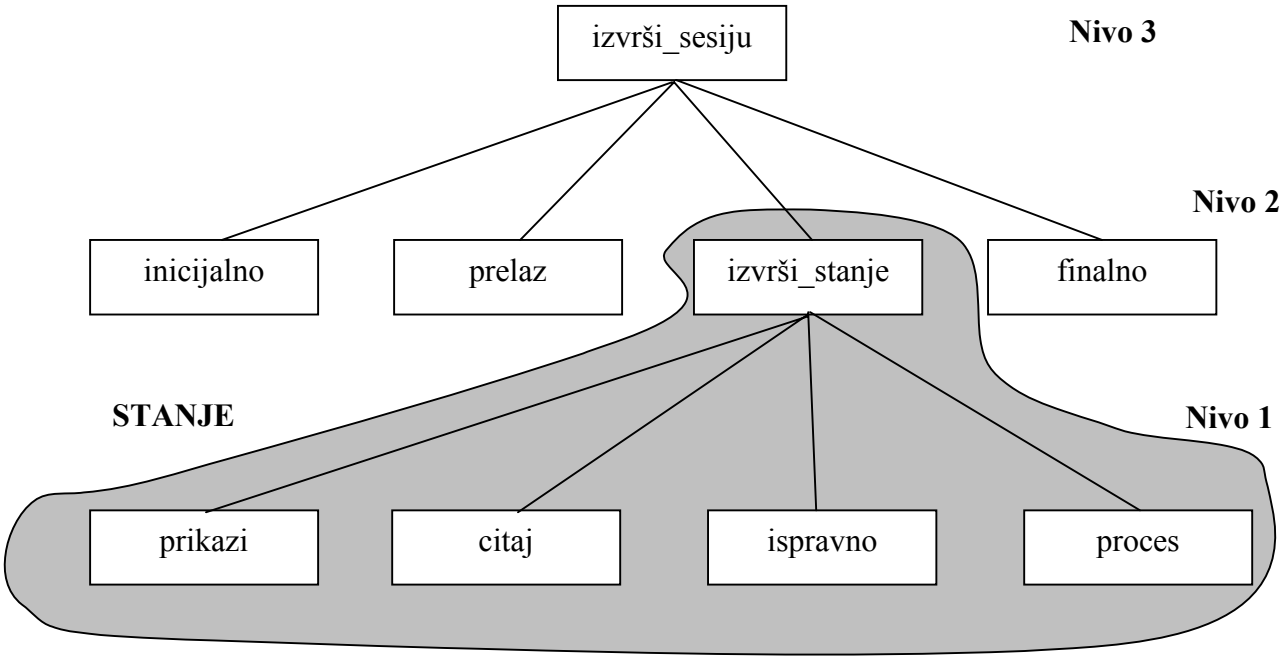
### *Stanje kao klasa*

Primer stanja je tipičan primer. Takav tip pdoatka, koji se često pojavljuje pri prenosu pdoataka između rutina je glavni kandidat da bude klasa.

Pojam stanja je veoma važan kod rešavanja originalnog problema, ali je u funkcionalnoj arhitekturi izgubio svoj značaj. Stanje je predstavljeno samo pomoću promenljive, koju rutine prosleđuju jedna drugoj, kao da je stanje neka niža vrsta. Stanje bi trebalo da bude klasa i to jedna od glavnih u strukturi objektno orijentisanog sistema.

U tu klasu ćemo smestiti sve operacije koje karakterišu stanje. To su prikazivanje odgovarajućeg ekrana (rutina prikazi), čitanje unosa korisnika (rutina čitaj), obrada tačnog odgovora (rutina obrada) i sl. Tu je i rutina izvrši_stanje, koja izražava niz akcija koje se vrše kada sesija dođe u određeno stanje.

Na osnovu originalne sliek iz funkcionalne dekompozicije odozgo nadole, možemo istaći skup rutina koje će se nalaziti u klasi STANJE.




Klasa bi mogla da ima sledeći oblik:

```
class STANJE {
   ulaz: ODGOVOR;
   izbor: int;
   izvrši()
   prikaži()
   čitaj()
   ispravni()
   obrada()
   ...
}
```

Atributi ove klase su ulaz i izbor, a ostalo su metodi. Ako ih uporedimo sa rutinama iz funkcionalne dekompozicije, vidimo da su rutine izgubile svoje eksplicitne argumente stanja. Stanje se ipak pojavljuje u pozivima ovih rutina, ali kao s.rutina.

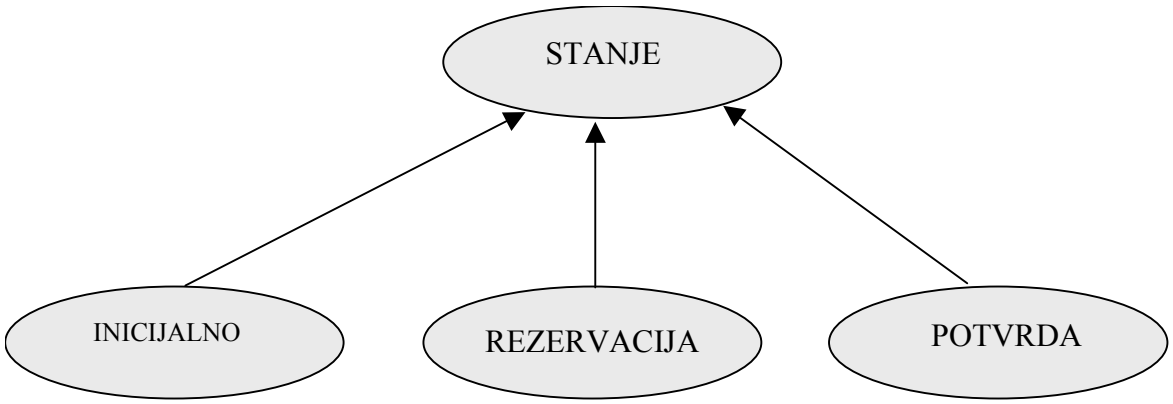
U prethodnom primeru rutina izvrši vraća izbor korisnika koji određuje sledeći korak. Takav pristup narušava pravila dobrog dizajna. Poželjnije je da se izvrši tretira kao komanda, čije izvršenje zavisi od rezultata uptia "koji je izbor korisnik napravio u prethodnom stanju". Ovo je dostupno preko atributa izbor.

### *Nasleđivanje*

Klasa STANJE ne opisuje određeno stanje, već opšti pojam stanja. Procedura izvrši je ista za sva stanja, ali su ostale rutine posebne za svako stanje.

U ovakvoj situaciji je idealno upotrebiti nasleđivanje i nepotpune klase. Na nivou klase STANJE proceduru izvrši znamo detaljno. Tu poznajemo i atribute. Takođe znamo da postoje rutine prvog nivoa (prikazi itd.), i njihove specifikacije, ali ne i njihove implementacije. Te rutine moraju biti nepotpune. Klasa STANJE koja opisuje skup varijanti, a ne potpunu apstrakciju je nepotpuna klasa.

Da bismo opisali specifično stanje uvešćemo naslednike klase STANJE koji implementiraju nepotpune metode.




Na primer klasa UPIT_O_LETOVIMA bi mogla da izgleda ovako:

```
class UPIT_O_LETOVIMA extends STANJE{
   void display(){
          specifična poruka za prikaz prozora
   }
ostali metodi
}
```

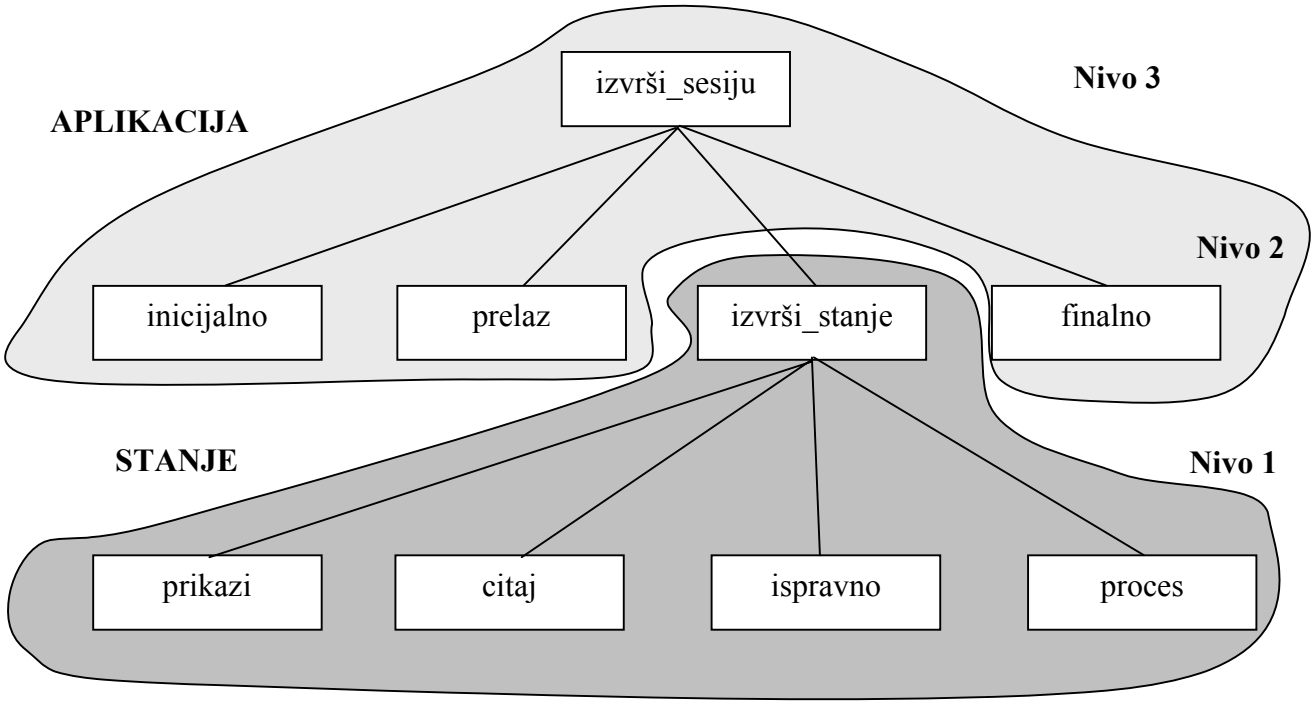
Ovakva arhitektura odvaja zajedničke elemente za sva stanja i elemente koji su specifični za dato stanje. Zajednički elementi, kao što je procedura izvrši() su koncentrisani u klasi STANJE i nema potrebe za njihovim navođenjem u naslednicima.

### *Opis celog sistema*

Da bismo upotpunili dizajn moramo voditi računa o rukovanju sesijom. Kod funkcionalne dekompozicije to je bio zadatak procedure izvrši_sesiju. Kako smo već pomenuli glavna funkcija sistema ne postoji. Sada je najbolje opet koristiti pristup apstraktnih podataka.

U našoj arhitekturi nedostaje još jedan bitan element, a to je pojam aplikacije, koji opisuje specifične interaktivne sisteme kao što je sistem rezervacije letova. Potrebna nam je nova klasa.

Ispostavlja se da preostale komponente funkcionalne dekompozicije, prikazane na slici, predstavljaju karakteristike aplikacije. Smestićemo ih u klasu APLIKACIJA. To su rutina izvrši_sesiju, rutina prelaz i sl.




Sve komponente funkcionalne dekompozicije su pronašle svoje mesto. Neke u klasi STANJE, a druge u klasi APLIKACIJA.

Sistem zasnovan na prozorima koji smo pručavali će uvek morati da ima operacije za obilazak dijagrama aplikacije (izvrši_sesiju, čitaj, prelaz i sl.). Duboko u strukturi pronaći ćemo iste gradivne elemente bez obzira na metod projektovanja koji smo primenili. Ono što se menja je način na koji ih grupišemo da bismo dobili modularnu strukturu.

Ono što je kod funkcionalne dekompozicije bio kraj rešenja, pisanje rutine izvrši_sesiju, sada je tek početak. Ako želimo možemo da aplikaciji dodamo još puno toga. Na primer:

metod za dodavanje novog stanja

dodavanje novog prelaza

uklanjanje stanja ili prelaza

uskladištenje cele aplikacije u bazu podataka

Sve navedene operacije predstavljaće karakteristike klase APLIKACIJA. One nisu ni više ni manje važne od onog što je bio glavni program (procedura izvrši_sesiju). Taj metod je sada samo jedan od metoda klase, koji više nije primaran. Time što smo se odrekli pojma vrha smo napravili prostor za evoluciju i višekratnu upotrebu.

## Zaključak

Primer koji smo pokazali ukazuje na prednosti objektno orijentisanog pristupa projektovanju. On naročito ukazuje na prednosti uklanjanja pojma glavnog programa. Ukoliko se usresredimo na apstrakcije podataka i z aboravimo, što je duže moguće, šta je glavna funkcija sistema, dobićemo strukturu koja je mnogo pogodnija za buduđe promene i za višekratnu upotrebu.

Prilikom primene metoda potrebno je malo discipline da bismo ga dosledno primenili. Ne treba pasti u iskušenje i postavljati pitanja tipa "Šta sistem radi?". To je ono što izdvaja pravog objektno

orijentisanog projektanta od onog ko samo koristi objektno orijentisane tehnike i jezik, ali i dalje razmišlja na funkcionalni način.

videli smo i neke smernice kako doći do ključnih apstrakcija u sistemu. Tu se misli na analizu prenosa podataka i traganje za pojmovima koji se pojavljuju u komunikaciji između elemenata sistema.

Takođe se moramo čuvati davanja prevelikog značaja mišljenju da su objektno orijentisani sistemi deirektno izvedeni iz "stvarnog sveta". Postoji mnogo načina za modeliranje stvarnosti, a neće svi dovesti do dobrog sistema. Prva verzija, zasnovana na naredbama got, bila je bliska realnom svetu isto kao i naredne, ali je zato bila katastrofa sa gledišta softverskog inženjeringa.

Objektno orijentisana dekompozicija koju smo na kraju napravili, klase STANJE, APLIKACIJA i sl. bila je dobra zato što su apstrakcije koje se krosite jasne, jednostvne z arukovanje, spremne za promene i mogu se upotrebljavati i u drugim aplikacijama. Možda će se u početku ove klase učiniti manje bliskim realnom svetu, ali ako ih dobro upoznate, one postaju realne kao i svi drugi pristupi.

Da biste proizveli dobar softver, nije važno koliko ste blizu nečijem opažanju realnog sveta, već koliko su dobre apstrakcije koje ste odabrali za modeliranje i strukturianje softvera. To je ono što razlikuje profesionalca od amatera, pronalaženje pravih apstrakcija.