1 Zašto softversko inženjerstvo?

U ovom poglavlju obrađivaćemo:
- šta se podrazumeva pod softverskim inženjerstvom;
- dostignuća softverskog inženjerstva;
- šta podrazumevamo pod pojmom „dobar softver";
- zašto je važan sistemski pristup;
- kako se softversko inženjerstvo promenilo od 1970. godine.

Softver prožima sve pore sveta koji nas okružuje, ali mi ponekad nismo ni svesni koliko on čini naš život udobnijim i uspešnijim. Na primer, posmatrajmo jednostavne radnje koje koristimo za pripremanje tosta za doručak. Programski kôd koji se nalazi u tosteru kontroliše koliko će hleb da se prepeče i kada će iskočiti iz aparata. Programi kontrolišu i regulišu isporuku električne energije u našem domu, a softver obračunava njen utrošak. U stvari, možemo da koristimo automatizovane programe za plaćanje računa za struju, naručivanje namirnica, čak i za kupovinu novog tostera! U današnje vreme rad softvera, kako onaj koji je svima vidljiv, tako i onaj koji je sakriven, prisutan je u gotovo svim aspektima našeg života, uključujući i sisteme od presudnog značaja za naše zdravlje i sveopšte stanje. Iz tog razloga, softversko inženjerstvo je danas važnije nego ikada. Dobra softver-inženjerska praksa mora da obezbedi da softver daje pozitivan doprinos načinu na koji živimo.

Ova knjiga stavlja u prvi plan ključna pitanja koja se postavljaju u softverskom inženjerstvu, opisujući naše znanje o tehnikama i alatima i kako oni utiču na konačne proizvode koje gradimo i koristimo. Napravićemo pregled i teorije i prakse: onoga šta znamo i kako to primenjujemo u tipičnom projektu gde se softver razvija ili održava. Takođe ćemo istražiti i stvari koje do sada nismo znali, ali koje mogu da nam pomognu da naše proizvode učinimo pouzdanijim, bezbednijim, korisnijim i pristupačnijim.

Počećemo pregledom kako analiziramo probleme i kako realizujemo rešenja. Onda ćemo istražiti razlike između problema koji se javljaju u računarskoj nauci i softverskom inženjerstvu. Naš krajnji cilj je da proizvedemo rešenja koja uključuju softver visokog kvaliteta, i razmotrimo karakteristike koje doprinose kvalitetu.

Takođe ćemo videti koliko smo bili uspešni u razvoju softverskih sistema. Istražujući nekoliko primera softverskih otkaza, videćemo gde se nalazimo na putu ovladavanja majstorstvom razvoja kvalitetnog softvera.

Nakon toga, pažnju ćemo posvetiti učesnicima u razvoju softvera. Nakon što opišemo uloge i odgovornosti kupca, korisnika i učesnika u razvoju, okrenućemo se proučavanju samog sistema. Videćemo da sistem može da se predstavi kao omeđena grupa objekata povezanih skupom aktivnosti. Alternativno, sistem možemo posmatrati očima inženjera; razvijanje sistema ima dosta sličnosti sa zidanjem kuće. Kada definišemo korake potrebne za izgradnju sistema, razmotrićemo uloge razvojnog tima u svakom od tih koraka.

Na kraju, razmotrićemo neke promene koje su uticale na način praktičnog sprovođenja softverskog inženjerstva. Predstavićemo osam Wassermanovih ideja da bismo praksu i ideje povezali u jedinstvenu koherentnu celinu.

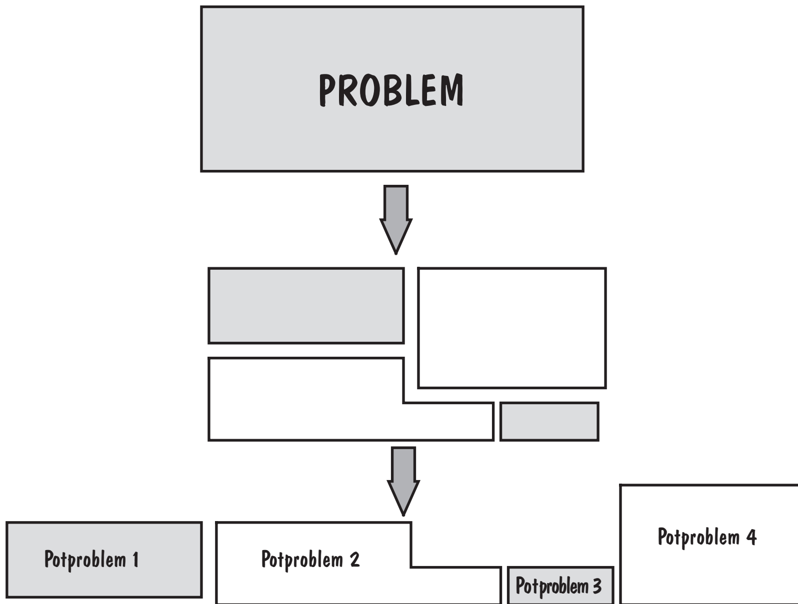
1.1 ŠTA JE SOFTVERSKO INŽENJERSTVO

Kao softverski inženjeri, koristimo naše poznavanje računara i računarstva kao pomoć prilikom rešavanja problema. Često je problem kojim se bavimo povezan sa računarom ili postojećim računarskim sistemom, ali ponekad poteškoće na kojima on počiva nemaju dodirnih tačaka sa računarima. Stoga je osnovno da prvo razumemo prirodu problema. Zato posebno moramo paziti da ne namećemo računare ili računarske tehnike kao rešenje svakog problema sa kojim se susretnemo. Prvo je potrebno da rešimo problem. Zatim, ako je potrebno, možemo da koristimo tehnologiju kao sredstvo za implementaciju rešenja. U daljim razmatranjima u sklopu ove knjige, pretpostavićemo da je naša analiza pokazala da je neka vrsta računarskog sistema neophodna ili poželjna za rešavanje problema koji razmatramo.

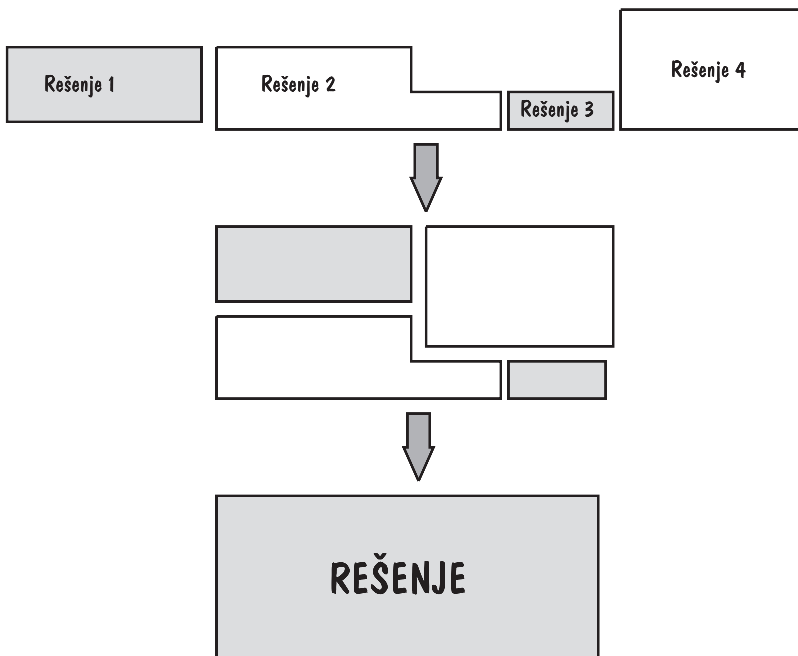
Rešavanje problema

Mnogi problemi su obimni i nekada su nezgodni, posebno ako predstavljaju nešto novo što nikada ranije nije rešavano. Stoga istraživanje problema moramo započeti **analizom**, tj. razlaganjem problema na delove koje možemo da razumemo i sa kojima ćemo pokušati da se izborimo. Na osnovu toga možemo jedan veliki problem opisati kao skup manjih međusobno povezanih problema. Slika 1.1 ilustruje kako se vrši analiza. Važno je da zapamtimo da su veze (na slici su to strelice, uz relativne položaje potproblema) jednako važne kao i sami potproblemi. Ponekad, upravo su relacije, a ne priroda potproblema, ključni faktori u iznalaženju rešenja složenog problema.

Po obavljenoj analizi problema, sledi sastavljanje kompletnog rešenja iz delova koji se odnose na različite aspekte problema. Slika 1.2 ilustruje ovaj obrnuti proces: **Sinteza** je sklapanje veće strukture na osnovu manjih gradivnih elemenata. Kao i u slučaju analize, sklapanje pojedinačnih rešenja može biti jednako izazovno kao i postupak traženja rešenja. Da bismo videli zašto je to tako, poslužićemo se analogijom sa pisanjem romana. Rečnik sadrži sve reči koje možda želite da upotrebite tokom pisanja. Ali najteži deo pisanja jeste odlučivanje kako da organizujemo i ukomponujemo reči u rečenice, odnosno rečenice u pasuse i poglavlja, da bismo na kraju uobličili celokupnu knjigu. Stoga, svaka tehnika rešavanja problema mora da sadrži dva dela: analizu problema radi određivanja njegove prirode i, na analizi zasnovanu, sintezu rešenja.



SLIKA 1.1 Postupak analize



SLIKA 1.2 Postupak sinteze

Da bismo pomogli sebi u rešavanju problema, primenićemo raznolike metode, alate, procedure i paradigme. **Metod** ili **tehnika** je formalna procedura za pravljenje nekog ishoda. Na primer, glavni kuvar može da pripremi preliv pomoću niza sastojaka kombinovanih u pažljivom redosledu i odabranim trenucima tako da se preliv zgusne, ali da se ne zgruša ili da ostane nesjedinjen. Postupak pripreme preliva uključuje merenje vremena i sastojke, ali može da zavisi i od vrste opreme koja se koristi za kuvanje.

Alat je instrument ili automatizovani sistem koji pomaže da se nešto obavi na bolji način. Taj „bolji način" može značiti da zahvaljujući alatu radimo preciznije, efikasnije ili delotvornije, ili dobijamo bolji kvalitet rezultujućeg proizvoda. Na primer, koristimo pisaću mašinu ili tastaturu i štampač za pisanje pisama jer je rezultujući dokument čitljiviji nego rukopis. Ili koristimo makaze kao alat jer možemo da sečemo brže i pravije nego kad rukom cepamo list papira. Međutim, alat nije uvek neophodan da bismo nešto napravili dobro. Na primer, bolji preliv može da se napravi zahvaljujući boljoj tehnici kuvanja a, ne zato što kuvar koristi bolju šerpu ili kutlaču.

Procedura je kao recept: kombinacija alata i tehnika, koji, u međusobnom skladu, proizvode dati proizvod. Na primer, kao što ćemo videti u kasnijim poglavljima, planovi za testiranje opisuju procedure za testiranje; oni nam govore koji alati treba koristiti nad kojim skupovima podataka i pod kojim okolnostima, da bismo odredili da li naš softver zadovoljava postavljene zahteve.

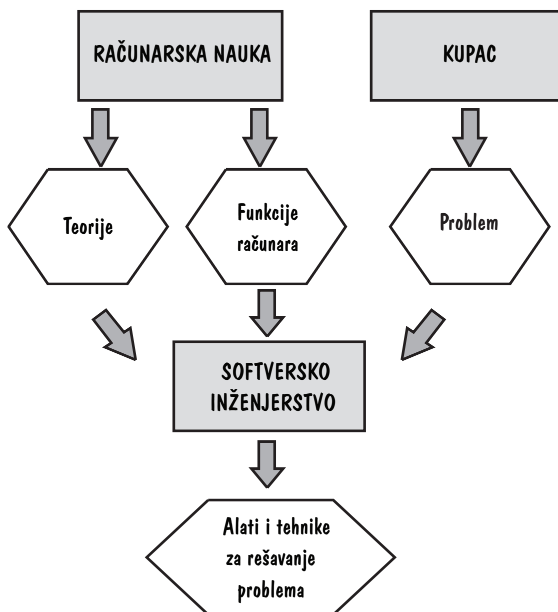
Na kraju, **paradigma** je kao stil kuvanja; ona predstavlja poseban pristup ili filozofiju gradnje softvera. Kao što jednostavno razlikujemo francusku kuhinju od kineske, na isti način razlikujemo i paradigme, kao što uostalom postoji i razlika između proceduralnog i objektno orijentisanog razvoja. Jedan nije bolji od drugog; svaki ima svoje prednosti i mane, a postoje situacije u kojima je jedan od njih pogodniji.

Softverski inženjeri koriste alate, tehnike, procedure i paradigme da bi poboljšali kvalitet softverskih proizvoda. Njihov cilj je da koriste efikasne i produktivne pristupe za generisanje efektivnog rešenja problema. U poglavlju koje sledi, biće istaknuti posebni pristupi koji služe kao podrška opisanim aktivnostima razvoja i održavanja. Na pratećoj veb stranici za ovu knjigu nalazi se ažurni skup pokazivača na alate i tehnike.

Gde se tu uklapa softversko inženjerstvo?

Da bismo razumeli kako se softverski inženjer uklapa u svet računarske nauke, pogledaćemo kao primer jednu drugu disciplinu. Uzmimo, na primer, hemiju i način na koji se u hemiji rešavaju problemi. Hemičar istražuje hemikalije: njihovu strukturu, uzajamno delovanje i teorijsku osnovu njihovog ponašanja. Hemijski inženjeri primenjuju rezultate hemijskog proučavanja na razne probleme. Za hemičare je hemija kao predmet proučavanja, ali je s druge strane, za hemijskog inženjera, hemija je i alat koji se koristi za bavljenje nekim opštim problemom (koji možda čak i nije „hemijski" po svojoj prirodi).

Računarstvo možemo da posmatramo u sličnom svetlu. Možemo da se usredsredimo na računare i programske jezike, ili možemo da ih gledamo kao alate koji se koriste u projektovanju i primeni rešenja nekog problema. Softversko inženjerstvo kao što je prikazano na slici 1.3. predstavlja ovaj drugi pogled. Umesto da istražujemo dizajn hardvera ili da dokazujemo teoreme koje opisuju rad algoritama, softverski inženjer se usredsređuje na računar kao sredstvo za rešavanje problema. Videćemo kasnije u ovom poglavlju da softver-inženjer koristi računarske funkcije kao delove generalnog rešenja, umesto da se bavi organizacijom ili teorijom rada računara.



SLIKA 1.3 Odnos između računarske nauke i softverskog inženjerstva

1.2 KOLIKO SMO BILI USPEŠNI?

Pisanje softvera je i umetnost i nauka, i naročito je važno da to shvate studenti računarskih nauka. Računarski naučnici i softver-inženjeri istraživači proučavaju računarske mehanizme i teorije da bi ih učinili produktivnijim ili efikasnijim. Međutim, oni takođe projektuju računarske sisteme i pišu programe za obavljanje poslova na tim sistemima, praktični rad koji je dobrim delom umetnost, domišljatost i veština. Postoji više načina da se određeni zadatak izvede na određenom sistemu, pri čemu su neki načini bolji od drugih. Jedan način je možda efikasniji i precizniji, lakše se modifikunje, koristi ili razume. Svaki haker može da napiše programski kôd koji nešto radi, ali su potrebni umeće i znanje profesionalnog softverskog inženjera da bi se proizveo stabilan i razumljiv kôd koji se lako održava i koji efikasno i efektivno radi ono zbog čeka je napravljen. Prema tome, kod softverskog inženjeringa suština je u projektovanju i razvoju visoko kvalitetnog softvera.

Pre nego što istražimo šta je potrebno za pravljenje kvalitetnih softverskih sistema, osvrnućemo se na to koliko smo bili uspešni do sada. Da li su korisnici zadovoljni sa postojećim softverskim sistemima? I da i ne. Softver nam je omogućio da radimo poslove brže i efikasnije nego ikada do sada. Prisetimo se kako smo živeli pre pojave, na primer, programa za obradu teksta, radnih tabela, elektronske pošte ili sofisticirane telefonije. Softver podržava održavanje i spasavanje života u medicini, poljoprivredi, transportu i većini drugih industrijskih grana. Pored toga, softver je omogućio da primenjujemo stvari koje nekada nismo ni mogli da zamislimo: mikrooperativne zahvate, multimedijalno obrazovanje, robotiku i mnoge druge.

Međutim, softver nije bez vlastitih problema. Sistemi često ne funkcionišu baš na očekivani način. Svi smo čuli priče o sistemima koji jedva da rade. Svi mi smo pisali programe u kojima je bilo grešaka: kôd koji sadrži greške, ali je dovoljno dobar da položimo

ispit ili da demonstriramo izvodljivost nekog pristupa. Jasno je da takvo ponašanje nije prihvatljivo kada razvijamo sistem koji treba da se isporuči kupcu.

Postoji ogromna razlika između greške u školskom projektu i one u velikom softverskom sistemu. U stvari, u literaturi i kuloarima se često razmatraju softverske greške i poteškoće u izradi softvera koji ne sadrži greške. Neke greške su zaista dosadne, dok nas druge koštaju dosta vremena i novca. Ima i onih koje su opasne po život. Podsetnik 1.1 objašnjava odnose između nedostataka, grešaka i otkaza. Pogledajmo nekoliko primera otkaza gde ćemo videti šta je pošlo naopako i zašto.

PODSETNIK 1.1 TERMINOLOGIJA ZA OPISIVANJE „PROGRAMSKIH BUBICA”

Često govorimo o „bubicama” u softveru, i pri tome mislimo na više različitih stvari, zavisno od konteksta. Pojam „bubica” može da bude pogreška u tumačenju programskog zahteva, sintaksna greška u delu koda ili (do tada nepoznat) uzrok pada sistema. Organizacija IEEE je sugerisala standardnu terminologiju (u IEEE standardu 729) za opisivanje „bubica” u softverskim proizvodima (IEEE 1983).

Nedostatak se događa kada ljudsko biće nešto zabrlja u izvođenju neke softerske aktivnosti, i to nazivamo **greška**. Na primer, projektant može loše da razume programski zahtev i osmisli program koji ne odgovara stvarnoj potrebi analitičara i korisnika. Ovaj nedostatak u projektu je podloga za grešku, pa može da dovede do drugih nedostataka, kao što su neispravan kôd ili netačan opis u korisničkom uputstvu. Stoga vidimo da jedna greška može da generiše više nedostataka, pri čemu nedostatak može da se nalazi u svakom razvijanom ili održavanom proizvodu.

Otkaz je odstupanje od neophodnog ponašanja sistema. Može biti otkriven pre ili nakon isporuke sistema, tokom testiranja, tokom korišćenja ili održavanja. Kako dokumenti sa programskim zahtevima mogu da sadrže nedostatke, otkazi ukazuju da se sistem ne ponaša kao što je neophodno, iako se možda ponaša u skladu sa specifikacijom.

Stoga nedostatak predstavlja unutrašnji pogled na sistem, kako ga vide oči učesnika u razvoju, dok otkaz predstavlja pogled na sistem spolja, tj. problem kakvog vidi korisnik. Svaki nedostatak nema uvek odgovarajući otkaz. Na primer, ako se programski kôd sa nedostatkom nikada ne izvršava ili se u to posebno stanje nikada ne ulazi, tada nedostatak nikada neće prouzrokovati otkaz. Slika 1.4 prikazuje genezu otkaza.

SLIKA 1.4 Kako ljudske greške prouzrokuju otkaze

U ranim 1980-im, američka poreska uprava angažovala je korporaciju Sperry za izgradnju sistema za automatsku obradu obrazaca za savezni porez na prihode. Prema onome što je objavio dnevnik *Washington Post*, „sistem ... je dokazano neodgovarajući za takvo radno opterećenje, koštao je skoro duplo nego što smo očekivali i mora biti uskoro zamenjen" (Sawyer 1985). U 1985. godini, dodatnih 90 miliona dolara bilo je potrebno za poboljšanje početne Sperryjeve opreme vredne 103 miliona dolara. Dodatno, jer je postojeći problem sprečio poresku upravu da refundira poreske obveznike u predviđenom roku, pa su bili prinuđeni da plati 40,2 miliona dolara kamate i 22,3 miliona nadoknade za prekovremeni rad svojih radnika koji su pokušavali da sustignu rokove. U 1996. godini situacija se nije poboljšala. Dnevne novine *Los Angeles Times* izvestile su 29. marta da još uvek ne postoji glavni plan za modernizaciju računara poreske uprave, već samo tehnički dokument od 6 000 stranica. Kongresmen Jim Lightfoot nazvao je taj projekat „fijasko od 4 milijarde dolara, koji je zapeo zbog neodgovarajućeg planiranja" (Vartabedian 1996). U poglavlju 2 videćemo zašto je planiranje projekta bitno za proizvodnju kvalitetnog softvera.

Više godina je javnost bez pogovora prihvatala upliv softvera u njenu svakodnevicu. Međutim, strateška odbrambena inicijativa (*Strategic Defense Initiative*, SDI), koju je predložio predsednik Regan, podigla je svest javnosti o teškoćama proizvodnje softverskog sistema koji ne sadrži greške. Izveštaji popularnih dnevnih novina i časopisa (kao što su Jacky 1985, Parnas 1985 i Rensburger 1985) izražavali su skepticizam u računarskoj zajednici. A sada, 20 godina kasnije, kako je od američkog kongresa traženo da dodeli sredstva za gradnju sličnog sistema, mnogi računarski naučnici i softverski inženjeri nastavili su da veruju da ne postoji način pisanja i testiranja softvera koji garantuje adekvatnu pouzdanost.

Na primer, mnogi softverski inženjeri misle da bi jedan antibalistički raketni sistem zahtevao makar 10 miliona redova koda; neki procenjuju red veličine čak sto miliona redova. Radi poređenja, softver koji služi za podršku američkoj svemirskoj letelici šatl sastoji se od 3 miliona redova koda, uključujući i zemaljske računare koji kontrolišu ispaljivanje rakete i let. Godine 1985. bilo je u samom šatlu 100 000 redova koda (Rensburger 1985). Stoga bi protivraketni softverski sistem zahtevao testiranje enormne količine koda. Povrh svega, ograničenja u pogledu pouzdanosti bilo bi nemoguće testirati. Da bismo videli zašto je to tako, razmotrićemo pojam bezbednosno kritičnog softvera. Obično kažemo da nešto što je **važno za bezbednost** (tj. nešto čiji otkaz predstavlja pretnju po život ili zdravlje), treba da ima pouzdanost od najmanje 10^{-9}. Kao što ćemo videti u poglavlju 9, to znači da sistem sme da otkaže samo jedanput u 10^9 časova rada. Ali, 10^9 časova je preko 114 000 godina, suviše dugo kao interval za testiranje.

Videćemo takođe u poglavlju 9 da inače korisna tehnologija može postati smrtonosna ako je softver projektovan ili programiran na neodgovarajući način. Na primer, medicinski krugovi bili su zaprepašteni kada je Therac-25, mašina za radio terapiju i rendgen, loše funkcionisala i ubila više pacijenata. Projektanti softvera jednostavno nisu predvideli istovremenu upotrebu više tastera sa strelicama na nestandardni način. Posledica toga je bila da je softver zadržavao visoke vrednosti parametara i emitovao visoko koncentrovane doze radijacije u situacijama kada su zahtevane male doze (Leveson i Turner 1993).

Sličan primer nepredviđene primene i njenih opasnih posledica opisan je u časopisu *Pilot*, o izveštaju podnesenom na Risks Forumu (*Pilot* 1996). Dva policajca u Škotskoj

koristili su pištolj-radar za registrovanje brzine vozila. Radar se neočekivano zaglavio, prikazujući brzinu veću od 300 milja na čas. Nekoliko sekundi kasnije, proleteo je nisko-leteći mlaznjak Harrier. Tragač meta na lovcu Harrier prepoznao je radar i shvatio da pripada „neprijatelju". Srećom, Harrier nije bio naoružan, jer bi uobičajeno ponašanje bilo da ispali raketu sa automatskim navođenjem!

Neočekivana upotreba sistema mora da se uzme u obzir u toku razvoja softvera. To se može obaviti na dva načina: zamišljanjem kako sve sistem može biti zloupotrebljavan (i pravilno upotrebljavan), kao i pretpostavljanjem da će sistem biti zloupotrebljavan, i projektovanje softvera za rukovanje takvim zloupotrebama. Ti pristupi biće razmatrani u poglavlju 8.

I pored toga što mnogi proizvođači teže softveru bez mana, u stvarnosti mnogi softverski proizvodi sadrže nedostatake. Tržište diktira brzi razvoj softvera jer je cilj da se proizvodi brzo isporuče, što ostavlja malo vremena za temeljno testiranje. Tim za testiranje po pravilu može da testira jedino one funkcije koje će se najverovatnije koristiti, ili one koje bi najverovatnije mogle da ugroze, odnosno iritiraju korisnike. Zato su mnogi korisnici opravdano zabrinuti kada instaliraju prvu verziju koda, znajući da programske greške neće biti ispravljene sve do druge verzije. Nadalje, nekada je teže realizovati modifikacije potrebne za ispravljanje uočenih nedostataka, nego ponovo napisati kompletan softver od početka. U poglavlju 11 istražićemo pitanja vezana za održavanje softvera.

Uprkos nekim spektakularnim uspesima i sveopštem prihvatanju softvera kao sastavnog dela života, još uvek ima dosta prostora za njegovo poboljšanje. Na primer, odsustvo kvaliteta je skupo, jer što duže nedostatak ostaje neotkriven, njegovo otklanjanje više košta. Procene pokazuju da trošak ispravljanja greške u fazi analize projekta, predstavlja svega jednu desetinu cene ispravljanja iste greške nakon isporuke sistema korisniku. Nažalost, većina grešaka se ne otkriva u ranoj fazi. Na greške napravljene rano u životnom ciklusu sistema odlazi polovina troškova ispravljanja grešaka otkrivenih tokom testiranja i održavanja. U poglavljima 12 i 13 razmotrićemo načine za ocenjivanje efikasnosti razvojne aktivnosti i načine za poboljšanje procesa u cilju što ranijeg otkrivanja grešaka.

Jedna od jednostavnih, ali moćnih tehnika koju ćemo predložiti jesu pregledi i inspekcije. Mnogi studenti su navikli na samostalni razvoj i testiranje softvera. Međutim, njihovo samostalno testiranje ume da bude neefikasnije nego što to oni misle. Na primer, Fagan je proučavao način na koji se otkrivaju nedostaci. On je otkrio da testiranje programa njegovim izvršavanjem na uzorku podataka predviđenom za testiranje otkriva samo petinu od svih ukupno otkrivenih nedostataka tokom razvoja sistema. Preostale četiri petine otkrivaju se neposrednim pregledanjem koda, odnosno kada kolege ispituju i komentarišu jedni drugima način projektovanja i programski kôd (Fagan 1986). To znači da kvalitet softvera može značajno da se poveća ako kolege jednostavno pregledaju naš rad. U kasnijim poglavljima pokazaćemo kako postupak pregledanja i inspekcije može da se koristi nakon svakog glavnog razvojnog koraka u cilju otklanjanja nedostataka što je ranije moguće. Videćemo u poglavlju 13 kako je moguće unaprediti i sam postupak inspekcije.

1.3 ŠTA JE DOBAR SOFTVER?

Kao što proizvođači traže načine da osiguraju kvalitet proizvoda koji prave, isto tako softverski inženjeri moraju da pronalaze metode za obezbeđivanje prihvatljivog kvaliteta i korisnosti njihovih proizvoda. Stoga dobro softversko inženjerstvo mora uvek da uključuje i strategiju za proizvodnju kvalitetnog softvera. Ali pre nego što smislimo strategiju, moramo da shvatimo šta se podrazumeva pod kvalitetnim softverom. Podsetnik 1.2 prikazuje kako gledište utiče na ono što zovemo „kvalitet". U ovom odeljku ispitaćemo po čemu se dobar softver razlikuje od lošeg.

PODSETNIK 1.2 GLEDIŠTA NA KVALITET

Garvin (1984) je razmatrao kako različiti ljudi doživaljavaju kvalitet. On opisuje kvalitet sa pet različitih gledišta:

- *transcendentalni pogled*, gde je kvalitet nešto što možemo da prepoznamo ali ne i da definišemo;
- *korisnički pogled*, gde je kvalitet usklađenost sa namenom;
- *pogled sa aspekta proizvodnje*, gde je kvalitet usklađenost sa specifikacijom;
- *pogled sa aspekta proizvoda*, gde je kvalitet vezan za karakteristike samog proizvoda;
- *pogled na bazi vrednosti*, gde kvalitet zavisi od toga koliko je kupac spreman da za njega plati.

Transcendentalni pogled je vrlo sličan Platonovom opisu ideala ili Aristotelovom konceptu oblika. Drugim rečima, kao što je svaki stvarni sto aproksimacija idealnog stola, tako je i kvalitet softvera u stvari ideal kome stremimo, ali koji možda nikada nećemo u potpunosti ostvariti.

Transcendentalni pogled je prilično metafizički, nasuprot mnogo konkretnijem pogledu koji ima korisnik. Korisnički pogled prihvatamo kada merimo karakteristike proizvoda, kao što su npr. gustina nedostataka ili pouzdanost, u cilju razumevanja sveukupnog kvaliteta proizvoda.

Pogled sa aspekta proizvodnje posmatra kvalitet u toku procesa izrade i nakon isporuke. U stvari, ovaj pogled proverava da li je proizvod inicijalno sagrađen ispravno, čime se izbegavaju skupe prepravke radi otklanjanja isporučenih nedostataka. Stoga fabrički pogled predstavlja u stvari pogled sa stanovišta postupka, tj. zagovara saobraznost dobrom postupku. Međutim, malo je dokaza koji govore o tome da strogo sprovođenje postupka daje proizvode sa manje nedostataka i otkaza. Ispravan postupak zaista može da ka visoko kvalitetnom proizvodu, ali može i da institucionalizuje proizvodnju osrednjeg kvaliteta. Neka od tih pitanja biće istražena u poglavlju 12.

Korisnički i proizvođački pogled posmatraju proizvod spolja, ali pogled sa aspekta proizvoda posmatra stvari iznutra i ocenjuje karakteristike samog proizvoda. Ovaj pogled najčešće zastupaju stručnjaci za softversku metriku. Oni podrazumevaju da će dobri interni pokazatelji kvaliteta dovesti i do dobrih eksternih pokazatelja kvaliteta, kao što su npr. pouzdanost i lako održavanje. Međutim, potrebno je još istraživanja za proveru tih pretpostavki i za određivanje koja gledišta kvaliteta utiču na stvarnu upotrebu proizvoda. Možda ćemo morati da razvijemo modele koji povezuju pogled proizvoda sa korisničkim pogledom.

Kupci ili prodavci često usvajaju korisnički pogled na kvalitet. Istraživači ponekad posmatraju stvari sa aspekta proizvoda, dok razvojni tim posmatra sa aspekta procesa proizvodnje. Ako razlike u gledištima nisu izričito naglašene, onda konfuzija i nerazumevanje mogu da dovedu do loših odluka i loših proizvoda. Pogled na bazi vrednosti može da poveže te sasvim različite poglede na kvalitet. Stavljanjem znaka jednakosti između kvaliteta i onoga šta je kupac voljan da plati, možemo sagledati kompromis koji treba napraviti između cene i kvaliteta, i na osnovu njega razrešavati konfliktne situacije, kada se pojave. Slično tome, kupci porede cenu proizvoda sa potencijalnom korišću, razmišljajući o kvalitetu kao vrednosti koju dobijamo u zamenu za novac.

Kitchenham i Pfleeger (1996) su istraživali odgovor na to pitanje u uvodniku za specijalno izdanje časopisa *IEEE Software*, posvećeno kvalitetu. Oni naglašavaju da kontekst pomaže da se utvrdi odgovor. Nedostaci koji mogu da se tolerišu u softveru za obradu teksta, ne mogu da se prihvate u sistemima gde je bezbednost faktor od izuzetnog značaja, ili onima sa kritičnom misijom. Stoga, kvalitet moramo posmatrati na najmanje tri načina: kvalitet proizvoda, kvalitet postupka izrade proizvoda, i kvalitet proizvoda u kontekstu poslovnog okruženja u kojem će se on koristiti.

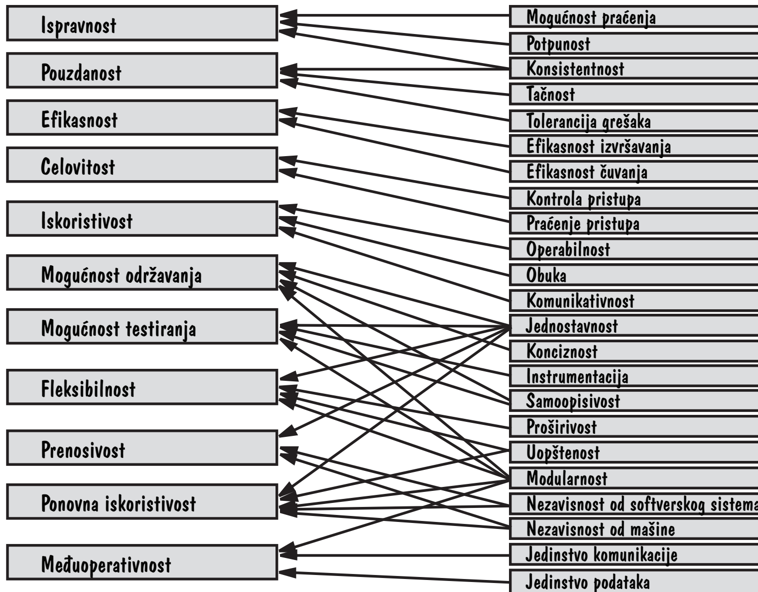
Kvalitet proizvoda

Ako pitamo različite ljude da navedu karakteristike koje učestvuju u ukupnom kvalitetu softvera, verovatno ćemo dobiti različite odgovore od svakog. Razlike se javljaju zato što značaj neke karakteristike zavisi od toga ko analizira softver. Korisnici smatraju da je softver visokog kvaliteta ako radi na način na koji oni žele, ako se lako uči i koristi. Međutim, nekada su kvalitet i finkcionalnost isprepleteni. Ako je nešto teško naučiti ili koristiti, ali ima funkcionalnost koja je vredna truda, onda visok kvalitet nije sporan.

Kvalitet softvera pokušavamo da merimo poređenjem jednog proizvoda sa drugim. Da bismo to postigli, identifikujemo one aspekte sistema koji doprinose njegovom ukupnom kvalitetu. Otuda, kada mere kvalitet softvera, korisnici ocenjuju spoljašnje karakteristike kao što su broj otkaza i tip otkaza. Na primer, oni mogu da klasifikuju otkaze kao minorne, glavne i katastrofične, i nadaju se da se događaju samo minorni otkazi.

Softver takođe moraju da procenjuju oni koji projektuju i pišu kôd i oni čiji je zadatak kasnije održavanje programa. Ovi praktičari teže razmatranju internih karakteristika proizvoda, nekada čak i pre nego što je proizvod isporučen korisniku. Detaljnije, praktičari često posmatraju broj i tip nedostataka kao dokaz kvaliteta proizvoda (ili kao dokaz nedostatka kvaliteta). Na primer, programeri prate broj nedostataka koji se nalaze u programskim zahtevima, projektu i inspekcijama koda i koriste ih kao indikatore željenog kvaliteta finalnog proizvoda.

Zatočesto pravimo modele koji dovode u vezu spoljašnji pogled korisnika i unutrašnji pogled programera na softver. Slika 1.5 je primer jednog od prvih modela kvaliteta, koji je napravio McCall sa svojim kolegama da bi pokazao u kakvom su odnosu spoljašnji faktori kvaliteta (na levoj strani) sa kriterijumima kvaliteta proizvoda (na desnoj stra-



SLIKA 1.5 McCallov model kvaliteta

ni). McCall je svakom kriterijumu sa desne strane pridružio merenje radi prikazivanja stepena uticaja na posmatrani element kvaliteta (McCall, Richards i Walters 1977). U poglavlju 12 ispitaćemo nekoliko modela kvaliteta proizvoda.

Kvalitet procesa

Postoji mnogo aktivnosti koje imaju uticaja na konačni kvalitet proizvoda. Ako neka od tih aktivnosti pođe naopako, to može da pogorša kvalitet proizvoda. Iz tog razloga, mnogi softverski inženjeri osećaju da je kvalitet postupka razvoja i održavanja važan jednako kao i kvalitet proizvoda. Jedna od prednosti modelovanja postupka jeste da možemo da ga analiziramo i nađemo načine da ga poboljšamo. Na primer, možemo da postavimo pitanja kao što su:

- Gde i kada ćemo verovatno da nađemo određenu vrstu nedostatka?
- Kako možemo što ranije da pronađemo nedostatke u postupku razvoja?
- Kako možemo da ugradimo toleranciju na greške, da bismo smanjili verovatnoću da nedostatak pređe u otkaz?
- Da li postoje alternativne aktivnosti koje mogu da načine proces efektivnijim ili efikasnijim, uz osiguranje kvaliteta?

Ova pitanja mogu da se primene na celokupan proces razvoja ili pojedinačne potpostupke, kao što su upravljanje konfiguracijom, ponovno korišćenje ili testiranje. Te postupke istražićemo u kasnijim poglavljima.

U 1990-im, značajan publicitet dobili su modelovanje i poboljšanje procesa u softverskom inženjerstvu. Standardi procesa inspirisani su radom Deminga i Jurana, a primenjivale su ih razne kompanije, između ostalih i IBM. Neki od standarda, kao što su CMM (*Capability Maturity Model*), ISO 9000 i SPICE *(Softvare Process Improvement and Capability dEtermination*) sugerišu da unapređenjem postupka razvoja softvera, možemo da poboljšamo kvalitet rezultujućeg proizvoda. U poglavlju 2 videćemo kako možemo da identifikujemo relevantne aktivnosti postupka i kako da modelujemo njegove efekte na poluproizvode i konačni proizvod. Poglavlja 12 i 13 detaljno ispituju modelovanje postupka i okvire poboljšanja.

Kvalitet u kontekstu poslovnog okruženja

Kada se u žiži ocenjivanja kvaliteta nalaze proizvodi i postupci, kvalitet obično merimo matematičkim izrazima koji uključuju nedostatke, otkaze i vreme. Ređe se opseg proširuje uključivanjem aspekata poslovanja, gde se kvalitet posmatra zavisno od proizvoda i usluga koje pruža poslovni sistem čiji je softver sastavni deo. Drugim rečima, posmatramo tehničku vrednost naših proizvoda i jedino na osnovu njih donosimo odluke, umesto da ih posmatramo sa mnogo šireg aspekta kakav je njihova poslovna vrednost. Pretpostavljamo da se unapređenje tehničkog kvaliteta automatski odslikava na poslovnu vrednost.

Više istraživača detaljno je proučavalo odnose između poslovne i tehničke vrednosti. Na primer, Simmonsova je ispitala mnoge australijske preduzetnike da bi odredila kako donose poslovne odluke u vezi sa informacionim tehnologijama. Ona predlaže okvir za razumevanje šta kompanije podrazumevaju pod „poslovnom vrednošću" (Simmons 1996). U izveštaju Favara i Pfleegera (1997), Steve Andriole, izvršni direktor za informatiku velike američke osiguravajuće kompanije Cigna Corporation, opisao je kako njegova kompanija razlikuje tehničku vrednost od poslovne:

> Mi merimo kvalitet (našeg softvera) pomoću očiglednih meta: odnos između vremena operativnosti i vremena zastoja, troškova održavanja, troškova izmena i tome slično. Drugim rečima, razvojem upravljamo na osnovu radnih performansi u okviru parametara troškova. Na koji način isporučilac obezbeđuje isplativu performansu manje je važno u odnosu na rezultate rada ... Pitanje poslova prema tehničkoj vrednosti je blisko i drago našem srcu ... i to je ono na šta usredsređujemo veliki deo pažnje. Mislim da bih se iznenadio kada bih saznao da jedna kompanija sa drugom ugovara tehničku vrednost na račun gubitka poslovne vrednosti. Makar i grešili na drugoj strani! Ako ne postoji jasna (očekivana) poslovna vrednost (kvantitativno izražena: broj obrađenih reklamacija itd.) onda ne pokrećemo projekat. Veoma ozbiljno shvatamo fazu specifikacije zahteva u sklopu projekta kao „utvrđivanje svrhe", kada postavljamo pitanje: „zašto zaista želimo ovaj sistem?" i „zašto nas se to tiče?".

Bilo je više pokušaja za nalaženje veza između tehničke i poslovne vrednosti na kvantitativan i smislen način. Na primer, Humphrey, Snyder i Willis (1991) napominju da je unapređenjem procesa razvoja u skladu sa CMM skalom „zrelosti" (biće razmatrana u poglavlju 12), kompanija Hughes Aircraft poboljšala produktivnost četiri puta i uštedela milione dolara. Slično tome, Dion (1993) izveštava da je Raytheonovo udvostručenje produktivnosti bilo praćeno sa 7,7 dolara zarade na svaki dolar investiran u unapređenje procesa. Osoblje u vazduhoplovnoj bazi Tinker u Oklahomi zabeležilo je poboljšanje produktivnosti od 6,35 puta (Lipke i Butler 1992).

Međutim, Brodman i Johnson (1995) detaljnije su proučili poslovnu vrednost unapređenja postupka. Oni su ispitali 33 kompanije koje su aktivnošću neke vrste unapredili postupak, pri čemu su istražili više ključnih pitanja. Između ostalog, Brodman i Johnson pitali su kompanije kako definišu povraćaj od investicije, koncept koji je jasno definisan u poslovnim krugovima. Oni napominju da školska definicija **povraćaja od investicije**, izvedena iz finansijskog okruženja, opisuje investiciju zavisno od toga kakvu je vrednost ona donela. To jest: „investicija ne sme da vrati samo početni kapital, već najmanje onoliko više koliko bi ta sredstva zaradila na drugom mestu, plus dodatak za rizik" (Putnam i Myers 1992). Obično, poslovno okruženje koristi jedan od tri modela za ocenu povraćaja od investicije: model naknade, računovodstveni model stope povraćaja i diskontni model toka gotovine.

Međutim, Brodman i Johnson (1995) su utvrdili da vlada i privreda SAD tumače povraćaj od investicije na vrlo različite načine, a ta oba tumačenja različita su od standardnog pristupa u poslovnim školama. Vlada tretira povraćaj od investicija u zavosnosti od dolara, posmatrajući i redukujući operativne troškove, predviđajući dolarsku štednju i računajući trošak primene novih tehnologija. Vladine investicije takođe su izražene u dolarima, kao npr. cena uvođenja novih tehnologija ili inicijativa za unapređenje postupka.

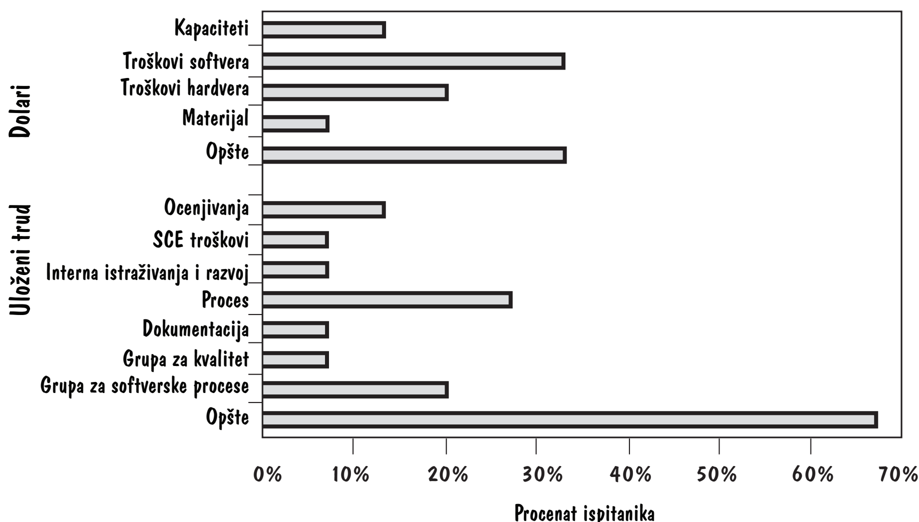
Na drugoj strani, privreda posmatra investicije zavisno od uloženog rada, umesto cene ili pak dolara. To jest, kompanijama je u interesu da uštede vreme ili da koriste manje ljudi, a njihova definicija povraćaja od investicije odražavala je takav njihov interes kao što manji uloženi rad. Među kompanijama koje su ispitivane, povraćaj od investicije obuhvata je elemente kao što su:

- obuka,
- raspored rada,
- rizik,
- kvalitet,
- produktivnost,
- postupak,
- kupac,
- troškovi,
- preduzetništvo.

Pitanje troškova sadržano u definiciji odnosi se na predviđanje troškova, poboljšanje performansi troškova i ostajanje u okviru budžeta, a ne na smanjivanje operativnih troškova ili reorganizaciju projekta ili organizacionog sistema. Slika 1.6 prikazuje učestanost sa kojom razne organizacije uključuju stavku investicija u svoju definiciju pojma povraćaja od investicije. Na primer, oko 5 procenata ispitanika uključilo je rad grupa za kvalitet u izračunavanje povraćaja od investicija, a oko 35 procenata uključilo je softverske troškove kada su razmatrali iznos investiranih dolara.

Razlika u pogledima unosi nespokojstvo, jer to znači da se povraćaji od investicija u raznim organizacijama ne mogu međusobno porediti. Ali postoje dobri razlozi za te različite poglede. Dolarske uštede nastale redukovanjem delatnosti, viši nivo kvaliteta i uvećana produktivnost pre se vraćaju vladi nego izvođaču radova. S druge strane, izvođači radova obično teže konkurentnosti i većim radnim kapacitetima, kao i ostvarenju većeg profita. Stoga je povraćaj od investicija za izvođača radova zasnovaniji na ulože-

nom radu nego na troškovima. Posebno, tačnija procena troškova i rasporeda rada može da rezultuje zadovoljstvom kupca i novim poslovima. Smanjenje vremena za iznošenje proizvoda na tržište i unapređenje kvaliteta proizvoda takođe pružaju poslovnu vrednost. Primećuje se da poslovnu vrednost nude takođe i kraći put do kupca, kao i unapređen kvalitet proizvoda.



SLIKA 1.6 Termini uključeni u definiciju povraćaja od investicija u privredi

Čak i ako se različite kalkulacije povraćaja od investicija mogu opravdati kod različitih organizacija, zabrinjavajuće je da povraćaj od investicija u softverske tehnologije nije isti kao finansijski povraćaj od investicija. U nekom trenutku, uspeh programa se mora saopštiti višim upravljačkim nivoima, od kojih mnogi nemaju dodirnih tačaka sa softverom, već sa osnovnom delatnošću kompanije, kao što su telekomunikacije ili bankarstvo. Mnogo konfuzije potiče od upotrebe iste terminologije za označavanje veoma različitih stvari. Stoga, kriterijum uspešnosti mora da ima smisla ne samo u slučaju softverskih projekata i postupaka, već i u slučaju mnogo opštijih delatnosti, čiju podršku oni predstavljaju. Detaljnije ćemo istražiti ovo pitanje u poglavlju 12, gde ćemo videti nekoliko uobičajenih mera za izbor tehnoloških opcija zasnovanih na poslovnoj vrednosti.

1.4 KO SE BAVI SOFTVERSKIM INŽENJERSTVOM?

Ključna komponenta softverskog razvoja jeste komunikacija između kupca i projektanta. Ako ta komunikacija doživi neuspeh, isto će zadesiti i sistem. Da bismo mogli da napravimo sistem koji treba da pomogne kupcu u rešavanju njegovog problema, moramo da razumemo šta kupac želi i šta mu je potrebno. Da bismo to uradili, usredsredićemo našu pažnju na pojedince uključene u razvoj softvera.

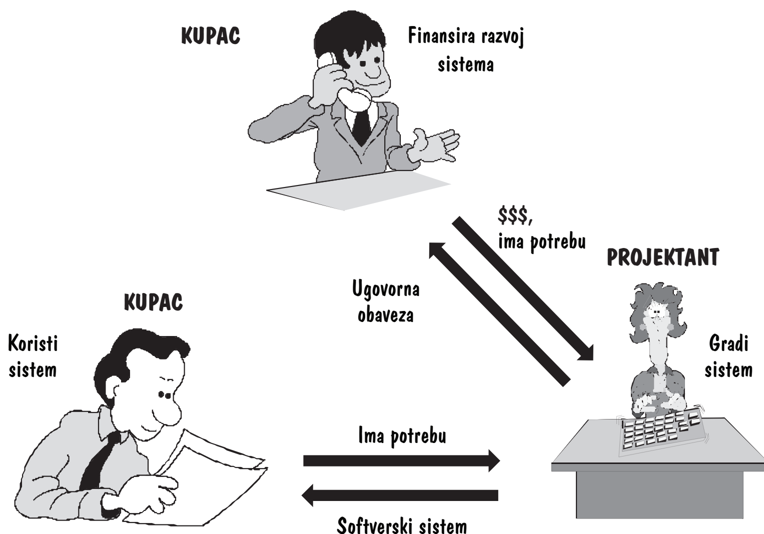
Broj osoba koje rade na razvoju softvera zavisi od veličine projekta i stepena složenosti. Međutim, nije bitno koliko je ljudi uključeno, uloge koje oni igraju tokom trajanja projekta jasno se razlikuju. Stoga, kod velikog projekta, svim pojedincima ili grupama

može biti dodeljena jedna od identifikovanih uloga. Kod malog projekta, jedna osoba ili grupa mogu istovremeno da imaju više uloga.

Obično, učesnici u projektu imaju jednu od tri uloge: kupca, korisnika i projektanta. **Kupac** je kompanija, organizacija ili pojedinac koji plaća za softverski sistem koji se razvija. **Razvoj** obavlja kompanija, organizacija ili pojecinac koji pravi softverski sistem za kupca. Ova kategorija uključuje i rukovodioce potrebne za koordinisanje i vođenje programera i testera. **Korisnik** je jedan ili više pojedinaca koji će stvarno koristiti sistem. To su oni koji sede za terminalom, unose podatke ili čitaju izlazne rezultate. Iako kod nekih projekata ista osoba ili grupa predstavljaju i kupca i korisnika i projektanta, u većini slučajeva se radi o različitim učesnicima. Slika 1.7 prikazuje osnovne odnose između tri tipa učesnika.

Kupac, koji kontroliše novčana sredstva, obično pregovara o ugovoru i potpisuje papire o tehničkom prijemu. Međutim, ponekad kupac nije u isto vreme i korisnik. Na primer, pretpostavimo da kompanija za vodosnabdevanje Wittemberg Water Works potpiše ugovor sa kompanijom Gentle Systems za pravljenje kompjuterizovanog obračunskog sistema. Predsednik Wittenberga može da opiše predstavnicima Gentle Systemsa tačno šta želi i potpisaće ugovor. Međutim, predsednik neće neposredno koristiti obračunski sistem. Korisnici će biti knjigovođe i šalterski radnici. Stoga je važno da razvojni tim razume tačno šta žele kupac i korisnik, odnosno šta im je potrebno.

S druge strane, pretpostavimo da je Wittenberg Water Works toliko velik da ima sopstveni sektor za razvoj računarskih sistema. Taj sektor može da odluči da želi automatski alat pomoću kojeg će pratiti troškove sopstvenih projekata i projektni raspored. Ako taj alat sami prave, taj sektor je istovremeno korisnik, kupac i projektant.



SLIKA 1.7 Učesnici u razvoju softvera

Poslednjih godina, razlika između kupca, korisnika i projektanta postala je mnogo složenija. Kupci i korisnici su uključeni u razvojni proces na više načina. Kupac može da odluči da kupi gotov, katalološki proizvod, koji će biti ugrađen u finalni proizvod koji će

razvojni tim da isporuči i za koji će davati podršku. Kada se to dogodi, kupac je uključen u odluke o arhitekturi sistema, gde postoji mnogo više ograničenja u pogledu razvoja. Slično tome, za razvoj se mogu angažovati razvojni inženjeri ili timovi spolja, koje nazivamo podizvođači, koji grade podsisteme i isporučuju ih razvojnom timu, radi uključenja u finalni proizvod. Podizvođači mogu da rade rame uz rame sa primarnim razvijaocima, isporučujući podsistem kasnije u razvojnom procesu. Podsistem može biti po sistemu „**ključ u ruke**”, gde se programski kôd ugrađuje kao celina (bez dodatnog koda koji treba integrisati), ili se mora posebno integrisati da bi se uspostavile veze između glavnog sistema i podsistema.

Stoga je pojam „sistem” važan u softverskom inženjerstvu, ne samo da bi se razumeli analitički prolemi i sintetisanje rešenja, već i za organizovanje procesa razvoja i dodeljivanje odgovarajućih uloga učesnicima . U sledećem odeljku, videćemo ulogu sistemskog prilaza u pozitivnoj praksi softverskog inženjerstva.

1.5 SISTEMSKI PRILAZ

Projekti koje razvijamo ne egzistiraju u vakuumu. Hardver i softver koje sastavljamo, zajedno su u interakciji sa korisnicima, sa drugim softverima, sa drugim delovima hardvera, postojećim bazama podataka (tj. sa pažljivo definisanim skupovima podataka i odnosima između podataka), ili čak sa drugim računarskim sistemima. Stoga je važno da obezbedimo kontekst projekta koji nazivamo **granice (opseg)** projekta, tj. šta je uključeno u projekat, a šta ne. Na primer, uzmimo da vaš pretpostavljeni od vas zahteva da napišete program za štampanje platnih listića za radnike vašeg preduzeća. Morate da znate pri tome da li vaš program treba da čita radne sate iz drugog sistema i da jednostavno štampa rezultate, ili možda mora takođe da izračunava informacije o plati. Dalje, potrebno je znati da li program treba da izračunava poreze, doprinose za penzije i razne dodatke, odnosno da li i to treba da bude prikazano na svakom platnom listiću. U stvari, postavlja se pitanje gde započinje projekat i gde se završava? Ovo pitanje važi za svaki sistem. Sistem predstavlja skup objekata i aktivnosti, i veza koje međusobno povezuju objekte i aktivnosti. Obično, definicija sistema obuhvata, za svaku aktivnost, spisak obaveznih ulaznih informacija, radnji koje se preduzimaju, kao i izlaznih informacija koje će se proizvesti. Stoga, da bismo uopšte mogli početi, moramo da znamo da li je neki objekat ili aktivnost uključen u sistem ili ne.

Elementi sistema

Opisaćemo sistem imenujući njegove delove i identifikujući kako se sastavni delovi međusobno odnose. Ta identifikacija je prvi korak u analizi problema koji nam je predstavljen.

Aktivnosti i objekti. Najpre, između aktivnosti i objekata postoji razlika. Aktivnost je nešto što se događa u sistemu. Obično se opisuje kao događaj iniciran nekim okidačem; aktivnost transformiše jednu stvar u drugu menjajući joj osobine. Ta transformacija može da znači da se podatak prenosi sa jedne lokacija na drugu, ili se jedna vrednost menja u drugu, ili se kombinuje sa drugim podacima da bi se napravila ulazna informacija za neku drugu aktivnost. Na primer, pojedinačni podatak može da se prenese iz jedne datoteke u drugu. U tom slučaju, karakteristika koja je izmenjena jeste njegova lokacija.

Vrednost podataka može da se uveća. Na kraju, adresa podataka može da se uključi u listu parametara sa adresama više drugih podataka u cilju poziva neke druge rutine koja obrađuje sve navedene podatke odjednom.

Elementi uključeni u aktivnosti zovu se **objekti** ili **entiteti**. Obično, ti objekti su povezani na neki način. Na primer, objekti mogu da budu uređeni u tabelu ili matricu. Često su grupisani u zapise, gde je svaki zapis uređen u propisanom formatu. Na primer, zapis o istoriji rada nekog radnika može da sadrži objekte (koje zovemo polja) za svakog radnika, kao na primer:

Ime	Poštanski broj
Srednje slovo	Plata po času
Prezime	Dodaci po času
Ulica i broj	Obračunati slobodni sati
Grad	Obračunato bolovanje
Država	

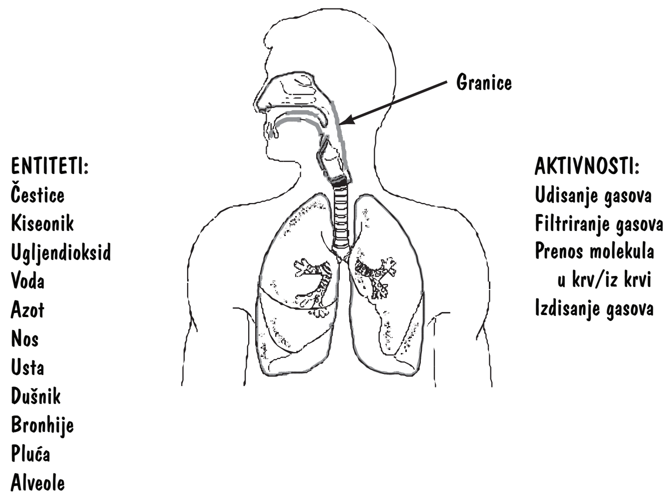
Ne samo da je svako polje u zapisu definisano, već je navedena veličina svakog polja i njihov međusobni odnos. Stoga, opis zapisa u svakom polju mora da se sastoji od tipa podatka, početne pozicije u zapisu i dužine polja. Dalje, kako postoji zapis za svakog radnika, zapisi se kombinuju u datoteku, koja takođe ima svoje devinisane karakteristike (na primer, maksimalni broj zapisa).

Ponekad, objekti su definisani na neznatno različit način. Umesto da posmatramo svaku stavku kao polje u većem zapisu, objekti se posmatraju kao nezavisni. Opis objekta sadrži spisak karakteristika svakog objekta, kao i spisak svih akcija koje mogu da se odigraju upotrebom objekta ili koji utiču na objekat. Na primer, razmotrićemo objekat „poligon". Opis objekta može da kazuje da taj objekat ima karakteristike kao što je broj stranica i njihova dužina. Akcije mogu da uključuju izračunavanje površine ili obima. Može da postoje karakteristika „tip poligona", koja omogućuje indentifikovanje svakog primerka „poligona", kao na primer „romb" ili „četvorougaonik". Opis objekta može biti sadržan u samom tipu. Na primer, „četvorougaonik" može da bude sastavljen od tipova „kvadrat" i „nije kvadrat". Objasnićemo ove koncepte u poglavlju 4, kada budemo istraživali analizu zahteva, a detaljno u poglavlju 6 kada budemo razmatrali objektno orijentisani razvoj.

Odnosi i granice sistema. Jednom kad definišemo entitete i aktivnosti, onda uparujemo entitete sa odgovarajućim aktivnostima. Odnosi između entiteta i aktivnosti jasno su i pažljivo definisani. Definicija entiteta sadrži opis odakle potiče entitet. Neke stavke se nalaze u postojećim datotekama, dok se druge stvaraju tokom same aktivnosti. Odredište entiteta je takođe važno. Neke stavke koristi samo jedna aktivnost, dok su druge namenjene kao ulaz za druge sisteme. To jest, neke stavke iz jednog sistema se koriste od strane aktivnosti izvan opsega sistema koji ispitujemo. To znači da je posmatrani sistem ograničen i ima ograničeni domet. Neke stavke prelaze granice da bi ušle u naš sistem, a druge su proizvodi našeg sistema i putuju napolje da bi ih koristili drugi sistemi.

Koristeći te koncepte, možemo da definišemo **sistem** kao zbirku stvari: skup entiteta, skup aktivnosti, opis odnosa između entiteta i aktivnosti, i definiciju granica sistema. Ova definicija sistema važi ne samo za računarske sisteme već i za sve drugo gde su jedni objekti na neki način u interakciji sa drugim objektima.

Primeri sistema. Da bismo videli definiciju sistema na delu, posmatraćemo respiratorni sistem, tj. delove koji omogućavaju udisanje kiseonika i izbacivanje ugljen-dioksida i vode. Lako možemo da definišemo granice sistema. Ako imenujemo pojedinačni organ tela, možemo da kažemo da li on jeste ili nije deo respiratornog sistema. Molekuli kiseonika i ugljen-dioksida su entiteti ili objekti koji se kreću kroz sistem na način koji je jasno definisan. Možemo takođe da opišemo aktivnosti u sistemu zavisno od interakcija entiteta. Ako je neophodno, možemo da ilustrujemo sistem prikazujući šta ulazi u njega i šta iz njega izlazi. Možemo takođe da obezbedimo tabele koje opisuju sve entitete i aktivnosti u koje su one uključene. Slika 1.8 ilustruje respiratorni sistem. Napomenimo da svaka aktivnost uključuje entitete i može biti definisana entitetima koji predstavljaju ulaz, načinom obrade entiteta i finalnim proizvodom (izlazom).

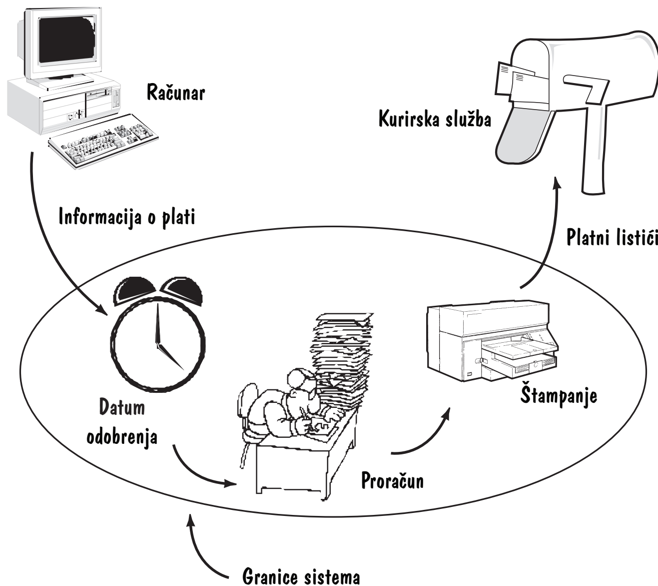


SLIKA 1.8 Respiratorni sistem

Takođe, moramo da jasno opišemo računarski sistem. Sa potencijalnim korisnikom radimo na definisanju granica sistema: Gde naš rad započinje i gde prestaje? Pored toga, potrebno je da znamo čime je ograničen sistem i tako odredimo poreklo ulaza i odredište izlaza. Na primer, u sistemu koji štampa platne listiće, informacija o plati može da dolazi iz računara kompanije. Izlaz sistema može biti skup platnih listića poslat u kurirsku službu da bi se otpremio odgovarajućim primaocima. U sistemu prikazanom na slici 1.9, možemo da vidimo granice i shvatimo entitete, aktivnosti i njihove odnose.

Međusobno povezani sistemi

Koncept razgraničenja je važan jer postoji vrlo malo sistema koji su nezavisni od drugih sistema. Na primer, respiratorni sistem je u interakciji sa probavnim sistemom, sistemom krvotoka, nervnim sistemom i drugima. Respiratorni sistem ne bi mogao da funkcioniše bez nervnog sistema, čak ni krvotok ne bi mogao da funkcioniše bez respiratornog sistema. Međusobne zavisnosti su složene. Činjenica je da su mnogi problemi u okruženju nastali i intenzivirali se jer se nije cenila kompleksnost eko sistema. Ipak, kada su opisane granice sistema lakše je videti šta je unutra a šta van, kao i šta prelazi granicu.



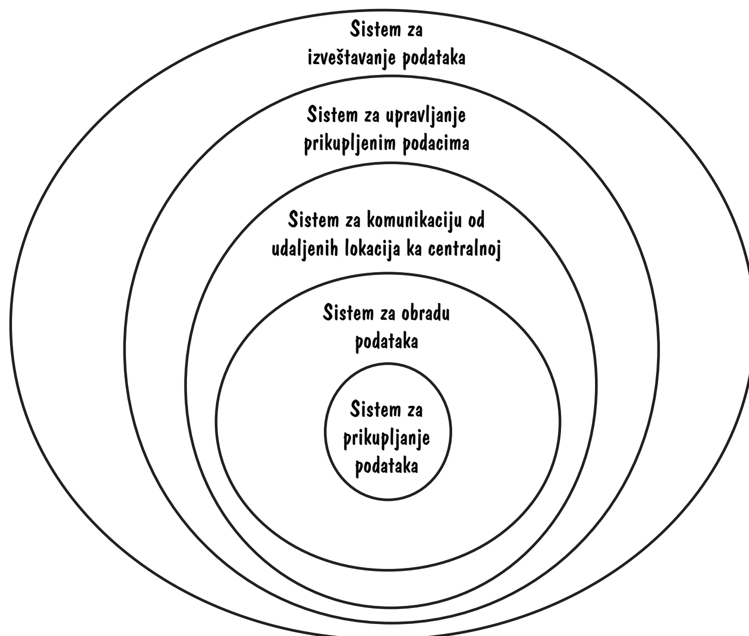
SLIKA 1.9 Definicija sistema za pravljenje platnih listića

Dalje, moguće je da jedan sistem postoji unutar drugog sistema. Kada opisujemo računarski sistem, često se bavimo samo malim delom onoga što je zaista nadređeni sistem. Takvo usredsređivanje omogućava nam da definišemo i pravimo manje kompleksan sistem nego što je onaj koji ga okružuje. Ako pažljivo dokumentujemo interakcije između sistema koji utiče na naš sistem, ne gubimo ništa koncentrišući se na taj manji deo većeg sistema.

Pogledaćemo primer kako to možemo da uradimo. Pretpostavimo da razvijamo sistem za nadgledanje vodotoka gde se podaci sakupljaju na mnogim tačkama duž rečne doline. U sabirnim centrima se radi nekoliko proračuna, a rezultati se šalju na centralnu lokaciju radi sveobuhvatnog izveštavanja. Takav sistem može biti organizovan tako što računar na centralnoj lokaciji komunicira sa više desetina manjih računara na udaljenim lokacijama. Pri tome je potrebno da razmotrimo više sistemskih aktivnosti, uključujući način kako se prikupljaju podaci o vodi, proračuni koji se izvode na udaljenim lokacijama, prenos informacija do centralne lokacije, skladištenje primljenih podataka u bazi podataka ili zajedničkoj datoteci, kao i pravljenje izveštaja od dobijenih podataka. Sistem možemo da posmatramo kao skup sistema, svaki sa posebnom namenom. Posebno, možemo da posmatramo samo komunikacione aspekte velikog sistema i razvijemo komunikacioni sistem za prenos podataka od skupa udaljenih lokacija do centralne lokacije. Ako pažljivo definišemo granicu između komunikacija i velikog sistema, projektovanje i razvoj komunikacionog sistema može se realizovati nezavisno od celokupnog sistema.

Kompleksnost čitavog sistema za nadgledanje vodotoka mnogo je veća nego kompleksnost komunikacionog sistema, tako da tretiranje odvojenih, manjih delova, čini posao mnogo jednostavnijim. Ako je definicija razgraničenja detaljna i ispravna, pravljenje velikog sistema od manjih sistema je relativno lako. Možemo da opišemo proces gradnje posmatrajući veliki sistem kroz slojeve, kao što je prikazano na slici 1.10, koja prikazuje primer nadgledanja vodotoka. Svaki sloj predstavlja sistem, ali svaki sloj sa

svojim podslojevima takođe formira sistem. Kružnice na slici predstavljaju razgraničenja odgovarajućih sistema, a čitav skup kružnica se udružuje u celovit sistem za nadgledanje vodotoka.



SLIKA 1.10 Slojevi sistema za nadgledanje vodotoka

Uočavanje da jedan sistem sadrži drugi sistem je važno, jer odražava činjenicu da je objekat ili aktivnost u jednom sistemu deo svakog sistema predstavljenog spoljašnjim slojevima. Kako svaki sloj unosi dodatnu složenost, razumevanje pojedinačnih objekata ili aktivnosti postaje sve teže. Stoga fokusiranjem u startu na najmanji mogući sistem vršimo maksimalno pojednostavljivanje, čime se poboljšava naše dalje razumevanje sistema.

Tu ideju koristimo kada pravimo sistem kojim menjamo stariju verziju, bilo ručno ili automatski. Želimo što bolje da razumemo kako rade stari i novi sistemi. Često, što je veća razlika između dva sistema, to su projektovanje i razvoj teži. Ova poteškoća nije prouzrokovana samo time što ljudi imaju otpor prema promenama, nego i time što razlike otežavaju učenje. Tokom gradnje ili sinteze našeg velikog sistema, od velike pomoći je pravljenje novog sistema koji predstavlja inkrementalni niz međusistema. Umesto da idemo od sistema A ka sistemu B, biće moguće da idemo od A preko A' i preko A'' do B. Na primer, pretpostavimo da je A ručni sistem koji se sastoji od tri glavne funkcije, pri čemu je B automatizovana verzija sistema A. Možemo da definišemo sistem A' kao novi sistem sa automatizovanom funkcijom 1, pri čemu su funkcije 2 i 3 još uvek ručne. Zatim A'' ima automatizovane funkcije 1 i 2, dok je 3 još uvek ručna. Na kraju, B ima sve

tri automatizovane funkcije. Deleći „rastojanje" od A do B na trećine, imamo niz malih problema sa kojima se lakše možemo izboriti nego sa celinom.

U našem primeru, dva sistema su veoma slična, funkcije su iste, ali se razlikuje stil kako su implementirane. Međutim, ciljni sistem je često veoma različit od postojećeg. Povrh svega, obično je poželjno da ciljni sistem nema ograničenja koja postavljaju postojeći hardver ili softver. Pristup **inkrementalnog razvoja** uključuje niz stanja, gde svako sledeće oslobađa prethodno od još jednog takvog ograničenja. Na primer, stanje 1 dodaje novi hardverski deo, stanje 2 vrši zamenu softvera koji izvodi određeni skup funkcija itd. Sistem se polako udaljava od starog hardvera i softvera, sve dok ne bude odražavao novi dizajn sistema.

Stoga, razvoj sistema može prvo da inkorporira skup izmena nad stvarnim sistemom, a zatim ih dopuni nizom izmena koje generišu celokupnu šemu dizajna, umesto pokušaja da se put od sadašnjeg do budućeg stanja pređe u jednom koraku. Sa takvim pristupom, moramo istovremeno da posmatramo sistem na dva različita načina: statički i dinamički. Statički pogled nam govori kako sistem sada radi dok dinamički pogled prikazuje kako se sistem menja u nešto što će na kraju i da postane. Jedan pogled nije potpun bez drugog pogleda.

1.6 INŽENJERSKI PRISTUP

Jednom kada shvatimo prirodu sistema, spremni smo da započnemo njegovo konstruisanje. Na ovom mestu, „inženjerski" deo softverskog inženjerstva postaje relevantan i komplementaran sa onim šta smo do sada uradili. Podsetimo se da smo počeli ovo poglavlje priznanjem da je pisanje softvera jednako umetnost kao i nauka. Umetnost proizvodnje sistema uključuje zanat proizvodnje softvera. Kao umetnici, razvijamo tehnike i alate za koje je dokazano da pomažu u proizvodnji korisnih, visokokvalitetnih proizvoda. Na primer, možemo da upotrebimo prevodilac sa optimizacijom kao alat za generisanje programa koji se brzo izvršavaju na mašinama koje koristimo. Ili možemo da uključimo specijalne rutine za sortiranje ili pretraživanje, kao tehnike za uštedu vremena ili prostora na sistemu. Ove softverske tehnike koriste se na isti način na koji se tehnike i alati koriste u zanatskom delu izrade lepih delova nameštaja ili tokom gradnje kuće. Zaista, popularna zbirka programerskih alata se zove „programerska tezga" (*Programmer's Workbench*), jer programeri koriste te alate kao što stolar koristi stolarsku tezgu.

Shodno činjenici da je gradnja sistema slična gradnji kuće, u graditeljstvu ćemo potražiti još primere koji će pokazati zašto je „umetnički" pristup u razvoju softvera važan.

Gradnja kuće

Pretpostavimo da su Chuck i Betsy Howell unajmili nekoga da im sagradi kuću. Zbog svoje veličine i kompleksnosti kuća, obično je neophodno više ljudi u građevinskom timu. Prema tome, Howellovi su gradnju ugovorili sa građevinskim preduzećem McMullen. Prvi događaj uključen u gradnju kuće je sastanak između Howellovih i McMullena, tako da Howellovi mogu da objasne šta žele. Taj sastanak istražuje ne samo kako Howellovi žele da kuća izgleda, već i koja svojstva treba da ima. Zatim je preduzeće McMu-

llen iscrtalo tlocrte, kao i arhitektonsko rešenje kuće. Nakon detaljne diskusije između Howellovih i McMullena, napravljene su određene izmene. Kada su Howellovi dali svoj pristanak McMullenu, građevinski radovi su započeli.

Tokom procesa gradnje, Howellovi će verovatno posećivati gradilište, razmišljajući o željenim izmenama. Tokom gradnje može doći da više takvih izmena, ali na kraju kuća će biti dovršena. Tokom gradnje i pre nego što se Howellovi usele, proveravane su razne komponente kuće. Na primer, električari su proveravali strujna kola, vodoinstalateri su se uverili da cevi ne cure, a stolari su sredili drvenariju i podove. Na kraju, Howellovi su se uselili. Ako ima nečega što nije ispravno sagrađeno, oni će zvati preduzeće McMullen da to popravi, ali na kraju Howellovi preuzimaju potpunu odgovornost za kuću.

Pogledaćemo detaljnije šta je sve bilo obuhvaćeno procesom. Prvo, mnogo ljudi radi na kući u isto vreme, pa je neophodna dokumentacija. Ne samo tlocrti i arhitektonski crteži, već i detalji moraju biti zapisani, tako da specijalisti, kao što su vodoinstalateri i električari mogu da montiraju svoje proizvode kako bi kuća postala jedna celina.

Drugo, nerazumno je očekivati da Howellovi opišu svoju kuću na početku procesa i jednostavno ušetaju kada je kuća kompletirana. Umesto toga, Howellovi mogu da modifikuju projekat kuće više puta tokom građevinskih radova. Te modifikacije mogu biti rezultat više situacija:

- materijali koji su bili specifikovani na početku nisu više raspoloživi. Na primer, neke vrste krovnih pokrivača više se ne proizvode.
- Howellovi mogu da dobiju nove ideje kada vide kuću koja dobija oblik. Na primer, oni mogu da shvate da mogu da dodaju svetlarnik na kuhinju za malu dodatnu cenu.
- Raspoloživost ili finansijska ograničenja mogu da zahtevaju od Howellovih da izmene zahteve u cilju zadovoljavanja roka ili budžeta. Na primer, posebni prozori koje su Howellovi hteli da naruče neće biti spremni na vreme da bi kuća bila dovršena do zime, tako da će biti zamenjeni prozorima kojih ima na skladištu.
- Pokazuje se da se ne mogu ispoštovati stavke ili zahtevi dizajna o kojima je na početku postignut dogovor. Na primer, ispitivanje vodopropustljivosti tla možda otkrije da zemljište koje okružuje kuću ne može izdržati onaj broj kupatila koje su Howellovi na početku zahtevali.

Preduzeće McMullen takođe može da preporuči neke izmene po započetim građevinskim radovima, možda zbog boljeg rešenja ili zato što ključni član konstrukcionog tima nije bio raspoloživ pre toga. Takođe, i McMullen i Howellovi mogu zajedno da promene mišljenje o nekom svojstvu kuće, čak i nakon što je to svojstvo ugrađeno.

Treće, McMullen mora da obezbedi šematske planove, dijagrame električnih instalacija i vodovoda, uputstva za kućne uređaje i svu drugu dokumentaciju koje će omogućiti Howellovima da naprave modifikacije ili popravke nakon što se usele.

Sumiraćemo ovaj proces gradnje na sledeći način:

- određivanje i analiziranje zahteva;
- izrada i dokumentovanje celokupnog projekta kuće;
- izrada detaljne specifikacije kuće;

- identifikovanje i određivanje komponenti;
- gradnja svake komponente kuće;
- testiranje svake komponente kuće;
- integrisanje komponenti i pravljenje finalnih izmena nakon što se stanari usele;
- nastavljanje održavanja u ime stanara kuće.

Videli smo kako učesnici moraju da ostanu fleksibilni i omoguće izmene prvobitne specifikacije u različitim trenucima tokom izgradnje.

Važno je zapamtiti da se kuća gradi u određenim socijalnim, ekonomskim i pravnim okvirima. Kao što su kod sistema za nadgledanje vodotoka na slici 1.10 opisane međuzavisnosti podsistema, i o kući moramo da razmišljamo kao o podsistemu u okviru neke veće šeme. Na primer, konstrukcija kuće se radi u kontekstu građevinskih propisa i regulativa grada ili okruga. Radnici preduzeća McMullen imaju dozvole od grada ili okruga, i od njih se očekuje da svoje poslove izvode prema građevinskim standardima. Takođe, gradilište obilazi građevinski inspektor, koji proverava da li se poštuju standardi. Građevinski inspektori postavljaju standarde u pogledu kvaliteta, dok inspekcije služe kao kontrolni mehanizam koji se stara da građevinski projekat zadovolji određeni kvalitet. Mogu da postoje i socijalna ili običajna ograničenja koja sugerišu uobičajeno ili prihvatljivo ponašanje. Na primer, nije običaj da ulazna vrata direktno vode u kuhinju ili spavaću sobu.

U isto vreme, moramo da priznamo i da ne možemo precizno da propišemo aktivnosti građenja kuće. Moramo da ostavimo mesta za odluke zasnovane na iskustvu, za bavljenje neočekivanim ili nestandardnim situacijama. Na primer, mnoge kuće sastavljene su od već postojećih komponenti. Vrata se isporučuju sa gotovim okvirom, za kupatila se koriste gotove kabine za tuširanje itd. Međutim, standardni proces gradnje kuće može biti izmenjen da bi se prihvatilo neko neuobičajeno svojstvo ili zahtev. Pretpostavimo da je kostur kuće postavljen, kao i suvi zidovi i međuspratovi, a sledeći korak je postavljanje pločica na pod kupatila. Zidari sa zgražanjem otkrivaju da zidovi i pod nisu tačno pod pravim uglom. Taj problem možda nije rezultat lošeg procesa. Kuće se grade iz delova koji imaju svoje prirodno ili fabričko odstupanje, tako da može doći do problema u vezi sa nepreciznošću. Pravougaone podne pločice istaći će nepreciznost ako se postavljaju na uobičajeni način. Međutim, to je mesto gde dolaze na red umetnost i iskustvo. Zidar će postavljati pločice jednu po jednu, praveći mala podešavanja kod svake od njih, tako da sveukupna varijacija bude neprimetna za sve osim najpronicljivijih.

Stoga, gradnja kuće je kompleksan zadatak sa mnogim mogućnostima za usputne izmene u procesima, proizvodima ili resursima, ublažen zdravom dozom umetnosti i iskustva. Proces gradnje kuće može biti standardizovan, ali uvek postoji potreba za ekspertskim procenama i kreativnošću.

Gradnja sistema

Softverski projekti se odvijaju na sličan način kao i proces gradnje kuće. U našem primeru, Howellovi su u ovom slučaju bili i kupci i korisnici, a preduzeće McMullen je sprovodilo razvoj. Da su Howellovi tražili od McMullana da sagradi kuću da bi u njoj živeli roditelji gospodina Howella, tada bi korisnici, kupci i graditelji (razvijaoci) bili

različiti. Po analogiji, razvoj softvera uključuje korisnike, kupce i razvojni tim. Ako se od nas traži da razvijemo softverski sistem za kupca, prvi korak je sastajanje sa kupcem radi određivanja zahteva. Ti programski zahtevi, kao što smo ranije videli, opisuju sistem. Bez poznavanja granica, entiteta i aktivnosti, nemoguće je da opišemo softver, kao i to kako će on biti u interakciji sa okruženjem.

Jednom kada definišemo zahteve, pravimo projekat sistema radi zadovoljavanja zadatih zahteva. Kao što ćemo videti u poglavlju 5, projekat sistema prikazuje kako će sistem izgledati iz perspektive kupca. Stoga, kao što su Howellovi gledali tlocrte i arhitektonske planove, ovde treba da kupcu predstavimo snimke ekrana koji će se koristiti, izveštaje koji će se generisati i sve ostalo što će opisati interakciju u kojoj će biti korisnici sa kompletiranim sistemom. Ako sistem ima ručne rezervne procedure ili zamenske procedure, one se takođe opisuju. U početku, Howellovi su bili zainteresovani samo za izgled i funkcionalnost svoje kuće. To je bilo tako sve dok nisu morali da donesu odluke o pojedinostima, na primer da li žele bakarne ili plastične cevi. Slično tome, faza projektovanja softverskog sistema opisuje samo izgled i funkcionalnost.

Zatim kupac pregleda projekat. Kada ga odobri, projekat kompletnog sistema se koristi za generisanje pojedinačnih potprojekata. Napomenimo da se programi pominju tek u ovom koraku. Sve dok se ne odrede funkcionalnost i izgled, najčešće nema smisla da razmatramo kodiranje. U našem primeru kuće, sada bismo bili spremni da govorimo o tipovima cevi ili kvalitetu električne instalacije. Odluku o plastičnim ili bakarnim cevima možemo da donesemo jer sada znamo gde voda treba da teče. Slično tome, kada su svi odobrili projekat sistema, spremni smo da vodimo raspravu o programima. Osnova za našu raspravu jeste dobro definisani opis softverskog projekta kao sistema. Projekat sistema uključuje kompletan opis funkcija i interakcija.

Kada se programi napišu, pre nego što se povežu u celinu oni se testiraju kao individualni delovi koda. Prva faza testiranja zove se testiranje modula ili testiranje jedinica. Kada se uverimo da ti delovi rade kako želimo, sastavljamo ih u celinu, a zatim treba da proverimo da li ti delovi ispravno rade posle povezivanja sa drugim delovima. Druga faza testiranja često se zove integrativno testiranje, jer smo naš sistem gradili dodajući deo na deo, sve dok čitav sistem nije bio spreman za rad. Finalna faza testiranja, koja se zove testiranje sistema, podrazumeva testiranje celokupnog sistema da bismo se uverili da li su pravilno implementirane funkcije i interakcije koje su zadate na početku. U ovoj fazi, sistem se poredi sa navedenim zahtevima. Razvojni tim, kupac i korisnici proveravaju da li sistem služi svojoj nameni.

Na kraju se finalni proizvod isporučuje. Tokom upotrebe, otkrivaju se neslaganja i problemi. Ako se radi o sistemu „ključ u ruke", kupac nakon isporuke preuzima odgovornost za sistem. Mnogi sistemi nisu tipa „ključ u ruke", tako da razvojni tim ili neka druga organizacija obezbeđuju održavanje, ako nešto krene naopako ili se javi potreba da se nešto izmeni.

Stoga, razvoj softvera obuhvata sledeće aktivnosti:

- analizu i definisanje zahteva;
- projektovanje sistema;
- projektovanje programa;

- pisanje programa (implementaciju programa);
- testiranje jedinica;
- integrativno testiranje;
- testiranje sistema;
- isporuku sistema;
- održavanje.

U idealnoj situaciji, aktivnosti se izvode vremenski jedna po jedna. Kada stignemo do kraja spiska, tada je softverski projekat kompletiran. Međutim, u praksi, mnogi koraci se ponavljaju. Na primer, prilikom pregledanja projekta sistema, zajedno sa kupcem možemo da otkrijemo da neki zahtevi još nisu dokumentovani. Možemo da radimo sa kupcem na dodavanju zahteva i mogućoj izmeni projekta sistema. Slično tome, kada pišemo i testiramo kôd, možemo da otkrijemo da uređaj ne funkcioniše kao što je opisano u dokumentaciji. Možda će biti potrebno preprojektovanje koda, ponovno razmatranje projekta sistema, ili čak ponovo razgovarati sa kupcem o tome kako zadovoljiti konkretan zahtev. Zbog svega toga, **proces razvoja softvera** definišemo kao svaki opis razvoja softvera koji sadrži neku od prethodno nabrojanih devet aktivnosti, organizovane tako da zajedno proizvode proveren kôd. U poglavlju 2, istražićemo nekoliko različitih razvojnih procesa koji se koriste za izradu softvera. Sledeća poglavlja istražiće svaki potproces i njihove aktivnosti, od analize zahteva do održavanja. Pre nego što to uradimo, pogledajmo ko razvija softver i kako se izazov razvoja softvera menjao tokom godina.

1.7 ČLANOVI RAZVOJNOG TIMA

U ovom poglavlju smo već videli da kupac, korisnik i razvojni tim igraju glavne uloge u definisanju i stvaranju novog proizvoda. Razvojni tim se sastoji od softverskih inženjera, ali svaki inženjer može da se specijalizuje za poseban aspekt razvojnog procesa. Videćemo detaljnije uloge članova razvojnog tima.

Prvi korak u svakom razvojnom procesu je utvrđivanje šta korisnik želi i dokumentovanje tih zahteva. Kao što smo videli, analiza je postupak razbijanja stvari na sastavne delove, tako da možemo bolje da ih shvatimo. Stoga, razvojni tim uključuje jednog ili više *analitičara zahteva* u rad sa kupcem, da bi se ono što kupac hoće razložilo u pojedinačne zahteve.

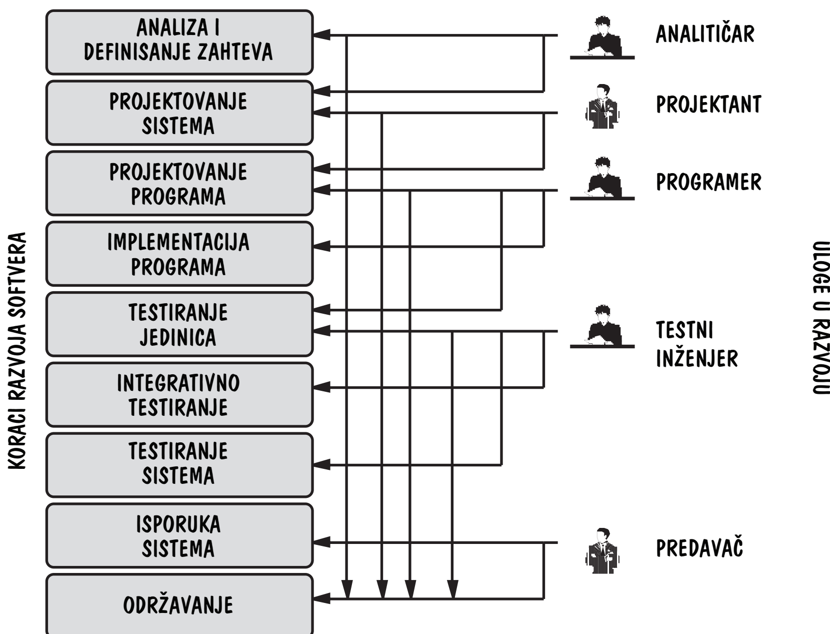
Kada zahtevi postanu poznati i dokumentovani, analitičar radi sa *projektantima* na generisanju opisa funkcija sistema. Dalje, projektanti rade sa programerima na predstavljanju sistema na takav način da *programeri* mogu da pišu kôd koji implementira ono šta je formulisano u zahtevima.

Posle generisanja programskog koda, on mora da se testira. Prvo testiranje često obavljaju sami programeri. Ponekad se koriste drugi *testeri* kao pomoć u otkrivanju grešaka koje previde sami programeri. Kada se jedinice koda integrišu u funkcionalne grupe, timovi za testiranje rade zajedno sa implementacionim timom na verifikovanju ispravnosti rada u skladu sa specifikacijom.

Kada je razvojni tim zadovoljan sa funkcionalnošću i kvalitetom sistema, pažnja se okreće *kupcu*. Tim za testiranje i kupac rade zajedno na verifikovanju celokupnog sistema, poređenjem stvarnog funkcionisanja sistema sa specifikacijom zahteva. Nakon toga, *instruktori* obučavaju korisnike za operativno korišćenje sistema.

U slučaju mnogih softverskih sistema, tehnički prijem od strane kupca ne znači i kraj razvoja. Ako se otkriju greške posle prijema sistema, njih ispravlja *tim za održavanje*. Pored toga, potrebe kupca se mogu menjati tokom vremena, pri čemu na sistemu moraju da se urade odgovarajuće izmene. Stoga, u održavanju mogu učestvovati analitičari koji određuju koje su nove ili izmenjene potrebe, projektanti koji određuju šta u projektu sistema treba izmeniti, programeri za implementiranje izmena, inženjeri za testiranje radi provere da li izmenjeni sistem još uvek ispravno radi, kao i instruktori koji objašnjavaju korisnicima kako se izmena odražava na upotrebu sistema. Slika 1.11 ilustruje u kojim koracima razvoja učestvuju koji pripadnici razvojnog tima.

Studenti često rade samostalno ili u malim grupama, kao razvojni tim za školski projekat. Dokumentacija koju zahteva predavač je minimalna. Od studenata se obično ne traži da pišu korisnička uputstva ili dokumente za obuku. Pored toga, zadaci su prilično nepromenljivi jer se zahtevi ne menjaju tokom životnog veka projekta. Na kraju, sistem koji je napravio student verovatno će biti obrisan na kraju godine. Njegova uloga bila je da demonstrira problem koji se javlja kod stvarnog kupca. Stoga, veličina programa, kompleksnost sistema, potreba za dokumentacijom i potreba za održavanjem praktično ne postoje u školskim projektima.



SLIKA 1.11 Uloge razvojnog tima

Međutim, u slučaju stvarnog korisnika, dimenzije sistema i kompleksnost mogu biti veliki, kao i potreba za dokumentacijom i održavanjem. Kod projekta koji ima više hiljada redova koda i mnogo interakcije između članova razvojnog tima, prilično je teško kontrolisati različite aspekte projekta. Više različitih osoba može biti uključeno u sistem na početku i tokom razvoja, radi podrške svim učesnicima u razvojnom timu.

Bibliotekari pripremaju i skladište dokumentaciju koja se koristi tokom života sistema, uključujući specifikacije zahteva, projektne opise, programsku dokumentaciju, uputstva za obuku, podatke sa testiranja, rokove itd. Sa bibliotekarima rade članovi *tima za upravljanje konfiguracijom*. Zadaci upravljanja konfiguracijom su održavanje korespondencije između zahteva, projekta, implementacije i testova. Unakrsne reference kazuju razvojnom timu koji program treba da se modifikuje ako se vrši izmena u zahtevima, kao i na koje delove programa se odražavaju neke predložene izmene. Osoblje koje upravlja konfiguracijama takođe koordiniše različite verzije sistema koji treba sagraditi i podržavati. Na primer, softverski sistem može biti implementiran na različitim platformama ili se isporučuje u nizu verzija. Upravljanje konfiguracijom obezbeđuje konsistentnu funkcionalnost pri prelasku sa jedne platforme na drugu, kao i očuvanje stepena funkcionalnosti sa novom verzijom.

Razvojne uloge mogu da podrazumevaju jednu osobu ili više njih. Kod malih projekata, dva ili tri čoveka mogu da podele sve uloge. Međutim, kod većih projekata, razvojni tim se često deli na jasno odvojene grupe zasnovane na njihovim funkcijama u razvoju. Ponekad, oni koji održavaju sistem nisu isti oni koji su inicijalno projektovali i napisali sistem. Kod velikog razvojnog projekta, kupac može čak da unajmi jednu kompaniju da uradi inicijalni razvoj, a drugu da vrši održavanje. Kada budemo razmatrali aktivnosti razvoja i održavanja u kasnijim poglavljima, videćemo koje su sposobnosti potrebne za svaku vrstu razvojne uloge.

1.8 KAKO SE IZMENILO SOFTVERSKO INŽENJERSTVO

Uporedili smo pravljenje softvera sa gradnjom kuće. Svake godine, širom zemlje se sagrade stotine kuća u koje se useljavaju zadovoljni kupci. Svake godine, razvojni timovi prave stotine softverskih projekata, ali su kupci vrlo često nezadovoljni rezultatom. Gde je tu razlika? Kad je tako lako nabrojati korake u razvoju sistema, zašto onda softverski inženjeri imaju problema prilikom pravljenja kvalitetnog softvera?

Prisetimo se našeg primera gradnje kuće. Tokom procesa gradnje, Howellovi su neprestano pregledali planove. Oni su takođe imali mnogo prilika da promene mišljenje o onome što su želeli. Na isti način, razvoj softvera omogućava kupcu da pregleda planove na svakom koraku i da napravi izmene u projektu. Nakon svega, ako razvojni tim proizvede odličan proizvod koji ne zadovoljava korisnikove potrebe, svi su uzalud utrošili i vreme i trud na rezultujući sistem.

Stoga je od izuzetnog je značaja da se alati i tehnike za softversko inženjerstvo koriste prilično fleksibilno. U prošlosti, tokom razvoja pretpostavili smo da kupac zna od samog početka šta želi. Ta nepromenljivost nije uobičajena. Kako se dostižu razna stanja projekta, ograničenja se pojavljuju tamo gde na početku nisu bila očekivana. Na primer, pošto smo izabrali hardver i softver za neki projekat, možda shvatimo da su promenjeni zahtevi onemogućili da se naprave meniji na osnovu određenog sistema za upravljanje

bazom podataka tačno onako kako je obećano korisniku. Ili možda otkrijemo da je drugi sistem, sa kojim naš treba da ima vezu, promenio proceduru ili format očekivanih podataka. Može se čak dogoditi da hardver i softver ne rade baš onako kako je obećano u dokumentaciji proizvođača. Zato ne treba zaboraviti da je svaki projekat priča za sebe i da alati i tehnike moraju biti odabrani tako da odražavaju ograničenja koja postavlja konkretan projekat.

Takođe, treba imati u vidu da većina sistema ne radi samostalno već u vezi sa drugim sistemima, bilo primajući od njih informacije, ili ih za njih obezbeđujući. Razvoj takvih sistema je kompleksan jednostavno zato što zahteva veliku koordinaciju između sistema koji međusobno komuniciraju. Ta kompleksnost se posebno iskazuje kod sistema koji se paralelno razvijaju. U prošlosti, razvoj nije lako obezbeđivao tačnost i zaokruženost dokumentacije za međusistemski interfejs. U sledećim poglavljima bavićemo se pitanjem kontrolisanja interfejsa.

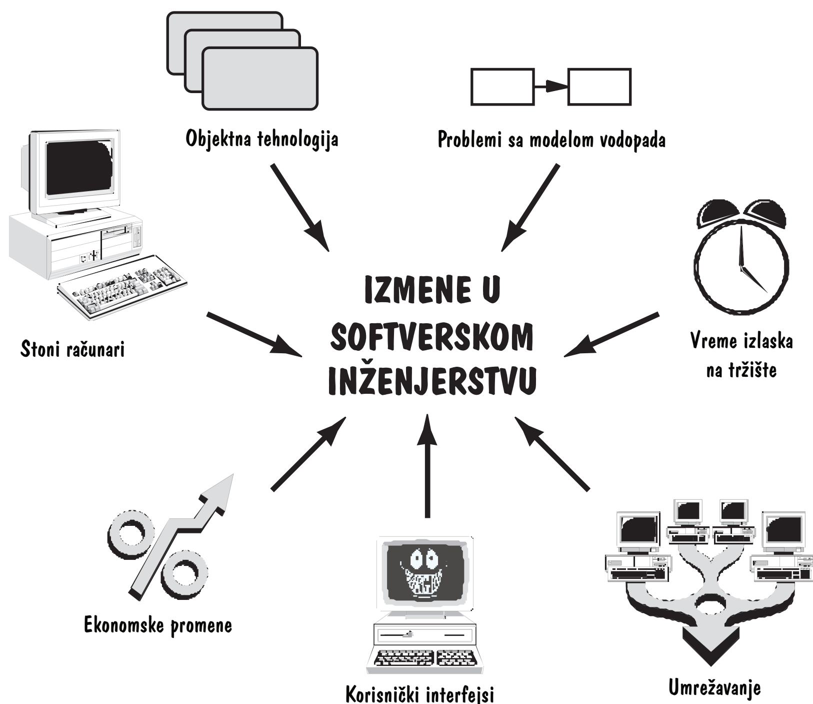
Priroda izmene

Ovi problemi su jedni od mnogih koji utiču na uspeh razvoja softverskog projekta. Koji god pristup da odaberemo, moramo da gledamo i unazad i unapred. Moramo da gledamo unazad, na ranije razvijene projekte, da bismo videli šta smo naučili, ne samo o obezbeđivanju kvaliteta softvera, već i o efikasnosti tehnika i alata. Unapred gledamo da bismo anticipirali verovatni razvoj i korišćenje softverskih proizvoda u budućnosti, što će izmeniti naš način rada. Wasserman (1995) ukazuje na to da su izmene od 1970-ih postale dramatične. Na primer, prve aplikacije su razvijane sa ciljem da se izvršavaju na jednoprocesorskom, obično velikom, računaru. Ulaz je bio linijski, obično preko kartica ili trake, dok je izlaz bio alfanumerički. Sistem je bio projektovan na jedan od dva osnovna načina: kao **transformacija**, gde je ulaz konvertovan u izlaz, ili kao **transakcija**, gde je ulaz određivao koja će funkcija biti izvršena. Današnji sistemi zasnovani su na softveru, veoma su raznovrsni i znatno složeniji. Po pravilu se izvršavaju na više sistema, ponekad su konfigurisani u klijent-server arhitekturi, sa distribuiranom funkcionalnošću. Softver ne obavlja samo primarne funkcije namenjene korisniku, već upravlja mrežom, zaštitom, prezentacijom i obradom korisničkog interfejsa, kao i podacima ili objektima. Tradicionalni pristup razvoju po modelu „vodopada" (naivni pristup, gde razvoj liči na mirnu plovidbu od tačke do tačke), pretpostavlja linearnu progresiju razvojnih aktivnosti, gde jedna počinje samo kada se prethodna završi (proučićemo u poglavlju 2), nefleksibilan je i nije pogodan za današnje sisteme.

Wasserman (1996) je rezimirao te izmene identifikujući sedam ključnih faktora koji su izmenili praksu softverskog inženjerstva, što je ilustrovano na slici 1.12:

1. vitalni značaj vremena izlaska na tržište za komercijalne proizvode;
2. ekonomske promene u računarskoj industriji: niže cene hardvera a viši troškovi razvoja i održavanja;
3. raspoloživost moćnih stonih računara;
4. intenzivna upotreba lokalnih i rasprostranjenih mreža;
5. raspoloživost i prihvatanje objektno orijentisane tehnologije;

6. grafički korisnički interfejs oslonjen na prozore, ikone, menije i pokazivače;
7. nepredvidivost modela vodopada,



SLIKA 1.12 Ključni faktori koji su izmenili razvoj softvera

Na primer, tržišni pritisak podrazumeva da preduzetnici moraju plasirati nove proizvode i usluge pre nego što to uradi konkurencija. U protivnom, sudbina poslovanja može biti dovedena u pitanje. Stoga tradicionalne tehnike za pregledanje i testiranje nisu korisne ako zahtevaju veliki utrošak vremena, a su zastoji zbog nedostatka i otkaza retki. Slično tome, vreme koje je ranije trošeno na optimizaciju koda u cilju poboljšanja brzine ili smanjenja zauzeća prostora možda više nije pametna investicija, jer dodatni disk ili memorijski modul mogu da bud značajno jeftinije rešenje problema.

Pored toga, stoni računari snagu razvoja stavljaju u ruke korisnicima, koji svoj sistem koriste za razvoj aplikacija zasnovanih na radnim tabelama i bazama podataka, malih programa pa čak i specijalizovanih korisničkih interfejsa i simulacija. Ovo premeštanje odgovornosti za razvoj znači da ćemo, kao softverski inženjeri, verovatno morati graditi još kompleksnije sisteme nego pre. Slično tome, ogromne mogućnosti umrežavanja raspoložive većini korisnika i učesnika u razvoju, pojednostavljuju pronalaženje željenih informacija i bez posebnih aplikacija. Na primer, pretraživanje World Wide Weba je brzo, lako i efikasno. Korisnik više nema potrebe da piše aplikacije za bazu podataka da bi pronašao ono što mu je potrebno.

Takođe je unapređen i posao projektanata. Objektno orijentisana tehnologija, povezana sa mrežama i repozitorijumima višekratno upotrebljivih komponenti, stavlja pro-

jektantima na raspolaganje mnoštvo takvih modula, koji se trenutno ili vrlo brzo mogu uključiti u nove aplikacije. Takođe i grafički korisnički interfejsi, koji se često razvijaju pomoću posebnih alata, pomažu da komplikovane aplikacije dobiju poznato lice. Zahvaljujući tome što smo postali sofisticirani u načinu na koji analiziramo probleme, sada možemo da podelimo sistem tako da paralelno razvijamo podsisteme, što zahteva model procesa razvoja koji je potpuno drugačiji od modela vodopada. Videćemo u poglavlju 2 da nam je na rasplaganju mnogo opcija u okviru takvog procesa, uključujući one koje omogućavaju izradu prototipova (kako bismo proverili da li su zahtevi korektni i ocenili da li je projekat isplativ) i aktivnosti koje primenjujemo korak po korak. Ti koraci nam pomažu da osiguramo da u zahtevima i projektantskim rešenjima bude što manje nedostataka pre njihove transformacije u programski kôd.

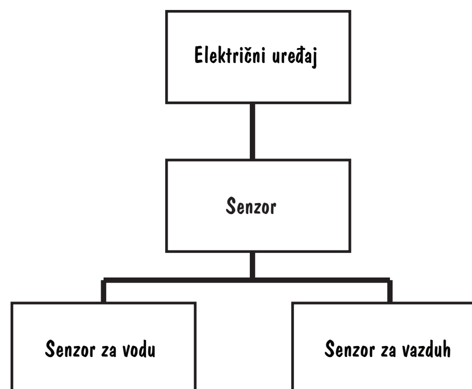
Wassermanova disciplina softverskog inženjeringa

Wasserman (1996) ukazuje na to da svaka od sedam tehnoloških izmena značajno utiče na proces razvoja softvera. Međutim, svih sedam u svojoj ukupnosti promenile su naš način rada. U svojim prezentacijama, DeMarco opisuje taj radikalni pomak tvrdnjom da smo do sada rešili lake probleme, što znači da je one koji su preostali mnogo teže rešiti sada, nego što je to bilo pre. U odgovoru na taj izazov, Wasserman predlaže osam fundamentalnih pojmova u softverskom inženjerstvu koji istinski utemeljuju disciplinu softverskog inženjerstva. Sa njima ćemo se ovde upoznati samo u kratkim crtama, a vratićemo im se u kasnijim poglavljima, da bismo ukazali gde i kako oni mogu da se praktično primene.

Apstrakcija. Ponekad je razmatranje problema u njegovom „prirodnom stanju” (tj. kako ga izražava kupac ili korisnik) prilično obeshrabrujući zadatak. U tom trenutku ne vidimo jasan način da se efikasno ili uopšte uhvatimo u koštac sa problemom. **Apstrakcija** predstavlja opis problema na nekom nivou opštosti koji omogućava usredsređivanje na njegove ključne aspekte, bez bavljenja detaljima. Taj pojam je različit od **transformacije**, gde problem prevodimo u drugo okruženje koje bolje razumemo. Transformacija se često koristi za prevođenje problema iz realnog sveta u matematički svet, tako da do rešenja problema dolazimo proračunima.

Apstrakcija po pravilu obuhvata identifikovanje klasa objekata, koje nam omogućavaju grupisanje pojmova sa ciljem da se usredsredimo na zajedničke osobine klase i time smanjimo broj razmatranih elemenata. Možemo da govorimo o osobinama ili atributima članova klase i da ispitamo odnose između osobina i klasa. Na primer, pretpostavimo da se od nas traži da napravimo sistem za nadgledanje prirodne sredine neke velike i složene reke. Opremu za nadgledanje čine senzori za kvalitet vazduha i zakvalitet vode, za merenje temperature, brzine i drugih karakteristika životne sredine. Ali, za naše potrebe, definisaćemo klasu „senzor”, kod koje svaki član klase ima određene osobine, nezavisne od parametra koji nadzire: visinu, težinu, zahteve napajanja, plan održavanja i sl. Bavimo se klasama, umesto elementima, učeći o kontekstu problema, smišljajući rešenje. Na taj način, klase nam pomažu da pojednostavimo problem i pažnju usmerimo na njegove suštinske elemente ili osobine.

Moguće je uspostaviti i hijerarhiju apstrakcija. Na primer, senzor je tip električnog uređaja, pri čemu imamo dve vrste senzora: senzore za vodu i senzore za vazduh.



SLIKA 1.13 Jednostavna hijerarhija opreme za nadgledanje

Na osnovu toga možemo formirati jednostavnu hijerarhiju prikazanu na slici 1.13. Skrivajući neke detalje, možemo da se usredsredimo na suštinsku prirodu objekata kojima moramo da se bavimo, i na taj način dođemo do jednostavnih i elegantnih rešenja. U poglavljima 5, 6 i 7 detaljnije ćemo razmotriti apstrakciju i skrivanje informacija.

Metodi i notacije za analizu i dizajn. Kada projektujemo program koji predstavlja školski zadatak, obično radimo samostalno. Dokumentacija koja se pri tome generiše predstavlja formalni opis vaših zabeleški, koje su namenjene samo vama, o tome zašto ste izabrali određeni pristup, šta predstavljaju imena promenljivih i koji su algoritmi implementirani. Međutim timski rad podrazumeva komunikaciju sa drugim učesnicima u procesu razvoja. Mnogi inženjeri, bez obzira na to kojom vrstom inženjerstva se bave, koriste standardnu notaciju koja im pomaže u komunikaciji i dokumentovanju donetih odluka. Na primer, arhitekta crta nacrte i planove koje onda drugi arhitekti mogu da razumeju. Još važnije je to da zajednička notacija omogućava građevinskim izvođačima da shvate ideje i namere arhitekte. Kao što ćemo videti u poglavljima 4, 5, 6 i 7, u softverskom inženjerstvu ne postoji sličan standard, a pogrešno tumačenje predstavlja jedan od ključnih problema današnjeg softverskog inženjerstva.

Metode analize i dizajna su više od medijuma za komunikaciju. Omogućavaju nam izgradnju modela i proveru njihove sveobuhvatnosti i konsistentnosti. Uz njih mnogo lakše možemo ponovno koristiti zahteve ili komponente iz ranijih projekata i time relativno lako povećati produktivnost i kvalitet.

Međutim, ima mnogo otvorenih pitanja koja treba razrešiti, pre nego što pređemo na zajednički skup metoda i alata. Kao što ćemo videti u kasnijim poglavljima, različiti alati i tehnike odnose se na različite aspekte problema pa je neophodno identifikovati osnovne elemente modelovanja, koji će nam omogućiti da primenom jedne tehnike obuhvatimo sve važne aspekte problema. U suprotnom, moramo razviti tehniku predstavljanja koja se, uz moguća prilagođavanja, može koristiti u svim metodima.

Pravljenje prototipa korisničkog interfejsa. **Izrada prototipa** podrazumeva realizaciju umanjene verzije sistema, obično sa ograničenom funkcionalnošću, u cilju da:

- korisniku ili kupcu pomognu u identifikaciji ključnih zahteva;
- demonstriraju izvodljivost projekta ili pristupa.

Postupak izrade prototipova je često iterativan. Realizujemo prototip, ocenjujemo ga (pomoću povratne informacije od korisnika i kupca), razmatramo kako izmene mogu da unaprede proizvod ili način projektovanja, a zatim pravimo sledeći prototip. Te iteracije završavaju se kada kupac zaključi da imamo zadovoljavajuće rešenje za problem koji je pred nama.

Izrada prototipa često se koristi za projektovanje dobrog **korisničkog interfejsa**, tj. dela sistema sa kojim korisnik ima interakciju. Međutim, postoje druge mogućnosti za upotrebu prototipova, čak i u **ugrađenim sistemima**, tj. sistemima u kojima softverske funkcije nisu neposredno vidljive korisnicima. Prototip može da prikaže korisniku koje će funkcije biti raspoložive, bez obzira na to da li su implementirane u softveru ili u hardveru. Kako je korisnički interfejs na neki način spona između domena primene i softverskog razvojnog tima, izrada prototipa može da na površinu izbaci pitanja i pretpostavke koje možda nisu bile jasne primenom drugih pristupa u analizi zahteva. Uloga izrade prototipa korisničkog interfejsa biće razmotrena u poglavljima 4 i 5.

Arhitektura softvera. Sveobuhvatna arhitektura sistema je važna ne samo za njegovo lakše implementiranje i testiranje, već i za brzinu i efikasnost održavanja i unošenja izmena. Kvalitet arhitekture može ili da stvori ili da uništi sistem. Shaw i Garlan (1996) su predstavili arhitekturu kao samostalnu disciplinu, čiji se efekti osećaju kroz celokupan proces razvoja. Arhitektonska struktura sistema treba da odražava principe dobrog projektovanja, koje ćemo proučiti u poglavljima 5 i 7.

Arhitektura sistema opisuje sistem oslanjajući se na kategorije koje pripadaju nekom skupu arhitektonskih celina i način na koji se one međusobno povezuju. Što su te celine nezavisnije, to je arhitektura modularnija, pa je nezavisno projektovanje i realizacija tih delova jednostavnija. Wasserman (1996) ukazuje na to da postoji najmanje pet načina kako možemo neki sistem dekomponovati na manje celine:

1. modularna dekompozicija: zasnovana na funkcionalnoj modularizaciji;
2. dekompozicija zasnovana na podacima: zasniva se na spoljašnjim strukturama podataka;
3. dekompozicija zasnovana na događajima: zasniva se na događajima koje sistem mora da prihvati i obradi;
4. dizajn spolja ka unutra: zasnovan na elementima koje korisnik spolja unosi u sistem;
5. objektno orijentisani dizajn: zasnovan na identifikovanju klasa objekata i njihovih veza.

Ovi pristupi se uzajamno ne isključuju. Na primer, možemo da projektujemo korisnički interfejs sa dekompozicijom orijentisan na događaje, dok za projektovanje baze podataka možemo koristiti objektno orijentisani pristup ili pristup koji pokreću podaci. Istražićemo detaljnije te tehnike u sledećim poglavljima. Značaj ovih pristupa je u tome što sumraju naša projektantska iskustva, omogućavajući nam da profitiramo ponovo upotrebljavajući ono što smo ranije uradili, kao i ono što smo tokom tog rada naučili.

Softverski proces. Do kasnih 1980-ih, mnogi softverski inženjeri polagali su posebnu pažnju na *proces* razvoja softvera, kao i na proizvode koji su rezultat tog procesa. Znalo

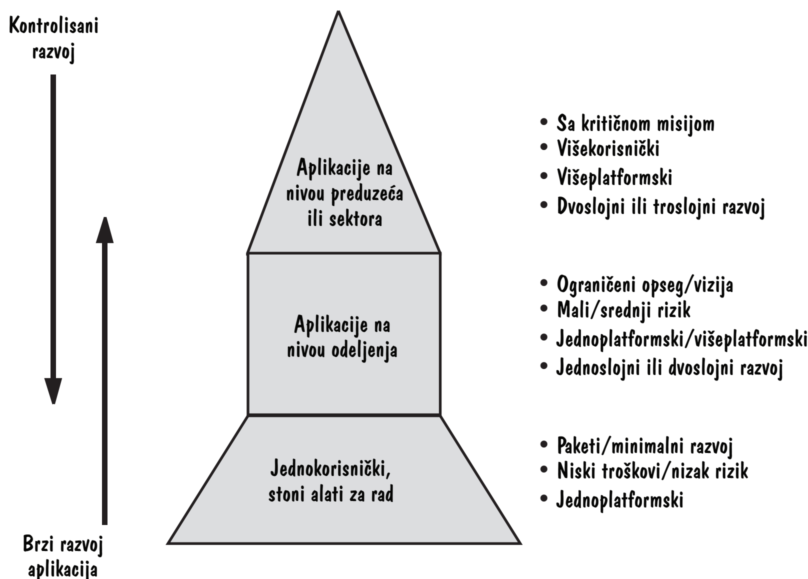
se da organizacija i disciplina u tim aktivnostima doprinose kvalitetu softvera i brzini kojom se softver razvija. Međutim, Wasserman napominje da

> velika raznolikost tipova aplikacija i kultura organizacionih sistema čine nemogućim striktno propisivanje procesa. Stoga se čini da softverski proces za softversko inženjerstvo nema jednako fundamentalan značaj kao što ga imaju apstrakcija i modularizacija (Wasserman 1996).

Umesto toga, on sugeriše da različite vrste softvera zahtevaju različite procese. Wasserman posebno sugeriše da aplikacije koje se primenjuju na nivou celog preduzeća zahtevaju visok stepen kontrole, dok nezavisne aplikacije i aplikacije za potrebe pojedinih odeljenja mogu koristiti prednosti brzog razvoja, kao što je prikazano na slici 1.14.

Koristeći savremene alate, mnoge male i srednje sisteme mogu izgraditi jedan ili dva projektanta, pri čemu svaki od njih mora da preuzme više uloga. U njihove alate spadaju editor teksta, programersko okruženje, podršku za testiranje, možda i mala baza za skladištenje ključnih podataka o proizvodima i procesima. Kako je stepen rizika takvih projekata relativno mali, potrebna je mala podrška u upravljanju i nadzoru.

Međutim, veliki, kompleksni sistemi zahtevaju više uređivanja, provera i usaglašavanja. Takvi sistemi često uključuju veliki broj kupaca i korisnika, a njihov razvoj je dugotrajan. Pored toga, projektanti nemaju uvek potpunu kontrolu nad procesom razvoja, jer neke važne podsisteme mogu isporučiti drugi, ili oni mogu biti implementirani u hardver. Takav tip visoko rizičnog sistema zahteva alate za analizu i projektovanje, upravljanje projektom, upravljanje konfiguracijom, sofisticirane alate za testiranje, kao i još rigorozniji sistem nadzora i analize uzroka. U poglavlju 2 pažljivo ćemo razmotriti nekoliko alternativnih procesa i videti u kakvom su odnosu promene u procesu i različiti ciljevi. Zatim, u poglavljima 12 i 13, procenićemo efikasnost nekih procesa, kao i načine njihovog unapređenja.



SLIKA 1.14 Razlike u razvoju (Wasserman 1996)

Višekratna upotreba. U razvoju i održavanju softvera, često koristimo prednost zajedničkih elemenata u sklopu aplikacija iskorišćavajući ranije razvijene elemente. Na primer, u više razvojnih projekata koristimo isti operativni sistem ili sistem za upravljanje bazama podataka, umesto da svaki put pravimo novi. Kada gradimo sisteme koji su slični, ali ne i identični onima koje smo ranije realizovali, tada možemo iskoristiti postojeće programske zahteve, delove projekta, grupe skriptova za testiranje ili test podataka. Barnes i Boolinger (1991) ukazuju na to da višekratna upotrebljivost nije nova ideja, pri čemu obezbeđuju mnogo zanimljivih primera kako višekratno koristimo ne samo programski kôd, već i mnogo više.

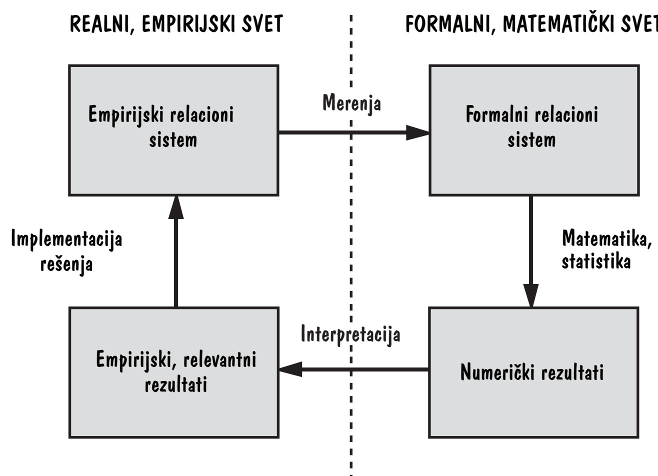
Prieto-Diaz (1991) je uveo pojam višekratno upotrebljivih komponenti kao prednost u poslovanju. Kompanije i organizacije ulažu u elemente koji su višekratno upotrebljivi, a zatim dobijaju dokazanu dobit, u trenucima kada se te stavke iznova koriste u sledećim projektima. Međutim, uspostavljanje dugoročnog i efikasnog programa višekratne upotrebe nije uvek lako, jer postoji nekoliko prepreka:

- Nekada je brže izgraditi malu komponentu, nego tražiti postojeću u skladištu višekratnih komponenti.
- Potrebno je dodatno vreme da bi se komponenta načinila dovoljno opštom da bi bila višekratno upotrebljiva.
- Teško je dokumentovati postignut kvalitet i obavljena testiranja, u takvoj meri da potencijalni korisnik potojeće komponente stekne poverenje u njen kvalitet.
- Nije jasno ko je odgovoran ako komponenta ne radi dobro ili ako je treba prepraviti.
- Potrebno je puno vremena da bi se shvatilo kako radi komponenta koju je neko drugi napisao, što može biti skupo.
- Često postoji sukob između opštih i pojedinačnih osobina.

U poglavlju 12 ponovo ćemo se pozabaviti višekratnom upotrebljivošću i proučiti nekoliko primera uspešnih višekratno upotrebljivih komponenti.

Merenje. Usavršavanje predstavlja pokretačku snagu istraživanja u softverskom inženjerstvu: poboljšanje procesa, korišćenja resursa i metoda u cilju izrade i održavanja boljih proizvoda. Međutim, ponekad se ciljevi usavršavanja izražavaju uopšteno, bez kvantitativne predstave o tome gde smo i kuda idemo. Iz tog razloga, merenje je u softveru postalo ključni element pravilnog rada u softverskom inženjerstvu. Kvantifikujući kad god možemo i šta god možemo, opisujemo akcije i njihove ishode pomoću uobičajenog matematičkog jezika koji nam omogućava da vrednujemo napredak. Pored toga, kvantitativni pristup omogućava poređenje napretka između potpuno različitih projekata. Na primer, dok je John Young bio rukovodilac u Hewlett-Packardu, on je postavio cilj od „10X", tj. desetostruko poboljšanje kvaliteta i produktivnosti svakog projekta u Hewlett-Packardu, bez obzira na tip aplikacije ili oblast primene (Grady i Caswell 1987).

Na nižem nivou apstrakcije, merenja omogućavaju da se vide specifične karakteristike procesa i proizvoda. Često je korisno da naše razumevanje realnog, empirijskog sveta transformišemo u elemente i odnose u formalnom, matematičkom svetu, gde sa njima možemo da manipulišemo radi daljeg razumevanja. Kao što je prikazano na slici 1.15, matematiku i statistiku koristimo za rešavanje problema, posmatranje trendova, ili opisivanje situacije (srednja vrednost i standardna devijacija). Ta nova informacija može da se preslika na realni svet i primeni kao deo rešenja empirijskog problema. U ovoj knjizi videćemo primere kako se merenje koristi za podršku analizi i donošenju odluka.



SLIKA 1.15 Merenja kao pomoć prilikom traženja rešenja

Alati i integrisana okruženja. Godinama su prodavci softvera reklamirali CASE (*Computer-Aided Software Engineering*) alate, kod kojih standardizovano i integrisano razvojno okruženje unapređuje razvoj softvera. Međutim, videli smo kako različiti učesnici u razvoju koriste različite procese, metode i resurse. Što se tiče jedinstvenog pristupa, lakše je reći nego uraditi.

S druge strane, istraživači su predložili nekoliko okvira koji nam omogućavaju da poredimo i suprotstavljamo postojeća i predložena razvojna okruženja. Ti okviri nam dozvoljavaju da istražimo usluge koje pružaju razvojna okruženja za softversko inženjerstvo, i da odlučimo koje je okruženje najpogodnije za dati problem ili razvoj aplikacije.

Jedna od glavnih poteškoća prilikom poređenja alata je činjenica da se njihovi proizvođači retko bave celokupnim razvojnim ciklusom. Oni se obično bave malim skupom aktivnosti, kao što su dizajn ili testiranje, a na samom korisniku je da integriše odabrane alate u kompletno razvojno okruženje. Wasserman (1990) je identifikovao pet pitanja kojima svaki postupak integrisanja alata mora da se pozabavi:

1. integracija platforme: sposobnost alata da radi u heterogenoj mreži;
2. integracija prezentacije: zajednički korisnički interfejs;
3. integracija procesa: povezivanje alata i procesa razvoja;
4. integracija podataka: način na koji alati dele podatke;
5. integracija kontrole: mogućnost da jedan alat nešto saopšti drugom alatu ili inicira neku njegovu akciju.

U svakom od sledećih poglavlja ove knjige, istražićemo alate koji podržavaju aktivnosti i koncepte opisane u ovom poglavlju.

Osam ovde opisanih koncepata mogu biti shvaćeni kao osam niti pomoću kojih je prošivena ova knjiga, i koje povezuju potpuno različite aktivnosti u sklopu discipline koju zovemo softversko inženjerstvo. Kada budemo naučili nešto više o softverskom inženjerstvu, podsetićemo se na te ideje, da bismo videli kako one ujedinjuju i uzdižu softversko inženjerstvo kao naučnu disciplinu.

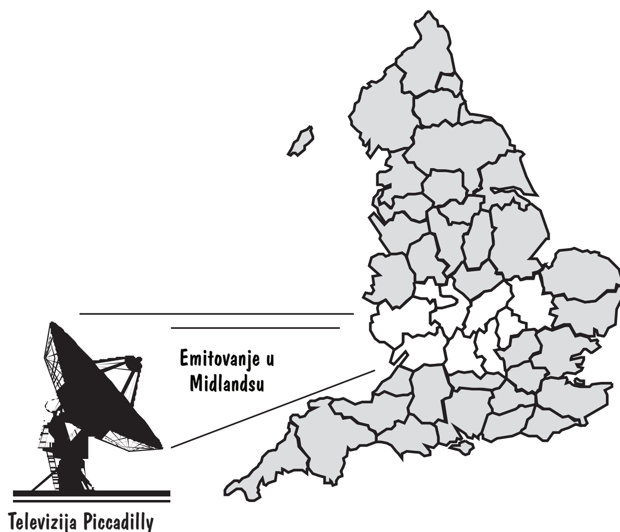
1.9 PRIMER IZ INFORMACIONOG SISTEMA

U sklopu ove knjige, videćemo da se svako poglavlje završava sa dva primera, i to prvo primerom iz informacionog sistema, a zatim primerom iz sistema koji radi u realnom vremenu. Koncepte koji su opisani u datom poglavlju, primenićemo na neke aspekte svakog primera, tako da možemo videti šta koncepti znače ne samo teoretski već i u praksi.

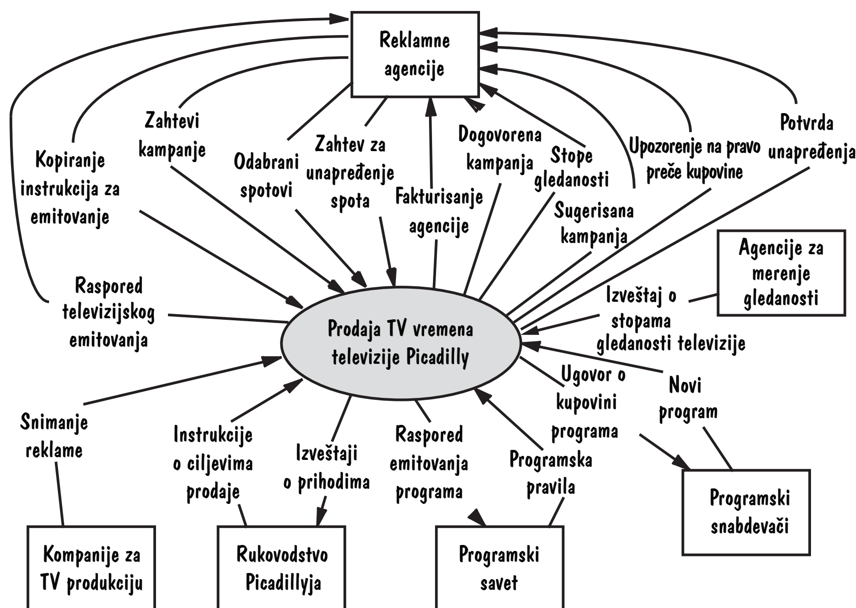
Naš primer iz informacionog sistema je preuzet (sa dozvolom) iz knjige *Complete Systems Analysis: The workbook, the Textbook, the Answers*, autora Jamesa i Suzanne Robertson (Robertson i Robertson 1994). Primer obuhvata razvoj sistema za samooglašavanje stanice Picadilly Television, vlasnika regionalne licence britanske televizije. Slika 1.6 ilustruje oblasti vidljivosti televizije Picadilly. Kao što ćemo videti, ograničenja koja se odnose na cenu televizijskog vremena su mnoga i promenljiva što problem čini zanimljivim i teškim. U ovoj knjizi naglašavamo aspekte problema i rešenje, dok knjiga Robertsonovih detaljno prikazuje metode za prikupljanje i analizu zahteva sistema.

U Velikoj Britaniji radiodifuzna agencija izdaje osmogodišnje licence komercijalnim televizijskim kompanijama, dajući im ekskluzivna prava da emituju svoje programe u okviru pažljivo definisanog regiona ili okruga. Zauzvrat, nosilac licence mora da emituje dramski program, komedije, sport, dečije i ostale programe, u propisanoj ravnoteži. Pored toga, postoje ograničenja o tome koji programi mogu da se emituju u koje vreme, kao i pravila o sadržaju programa i komercijalnog oglašavanja.

Komercijalni oglašivač ima nekoliko izbora da stigne do publike regiona Midlands, a to su: Picadilly, kablovski kanali i satelitski kanali. Međutim, Picadilly privlači većinu gledalaca. Prema tome, Picadilly mora da dostigne određenu stopu gledanosti da bi privukao deo budžeta oglašivača. Jedan od načina da privuče pažnju oglašivača je pomoću indeksa gledanosti koji odražava broj i tip gledalaca u različitim trenucima dana. Indeksi se daju zavisno od tipa programa, tipa publike, vremena u okviru dana, televizijske kompanije itd. Ali stopa oglašavanja ne zavisi samo od stope gledanosti. Na primer, cena po času može biti niža ako oglašivač kupi veći broj časova. Pored toga, postoje ograničenja tipa oglašavanja u određeno vreme i za određene programe. Na primer:



SLIKA 1.16 Oblast za koju televizija Picadilly ima dozvolu emotovanja



SLIKA 1.17 Kontekstni dijagram televizije Picadilly sa prikazanim granicama sistema (Robertson i Robertson 1994)

- Oglasi za alkoholna pića mogu da se prikazuju samo posle 9 uveče.
- Ako neki glumac učestvuje u emisiji, tada reklama sa istim glumcem ne sme da se emituje u periodu do 45 minuta posle emisije.
- Ako je reklama za neku klasu proizvoda (kao što je automobil) raspoređena u određenom reklamnom bloku, onda nijedna druga reklama nečega iz te klase ne može da se prikazuje tokom tog reklamnog bloka.

Kada budemo detaljnije ispitali ovaj primer, napomenućemo dodatna pravila i propise o oglašavanju i ceni. Konteksttni dijagram sistema na slici 1.17 prikazuje granice sistema i u kakvom je odnosu sistem sa datim pravilima. Zasenčena elipsa, koja predstavlja na slici Picadillyjev sistem, je upravo naš primer iz informacionog sistema, pri čemu je granica obvojnica elipse. Strelice i pravougaonici prikazuju stavke koje utiču na rad Picadillyjevog sistema, i posmatraćemo ih kao skup ulaza i izlaza, svaki sa svojim izvorom i odredištem.

U kasnijim poglavljima otkrićemo aktivnosti i elemente koji se nalaze unutar zasenčene elipse (tj. unutar granica sistema). Ispitaćemo izgled projekta i razvoj ovog sistema, koristeći tehnike softverskog inženjerstva koje će biti opisane u sledećim poglavljima.

1.10 PRIMER IZ PRAKSE

Naš primer sistema sa radom u realnom vremenu zasniva se na softveru ugrađenom u svemirskoj raketi Ariane-5, koja pripada evropskoj svemirskoj agenciji ESA (*European Space Agency*). Na letu prvencu, koji se odigrao 4. juna 1996., raketa Ariane-5 bila je

lansirana i ponašala se izvrsno nekih 40 sekundi. Zatim je počela da menja kurs. Prema naredbi zemaljske kontrole Ariane, raketa je uništena daljinskom komandom. Uništenje neosigurane rakete bio je gubitak ne samo rakete, već i četiri satelita koji su se u njoj nalazili. Ukupna šteta bila je oko 500 miliona dolara (Newsbytes matična stranica 1996; Lions i dr. 1996).

Softver je bio uključen u skoro sve aspekte sistema, od vođenja rakete, do internog rada njenih sastavnih delova. Otkazivanje rakete i zatim njeno uništavanje, postavljaju mnoga pitanja o kvalitetu softvera. Kao što ćemo videti u sledećim poglavljima, istražna komisija koja je ispitivala uzroke problema svu pažnju je usmerila na kvalitet softvera i njegovo obezbeđivanje. U ovom poglavlju, posmatrali smo kvalitet zavisno od poslovne vrednosti rakete.

Bilo je mnogo organizacija koje su bile stubovi uspeha Ariane-5. To je pre svega ESA, zatim francuska svemirska agencija CNES (*Centre National d´Etudes Spatiales*), koja je bila zadužena za sveopšte upravljanje programom Ariane, kao i 12 drugih evropskih zemalja. Gubitak rakete bio je samo još jedan događaj u nizu kašnjenja i problema koji su pogodili program razvoja rakete Ariane, kao recimo 1995., kada se dogodilo curenje kiseonika tokom testiranja motora i koje je usmrtilo dva inženjera. Međutim, incident iz juna 1996. je bio prvi koji je imao direktno veze sa otkazom softvera.

Posledice tog incidenta daleko su nadmašile pomenutih 500 miliona dolara izgubljene opreme. U 1996. godini, raketa Ariane-4 i prethodne varijante prednjačile su u lansiranju, ispred američkih, ruskih i kineskih. Stoga su u pitanje dovedeni kredibilitet programa i potencijalni budući poslovi u vezi sa raketama Ariane.

Budući poslovi bili su delom zasnovani na sposobnosti nove rakete da u orbitu nosi teži teret nego što su to mogli prethodni lansirni sistemi. Ariane-5 je bila projektovana da nosi jedan satelit do 6,8 tona, ili dva satelita sa ukupnom težinom od 5,9 tona. U daljem radu na razvoju do 2002. godine, očekivalo se dodavanje još jedne tone lansirnog kapaciteta. Uvećani kapacitet u pogledu nosivosti bio je čista poslovna prednost. Satelitski operatori često smanjuju troškove zajedničkim lansiranjima, što znači da je na raketi Ariane mogao da se lansira teret više kompanija.

Razmotrićemo šta kvalitet znači u kontekstu ovog primera. Ispostavilo se da je uništenje Ariane-5 rezultat loše specifikacije programskih zahteva koje je dao kupac. U ovom slučaju, razvojni tim može da tvrdi da je sistem visokog kvaliteta, ali da je sagrađen po pogrešnoj specifikaciji. Zaista, istražna komisija, formirana radi ispitivanja uzroka i uklanjanja posledica nesreće konstatuje sledeće:

> Komisijski nalazi zasnovani su na temeljnim i otvorenim prezentacijama projektnog tima Ariane-5, kao i na dokumentaciji koja je pokazala visok kvalitet programa Ariane-5 u pogledu inženjerskog rada, u opštem smislu i zaokruženosti i mogućnosti praćenja dokumenata (Lions i dr. 1996).

Međutim, sa stanovišta korisnika i kupaca, postupak izrade specifikacije je trebao da bude dovoljno dobar da prepozna nedostatke u specifikaciji i da natera kupca da ispravi specifikaciju pre nego što je nastala šteta. Istražna komisija potvrdila je sledeće:

> Isporučilac sistema SRI (podsistema u kojem je na kraju lokalizovan uzrok problema), sledio je isključivo specifikaciju koja mu je bila data, a koja je uslovljavala da u slučaju svakog otkrivenog izuzetka procesor treba da se zaustavi. Izuzetak koji se dogodio nije bio zbog slučajnog otkaza, već zbog greške u projektu. Izuzetak je bio prepoznat, ali nije bio pravilno prihvaćen i obrađen, zbog zauzetog stanovišta da se taj softver smatra ispravnim sve dok ne pokaže da ima nedostatak. Komisija ima razloga da veruje da je to stanovište zauzeto i u drugim oblastima softverskog projekta za raketu Ariane-5. Komisija zastupa suprotno stanovište, da za softver treba pretpo-

stavljati da ima nedostatke, sve dok se primenom najboljih postojećih metoda prihvaćenih u praksi ne dokaže da je softver ispravan (Lions i dr. 1996).

Ovim primerom ćemo se detaljnije pozabaviti u kasnijim poglavljima, gde ćemo videti kako odluke koje donose razvojni tim ili kupac utiču na izgled projekta, testiranje i održavanje. Videćemo kako loše sistemsko inženjerstvo na početku razvoja dovodi do niza pogrešnih odluka, koje dalje dovode do katastrofe. S druge strane, otvorenost svih zainteresovanih, uključujući agenciju ESA i istražnu komisiju, uz visokokvalitetnu dokumentaciju i ozbiljnu želju da se što pre dođe do istine, imali su za posledicu brzo rešenje iskrslog problema, uz donošenje efikasnog plana da se takvi problemi u budućnosti spreče.

Sistemski pogled omogućio je istražnoj komisiji, u saradnji sa razvojnim timom, da gleda na raketu Ariane-5 kao na skup podsistema. Taj skup je izraz analize problema, prema načelima opisanim u ovom poglavlju, što znači da različiti učesnici u razvoju mogu da rade na odvojenim podsistemima, sa izrazito odvojenim funkcijama. Na primer:

> Položaj rakete nosača i njeno pokretanje u svemiru mereni su pomoću sistema SRI (*Inertial Reference System*). Taj sistem ima sopstveni računar, gde se uglovi i brzine računaju na osnovu informacija od inercijalne „*strap-down*" platforme, sa laserskim žiroskopom i meračem ubrzanja. Podaci dobijeni od sistema SRI se prenose preko magistrale podataka ka računaru OBC (*On-Board Computer*), na kome se izvršava letački program i kontrolišu mlaznice potisnih motora na čvrsto gorivo, kao i kriogenski Vulcain motor, preko servo ventila i hidrauličkih pokretača (Lions i dr. 1996).

Međutim, sinteza rešenja mora da obuhvati pregled svih sastavnih delova, pri čemu se delovi posmatraju zajedno, radi određivanja da li je „lepak", koji sve te delove drži zajedno, dovoljan i odgovarajući. U slučaju rakete Ariane-5, istražna komisija sugerisala je da kupac i razvojni tim moraju da rade zajedno na pronalaženju kritičnog softvera, kao i na proverama da li softver može da pravilno reaguje ne samo na predviđeno, već i na nepredviđeno ponašanje.

> To znači da se kritični softver (tj. gde otkaz softvera dovodi misiju u pitanje) mora identifikovati na vrlo detaljnom nivou, da se izuzetno ponašanje mora ograničiti, kao i da razumna politika pravljenja rezervnih sistema mora da uzme u obzir i softverske otkaze (Lions i dr. 1996).

1.11 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju uvedeni su mnogi koncepti koji predstavljaju suštinu dobrog softverskog inženjerstva, kako u praksi, tako i u istraživanjima. Od strane pojedinačnih učesnika u razvoju softvera, ti koncepti se mogu koristiti na sledeće načine:

- Kada nam se predoči problem koji treba da rešimo (bez obzira na to da li rešenje uključuje ili ne uključuje softver), možemo ga analizirati dekomponovanjem na sastavne delove i veze između njih. Nakon toga, sintetizujemo rešenje, tako što rešavamo zasebne potprobleme i njihova rešenja integrišemo u jedinstvenu celinu.
- Morate shvatiti da su zahtevi promenljivi, čak i ako ista osoba analizira i rešava problem. Stoga rešenje treba da bude dobro dokumentovano i fleksibilno, a dokumentovati treba sve pretpostavke i algoritme koji se koriste (koje je onda lako kasnije izmeniti).

- Na kvalitet treba gledati iz nekoliko različitih uglova, shvatajući pri tom da tehnički kvalitet i poslovni kvalitet mogu da budu veoma različiti.
- Apstrakcija i merenje su pomoć za identifikovanje suštinskih aspekata problema i rešenja.
- Treba imati na umu i granice sistema, tako da se rešenja ne preklapaju sa sistemima koji su u interakciji sa onim koji mi gradimo.

1.12 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

Pretežan deo rada pojedinac obično obavlja kao član većeg razvojnog tima. Kao što smo videli u ovom poglavlju, razvoj obuhvata analizu zahteva, projektovanje, implementaciju, testiranje, upravljanje konfiguracijom, osiguranje kvaliteta i drugo. Pojedinci u timu mogu imati više uloga, pri čemu uspeh projekta u velikoj meri zavisi od komunikacije i koordinacije između članova tima. Videli smo u ovom poglavlju da uspehu projekta možemo da doprinesemo odabiranjem:

- procesa razvoja koji je pogodan za tim određene veličine, nivo rizika i za oblast primene;
- alata koji su dobro integrisani i podržavaju tip komunikacije koji se zahteva za dati projekat;
- alata za merenje i podršku, koji pružaju veći uvid i razumevanje.

1.13 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Mnoga pitanja koja su razmatrana u ovom poglavlju, dobar su predmet za dalja istraživanja. Napomenuli smo neka otvorena pitanja, uključujući potrebu da pronađemo:

- odgovarajuće nivoe apstrakcije da bismo olakšali rešavanje problema;
- odgovarajuća merenja da bismo otkrili suštinsku prirodu problema i rešenja, i da bi oni bili korisni;
- odgovarajuću dekompoziciju problema, gde je svaki potproblem rešiv;
- zajednički okvir ili notaciju za omogućavanje lake i efikasne integracije alata, kao i za maksimizovanje komunikacije između učesnika u projektu.

U kasnijim poglavljima opisaćemo mnoge tehnike. Neke su već korišćene i predstavljaju potvrđenu praksu u razvoju softvera, dok su druge samo predložene i demonstrirane na malim „igračkama” ili studentskim projektima. Nadamo se da ćemo vam pokazati kako da unapredite ono što radite sada i u isto vreme da vas inspirišemo da budete kreativniji i da u budućnosti razmišljate o isprobavanju novih tehnologija i procesa.

1.14 SEMINARSKI RAD

Nemoguće je učiti softversko inženjerstvo bez učešća u razvoju softverskog projekta zajedno sa vašim kolegama. Zato u svakom poglavlju ove knjige opisujemo seminarski rad koji možete da uradite sa timom školskih drugova. Projekat, zasnovan na realnom sistemu u realnoj organizaciji, omogućiće vam da se pozabavite vrlo realnim izazovima koje pred vas postavljaju analiza, projektovanje, implementacija, testiranje i održavanje. Pored toga, radeći u timu bavićete se i pitanjima razlika unutar tima i upravljanja projektom.

Seminarski rad predstavlja utvrđivanje vrste zajma koji možete ugovoriti sa bankom kada želite da kupite kuću. Banke generišu prihod na mnoge načine, često pozajmljujući novac od svojih ulagača po niskoj kamatnoj stopi, a zatim dajući na pozajmicu isti novac po višoj kamatnoj stopi. Međutim, dugoročni krediti za nekretnine, kao što su hipotekarni krediti, obično imaju rok otplate od 15, 25 ili čak 30 godina. To jest, morate 15, 25 ili 30 godina da otplaćujete kredit: glavnicu (novac koji ste prvobitno pozajmili) plus kamatu po određenoj stopi. Iako je prihod od kamata na takve kredite profitabilan, pozajmice dugoročno vezuju novac, sprečavajući banke da koriste svoj novac za druge transakcije. Zbog toga banke često prodaju svoje pozajmice organizacijama za konsolidaciju, uzimajući manji dugoročni profit uzamenu za oslobađanje kapitala koji mogu koristi za druge transakcije.

Aplikacija seminarskog rada zove se „Aranžer pozajmica” (*Loan Arranger*). Oblikovana je na način kako (nepostojeća) organizacija FCO (*Financial Consolidation Organization*) rukuje sa pozajmicama koje kupuje od banaka. Konsolidaciona organizacija pravi novac kupujući pozajmice od banaka i prodajući ih investitorima. Banka prodaje pozajmicu FCO organizaciji, dobijajući glavnicu nazad. Zatim, kao što ćemo videti, FCO prodaje pozajmicu investitorima koji su radi da čekaju duže nego banka da bi im se uloženo vratilo.

Da bismo videli kako ta transakcija radi, razmotrićemo kako se dobija pozajmica za kuću (hipotekarni kredit). Kuću od 150 000 dolara možete da kupite plaćajući 50 000 dolara kao učešće i uzimajući pozajmicu za preostalih 100 000 dolara. Uslovi koje daje First National Bank mogu biti, na primer, pozajmica na 30 godina sa 5% kamate. To znači da First National Bank daje rok od 30 godina (rok pozajmice) za otplatu pozajmljenog iznosa (glavnice) i kamate na sve ono što nije odmah otplaćeno. Na primer, 100 000 dolara može da se otplati uplaćujući jedanput mesečno u roku od 30 godina (tj. 360 rata), sa kamatom na neotplaćeni dug. Ako je početno stanje 100 000 dolara, banka izračunava mesečnu ratu koristeći veličinu glavnice, kamatne stope, dužine vremena u kome treba otplatiti pozajmicu, uz pretpostavku da sva mesečna plaćanja treba da predstavljaju isti iznos.

Na primer, pretpostavimo da banka kaže da mesečna rata treba da bude 536,82 dolara. Kamata za prvi mesec je (1/12) × (0,05) × (100 000), ili 416,67 dolara. Ostatak rate (536,82 - 416,67) je uplata za umanjenje glavnice, a to je 120,15 dolara. U drugom mesecu dug je 100 000 dolara manje 120,15, tako da je kamata umanjena na (1/12) × (0,05) × (100 000 -120,15), ili 416,17 dolara. Tako, tokom drugog meseca, samo 416,17 dolara mesečne rate predstavlja kamatu, a ostatak od 102,65 se odnosi na preostalu glavnicu. Tokom vremena, što se više smanjuje glavnica, plaća se manja kamata, sve dok se ne otplati čitava glavnica a vi postanete vlasnik nekretnine, neopterećene dugom banci.

First National Bank može da proda vašu pozajmicu organizaciji FCO u nekom trenutku tokom perioda otplaćivanja. First National i FCO pregovaraju o ceni. Dalje, FCO može da proda vašu pozajmicu korporaciji ABC Investment. Vi još uvek mesečno otplaćujete hipotekarni kredit, ali uplata ide u ABC, a ne u First National. Obično, FCO prodaje svoje pozajmice „u paketu”, a ne pojedinačno, tako da investitor kupuje skup pozajmica, na osnovu rizika, veličine glavnice i očekivane stope povraćaja. Drugim rečima, investitor kao što je ABC može da kontaktira FCO i navede koliko novca želi da investira, na koji period, koji rizik želi da preuzme (na osnovu biografija ljudi ili organizacija koje otplaćuju pozajmicu), i koliki profit očekuje.

Loan Arranger je aplikacija koja omogućava FCO analitičaru da odabere pakete pozajmica koje odgovaraju željama investitora. Aplikacija pristupa informacijama o pozajmicama koje je kupio FCO od raznih kreditnih institucija. Kada investitor odredi investicione kriterijume, sistem odabira optimalne pakete pozajmica koje zadovoljavaju kriterijum. Sistem treba da omogućava neke napredne optimizacije, kao što je biranje najboljih paketa pozajmica iz podskupa raspoloživih (na primer, od svih pozajmica u Massachusettsu, umesto od svih raspoloživih pozajmica). Pored toga, sistem treba ipak da dozvolianalitičaru datog klijenta da ručno bira pakete pozajmica. Sistem takođe automatizuje aktivnosti oko upravljanja informacijama, kao što su ažuriranje informacija o bankama, ažuriranje informacija o pozajmicama, kao i dodavanje novih pozajmica, kada banka jednom mesečno obezbedi takvu informaciju.

Da zaključimo, sistem Loan Arranger omogućava analitičaru pozajmica da pristupi informacijama o hipotekarnim kreditima (za kuće, ovde ih zovemo prosto „pozajmice") koje je FCO kupio od više kreditnih institucija, sa namerom prepakivanja pozajmica radi prodaje drugom investitoru. Pozajmice koje je FCO kupio radi investiranja i dalje prodaje, zbirno se zovu kreditni portfelj. Sistem Loan Arranger zapisuje te hartije od vrednosti u repozitorijumu informacija o pozajmicama. Analitičar pozajmica može da doda, gleda, ažurira ili obriše informaciju o kreditorima skupu pozajmica u portfoliju. Pored toga, sistem omogućava analitičaru da pravi pakete pozajmica radi prodaje investitorima. Korisnik sistema Loan Arranger je analitičar pozajmica, koji vodi računa o hipotekarnim kreditima koje je FCO kupio.

U kasnijim poglavljima istražićemo detaljnije sistemske zahteve. Za sada, ako je potrebno da se podsetite znanja o glavnici i kamati, nije loše da pogledate stare knjige iz matematike, ili da pogledate stranicu http://www.interest.com/hugh/calc/formula.html.

1.15 KLJUČNE REFERENCE

Informacije o softverskim nedostacima i otkazima mogu se pronaći na Risks Forumu, čiji je moderator Peter Neumann. Papirna verzija informacija koje se tamo nalaze štampa se u svakom izdanju *Software Engineering Notes*, čiji je izdavač SIGSOFT (*Special Interest Group on Software Engineering*) u okviru udruženja Association for Computer Machinery. Arhive rizika su raspoložive na ftp.sri.com, u odeljku risks. Konferencijska grupa na Risks Forumu je na internetu, na comp.risks, ili možete da se učlanite preko servera za automatske liste na risks-request@CSL.sri.com.

O projektu Ariane-5 više može da se pronađe na veb lokaciji organizacije European Space Agency, na adresi http://www.esrin.esa.it/htdocs/esa/ariane. Primerak zajedničke izjave za štampu ESA/CNES povodom neuspeha misije nalazi se na adresi http://www.esrin.esa.it/htdocs/tidc/Press/Press96/press19.html (na engleskom) . Francuska verzija izjave za štampu je na adresi http://www.cnes.fr/Acces/Espace/Vol_50x.html. Elektronska kopija dokumenta Ariane-5 Flight 501 Failure Report je na http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

Leverson i Turner (1993) opisali su detaljno softverski projekat i problematiku testiranja mašine Therac.

Izdanje časopisa IEEE Software iz januara 1996. posvećeno je kvalitetu softvera. Naročito u uvodnom članku autori Kitchenham i Pfleeger (1996) opisuju i kritikuju više scenarija za postizanje kvaliteta, dok autor Dromey (1996) u svom članku izlaže o definisanju kvaliteta na merljiv način.

U cilju dobijanja više informacija o primeru televizije Picadilly, pogledajte referencu (Robertson i Robertson 1994), ili istražite pristup Robertsonovih prema utvrđivanju zahteva, na adresi www.systemsguild.com.

1.16 VEŽBE

1. Sledeći članak pojavio se u *Washington Postu* (Associated Press 1996):

KAO RAZLOG PADA AVIONA NAVEDENA JE PILOTOVA RAČUNARSKA GREŠKA. AMERIČKI AVIOPREVOZNIK KAŽE DA JE JEDNO SLOVO U OZNACI BILO RAZLOG UDARA MLAZNJAKA U PLANINU U KOLUMBIJI.

Dalas, 23. avgust – Kapetan mlaznjaka kompanije American Airlines, koji se srušio u Kolumbiji prošlog decembra, uneo je pogrešnu jednoslovnu računarsku komandu koja je poslala avion u planinu, rekao je danas avioprevoznik.

U avionu je bilo 163 putnika, od kojih je samo četvoro preživelo pad.

Istražitelji American Airlinesa zaključili su da je kapetan Boinga 757 očigledno mislio da je uneo koordinate Calija, nameravanog odredišta. Međutim, na većini južnoameričkih aeronautičkih karti, jednoslovni kod za Cali je ista kao i za Bogotu, 132 milje u suprotnom smeru.

Koordinate za Bogotu uputile su avion prema planini, kako kaže u pismu Cecil Ewell, glavni pilot American Airlinesa i potpredsednik zadužen za letenje. Oznake za Bogotu i Cali su različite u mnogim računarskim bazama podataka, rekao je Ewell.

Portparol American Airlinesa, John Hotard potvrdio je da je Ewellovo pismo, prvi put objavljeno u listu *Dallas Morning News*, podeljeno ove nedelje svim pilotima ovog avioprevoznika, kao upozorenje na problem sa kodovima.

Otkriće American Airlinesa podstaklo je Federal Aviation Administration da izda proglas svim avioprevoznicima, upozoravajući ih na neusklađenosti između računarske baze podataka i aeronautičkih karti, kaže se u pomenutom listu.

Računarska greška ne predstavlja poslednju reč o tome šta je uzrok pada. Kolumbijska vlada je vršila istraživanja, a očekuje se da do oktobra objavi svoja saznanja.

Pat Carisco, portparol National Transportation Safety Boarda, kaže da kolumbijski istražitelji ispituju faktore kao što su obuka letačkog osoblja i kontrola letenja.

Računarsku grešku pronašli su istražitelji American Airlinesa, kada su poredili podatke sa navigacionog računara mlaznjaka, sa informacijama iz olupine, kaže Ewell.

Pogrešni podaci ostali su neotkriveni tokom 66 sekundi, dok je posada zbunjeno pratila naredbe kontrole letenja, da bi zauzela direktniji prilaz aerodromu Cali.

Tri minuta kasnije, dok se avion još uvek spuštao, a posada pokušavala da shvati zašto se avion okrenuo, on se srušio.

Ewell kaže da je pad predstavljao dve važne lekcije za pilote.

„Pre svega, nije bitno koliko mnogo puta ste išli u Južnu Ameriku ili neko drugo mesto – Stenovite planine – ne možete nikada, nikada i nikada bilo šta da pretpostavljate”, rekao je on u izjavi za štampu. Drugo, dodao je on, piloti moraju da shvate da ne mogu da dozvole da automatizacija preuzme odgovornost za let aviona.

Da li ovaj članak predstavlja dokaz da imamo softversku krizu? Da li je avijacija bogatija zbog softverskog inženjerstva? Kojim pitanjima treba da se bavimo tokom razvoja softvera tako da se problemi poput ovoga spreče u budućnosti?

2. Dajte primer analize problema gde su komponente problema relativno jednostavne, ali poteškoća u rešavanju problema leži u međusobnim vezama između potproblemskih komponenti.
3. Objasnite razliku između greške, nedostatka i otkaza. Dajte primer greške koja dovodi do nedostatka u programskim zahtevima, u načinu projektovanja i u programskom kodu. Dajte primer nedostataka u programskim zahtevima koji dovode do softverskog otkaza, zatim nedostatka u projektovanju koji dovodi do otkaza, kao i nedostatka u podacima za test koji dovodi do otkaza.
4. Zašto broj grešaka može da zavede prilikom merenja kvaliteta proizvoda?
5. Mnogi koji rade u razvoju izjednačavaju tehnički kvalitet sa ukupnim kvalitetom proizvoda. Dajte primer proizvoda čiji je tehnički kvalitet visok, ali ga kupac ne smatra kvalitetnim. Da li postoje etički problemi kada se pitanje kvaliteta sužava samo na tehnički kvalitet? Ilustrujte svoje stanovište na primeru sistema Therac-25.
6. Mnoge organizacije kupuju komercijalni softver, misleći da je tako jeftinije nego da razviju i održavaju softver u okviru kuće. Opišite prednosti i mane korišćenja COTS softvera. Na primer, šta se događa ako proizvođači COTS-ovih proizvoda više ne pružaju podršku? Šta moraju kupac, korisnik i razvojni tim da predvide kada projektuju proizvod koji koristi COTS softver u većem sistemu?
7. Koje su pravne i etičke implikacije upotrebe COTS softvera? Ili saradnje sa podizvođačima? Na primer, ko je odgovoran za ispravljanje problema kada glavni sistem ne radi zbog nedostatka u COTS softveru? Ko je odgovoran kada takav otkaz prouzrokuje štetu korisnicima, bilo direktnu (kao kada otkažu automatske kočnice na kolima), ili indirektnu (kada se pogrešna informacija prosleđuje drugom sistemu, kao što smo videli u vežbanju 1). Kakve su provere i odnosi potrebni za osiguravanje kvaliteta COTS softvera, pre nego što se integriše u veći sistem?
8. Primer televizije Picadilly, kao što je ilustrovano na slici 1.17, sadrži veliki broj pravila i ograničenja. Razmotrite tri od njih i objasnite prednosti i mane činjenice što se nalaze van granica sistema.
9. Kada je uništena raketa Ariane-5, ta vest je došla na prve stranice štampe u Francuskoj i drugde. Francuski list *Liberasion*, nazvao je to na naslovnoj strani „vatrometom od 37 milijardi franaka". U stvari, eksplozija je bila vest sa prvih stranica u skoro svim evropskim novinama, a bila je i glavna vest večernjih dnevnika većine evropskih TV mreža. Nasuprot tome, hakerska invazija na Panix, njujorškog dobavljača internetskih usluga, prouzrokovala je višečasovni ispad i nedostupnost Panix sistema. Vest koja je opisivala ovaj događaj pojavila se samo na naslovnoj stranici privredne rubrike *Washington Posta*. Kakva je odgovornost novinara kada izveštavaju o softverskim incidentima? Kako treba da se ocenjuje potencijalni uticaj softverskih otkaza i da se o njima izveštava?