14 - Debagovanje

Saša Malkov

Odlomak iz knjige Razvoj softvera
(u pripremi)

Deset malih bagova u kodu se skrilo,
pre poslednje popravke devet ih je bilo...
S.M.

14.1 Greške

Razvoj softvera je složen proizvodni proces. Kao i u svakom drugom proizvodnom procesu, i u toku razvoja softvera neminovno dolazi do pravljenja određenih propusta, koji se ispoljavaju na različite načine. U zavisnosti od faze razvijanja softvera u kojoj nastaju, propusti se mogu značajno razlikovati po složenosti, prikrivenosti i načinu ispoljavanja.

14.1.1 Greška, propust ili bag

Propusti u razvoju softvera se često nazivaju *greškama* ili *bagovima*. Primetimo da postoji određen problem sa upotrebom termina *greška*, zato što je u praksi prilično uobičajeno da se pojam procesa razvoja softvera izjednačava sa njegovim konačnim proizvodom – softverom, kao što se i konačan proizvod često povezuje sa jednom od

završnih faza izrade – programiranjem. Kada se kaže da u nekom programu postoji *greška* ili *bag*, obično se to vezuje sa propustima u programiranju. Međutim, termin *greška* u razvoju softvera ima mnogo šire značenje i obuhvata sve one propuste koji mogu da nastanu u svim fazama razvoja softvera, kao što su, na primer, analiza zahteva, planiranje, projektovanje ili programiranje.

U prvim odeljcima ovog poglavlja ćemo se baviti svim vrstama grešaka, a ne samo greškama u fazi programiranja, pa ćemo ravnopravno koristiti termine *greška* i *propust*. Kasnije, kako se budemo više posvećivali konkretnom problemu lociranja i otklanjanja grešaka u programskom kodu, *greške* će postepeno istisnuti *propuste* iz teksta.

Propusti prave različite vrste posledica. Uobičajeno kažemo da je *propust* sve ono što ima za posledicu *neispravno ponašanje* softvera. U zavisnosti od konteksta, *neispravno ponašanje* može da se ispolji kao izračunavanje neispravnog rezultata, nedopustivo veliko trajanje izračunavanje, neprihvatanje novih zahteva korisnika, neizdavanje rezultata i drugo.

Najuopštenije posmatrano, propust u razvoju softvera (greška, bag) je sve ono što stvara probleme u funkcionisanju softvera kao završnog proizvoda. Formalnije, možemo reći da propusti predstavljaju „sve ono što ima za posledicu da se softver ne ponaša u skladu sa specifikacijom“[48].

14.1.2 Dinamička priroda grešaka

Softver ima izrazito dinamičnu prirodu. Sve vreme tokom izvršavanja programa odvijaju se složene, međusobno povezane i uslovljene, operacije promene stanja programa i računarskog sistema. Ako svedemo posmatranje na pojedinačne atomične promene stanja, onda možemo da kažemo da je svaka od njih izrazito jednostavna, ali da ih je neverovatno mnogo – svake sekunde po nekoliko miliona ili milijardi. Sa druge strane, ako ih posmatramo po grupama, kao nešto veće složene promene stanja, onda se njihov broj smanjuje, ali njihova unutrašnja složenost nam otežava praćenje i analizu.

Posledica takve prirode softvera je da i greške u softveru imaju dinamičku prirodu. U jednoj od najboljih knjiga napisanih o debagovanju, Andreas Zeler [Zeller2006] je dinamičku prirodu grešaka pokušao da opiše prolaskom kroz četiri glavne faze kroz od nastajanja do ispoljavanja greške. Taj put se obično naziva sekvencom *bag-infekcija-infekcija-neuspeh*:

[48] Kao i svaka druga neformalna definicija, ni ova nije savršena, zato što ne može da se primeni na propuste koji nastaju tokom izrade specifikacije i prethodnih faza.

1. Programer[49] piše neispravan programski kod. Ta načinjena greška predstavlja uzrok *infekcije*.
2. Neispravnost prouzrokuje infekciju. Sa izvršavanjem programa izvršava se i neispravan deo programa, što za rezultat ima neispravno stanje programa.
3. Infekcija se prenosi. Neispravno stanje programa utiče na izvršavanje drugih delova programa. Posledica je da se neispravnosti postepeno pojavljuju i u drugim elementima stanja programa.
4. Infekcija izaziva neuspeh. Pod neuspehom se misli na spolja vidljivu grešku u ponašanju programa.

Pri tome, naravno, svaka od ovih faza ima svoje specifičnosti. Najpre, kada se pri pisanju programa napravi greška, ona vrlo često ne ide sama, već se u njenoj neposrednoj blizini često nalazi još neka greška.

Put infekcije je vrlo retko linearan – pre bi se moglo reći da ima prirodu eksponencijalnog širenja nalik na grananje drveta ili grafa. Ako se neispravan programski kod izvršava više puta, na različitim podacima, onda je moguće da jedna greška proizvede mnogo infekcija već u drugom koraku. Koliko god čudno izgledalo, takvi slučajevi su poželjniji, zato što se infekcija vrlo brzo širi i skoro neminovno dolazi do skorog neuspeha i uočavanja da postoji problem. Nasuprot tome, mnogo su nezgodnije greške koje imaju tendenciju da retko prenose infekciju, zato što mogu dugo da ostanu neprimećene.

Slično je i u trećem koraku – kao što neka greška ne mora da vodi infekcijama, tako se i infekcije različitom brzinom šire i manifestuju. Jedan broj infekcija se zadržava u unutrašnjem stanju programa i veoma retko se prenosti na spoljašnje elemente stanja i izaziva neuspehe. U tom slučaju imamo sličan slučaj kao sa greškom koja retko izaziva infekcije – ponovo imamo problem sa greškama koje će dugo ostati neprimećene.

Ovakva priroda grešaka ima za posledicu da skoro svaki put kada se bavimo otkrivanjem grešaka čiji su neuspesi uočeni, moramo da se bavimo ne samo traženjem i izučavanjem grešaka, nego najpre traženjem i izučavanjem *infekcija* i puteva infekcija. Iako ćemo u daljem tekstu uglavnom govoriti o greškama, zapravo se podrazumeva da svaki put proveravanjem ispravnosti ponašanja delova programa mi zapravo proveravamo ispravnost stanja programa, tj. proveravamo da li ima *nekih znakova infekcije*. Ispravnost stanja programa se praktično identifikuje sa

[49] Neispravnost u programski kod neposredno unosi programer. Ali to ne znači da je programer kriv, zato što, kao što smo već videli, to može biti posledica greške u projektovanju, ili čak u definisanju zahteva.

odsustvom infekcije, ili da budemo još precizniji – pretpostavka ispravnosti stanja programa je ekvivalentna sa odsustvom uočenih infekcija. Kao što ne znamo da li ima nekih skrivenih infekcija, tako ne znamo ni da li je stanje programa ispravno, već to možemo samo da pretpostavljamo.

14.1.3 Vrste propusta

U literaturi se sreću različite klasifikacije propusta. Jedna od uobičajenih deli propuste prema načinu ispoljavanja na:

- nekonzistentnosi u korisničkom interfejsu;
- neispunjena očekivanja;
- slabe performanse i
- padove softvera i oštećenja podataka.

Nekonzistentnosti u korisničkom interfejsu se odnose na različite vrste neusklađenosti između onoga što korisnik želi da uradi, načina na koji mora da izdaje svoje zahteve softveru, načina na koji softver izvršava zahtevanu operaciju i načina na koji softver izveštava korisnika o toku ili rezultatu operacije.

Na primer, većina programa za *MS Windows* koristi prečicu *Ctrl+F* za započinjanje operacije traženja. Međutim, neke verzije programa *MS Outlook* su tu prečicu koristile za prosleđivanje primljene elektronske poruke (engl. *forward*).

Propusti koji se manifestuju kao nekonzisentnosti u korisničkom interfejsu obično imaju uzroke u lošoj specifikaciji. Ona obično nastaje zbog loše urađene analize u fazi planiranja i projektovanja korisničkog interfejsa. Dobro stredstvo za prevenciju ove vrste problema jesu prototipovi. Na njima se u ranoj fazi mogu uočiti neki konceptualni problemi sa korisničkim interfejsom.

Neispunjena očekivanja

Dobijanje neočekivanog (tj. neispravnog) rezultata predstavlja verovatno najneugodniju vrstu propusta. Neispravni rezultati mogu da imaju različite oblike:

- neispravan numerički ili drugi rezultat – izračunata vrednost nije ispravna;
- neispravno reagovanje na akciju korisnika ili neispravan prikaz u korisničkom interfejsu;
- pogrešan pozitivan rezultat – dobijen je potvrdan odgovor umesto ispravnog odričnog odgovora i
- pogrešan negativan rezultat – dobijen je odričan odgovor umesto ispravnog potvrdnog odgovora.

U opštem slučaju, sve probleme ovog tipa možemo da svedemo na prvi oblik – svaki drugi oblik je samo drugačije posmatran rezultat izračunavanja. Suština je u tome da je specifikacijom programa definisano da će on nešto da izračuna, a u praksi ili to računa pogrešno ili računa nešto sasvim drugo.

Uzroci neispunjenih očekivanja se mogu naći u različitim fazama razvoja softvera, pa je njihovo pronalaženje i otklanjanje često veoma složeno. Propusti ovog tipa mogu nastati već u fazi definisanja zahteva, kada zbog propusta u komunikaciji može doći do nerazumevanja između klijenta i razvojnog tima. Ako se taj deo posla uradi kako valja, ova vrsta problema može da nastane i u skoro svakoj od faza analize i projektovanja softvera, u obliku propusta u projektovanju ili propusta u dokumentovanju određenih faza projekta. Na kraju, čak i kada je projektovanje urađeno u potpunosti ispravno, do problema može da dođe pri pisanju programa („kodiranju“) u vidu grešaka programera (neispravan odabir ili implementacija algoritma, neispravno čitanje ili tumačenje projektne dokumentacije,...).

Slabe performanse

Slabe performanse predstavljaju stalno ili povremeno sporo reagovanje programa na zahteve korisnika. Sporost je relativna i može se razmatrati samo u okvirima koji su određeni projektnim zahtevima ili uobičajenim potrebama korisnika. Iako same po sebi ne moraju da utiču na ispravnost konačnog rezultata, slabe performanse mogu da imaju presudan uticaj na upotrebljivost softvera. Ako redovna upotreba softvera podrazumeva interaktivan rad, već i umereno usporavanje može negativno da utiče na njegovu upotrebu, zbog učestalog primoravanja korisnika na čekanje.

Kada se razvija neki softver, obično se smatra da je primarni cilj da softver radi ispravno i daje ispravne rezultate, a da su performanse tek sekundarni cilj. Zaista, ako je izračunavanje neispravno, uopšte nije važno koliko je brzo izvršeno. Sa drueg strane, ako izračunavanje jeste ispravno, onda dužina izračunavanja može, ali ne mora, da utiče na uspešnost upotrebe softvera. Na primer, u mnogim slučajevima je potpuno nevažno da li će reakcija sofvera na zadatu komandu trajati 0.1ms ili 10ms, iako se ovde radi o veoma velikoj relativnoj razlici u brzini. Ipak, kada je trajanje odziva u opsegu vremenskih intervala koji odgovaraju iteraktivnom radu korisnika, onda je vreme odziva mnogo važnije, pa i mnogo manje relativne razlike mogu da igraju važnu ulogu. Na primer, razlika u trajanju izračunavanja od 200ms i 500ms se može prilično negativno odraziti na efikasnost upotrebe interaktivnog sistema za crtanje.

Kod softverskih sistema koji nisu razvijani za interaktivan rad, pojam slabih performansi ima nešto drugačije značenje. Na primer, ako izvršavanje nekih složenih izračunavanja zahteva nekoliko sati, tada nije toliko važno da li će izvršavanje biti završeno malo ranije ili kasnije, koliko može biti važno u kojoj meri su tokom

izračunavanja opterećeni resursi računarskog sistema, tj. da li je moguće tokom trajanja tog izračunavanja upotrebljavati računarski sistem za neke druge, manje zahtevne poslove.

Uzroci slabih performansi se nalaze u svim fazama razvoja softvera, od početnog procenjivanja opterećenja i količine raspoloživih resursa, do projektovanja i samog kodiranja. Dok se popusti u kodiranju u nekim slučajevima mogu otkloniti primenom bolje prilagođenog algoritma ili tehnike (npr. upotreba više niti), propusti koji su načinjeni u fazama procenjivanja i planiranja opterećenja se mnogo teže prevazilaze.

Padovi softvera i oštećenja podataka

U najopasnije vrste propusta spadaju oni koji se manifestuju različitim vidovima *padova softvera* ili oštećenja podataka. Pod padom softvera se podrazumevaju svi različiti slučajevi prekida rada programa, bilo da se radi o neplaniranom zatvaranju programa ili prestanku prijema komandi korisnika. Nisu retki slučajevi da pad jednog programa izaziva pad i nekih drugih programa ili čitavog operativnog sistema. Takvo indukovanje širokih padova najčešće je povezano sa velikim zauzećem sistemskih resursa od strane nestabilnog programa. Nisu svi operativni sistemi jednako otporni na ovu vrstu problema. Operativni sistemi predviđeni primarno za interaktivan rad korisnika (tzv. desktop operativni sistemi) po pravilu mnogo slabije izoluju i podnose ovakve probleme nego sistemi koji su specijalizovani za pružanje usluga (tzv. serverski operativni sistemi) ili sistemi opšte namene.

Oštećenja podataka su posebno neugodna. Za razliku od padova sistema, oštećenja podataka mogu da ostanu prikrivena, pa čak i da se veoma dugo dešavaju neprimećeno, a sa potencijalno veoma ozbiljnim posledicama. Ako se takve neispravnosti uoče tek posle dužeg vremenskog perioda, one mogu da proizvedu veoma ozbiljne posledice po ukupnu ispravnost podataka i ispravnost rada čitavog sistema, pa prevazilaženje takvih problema može da bude veoma složeneno, dugotrajno i skupo.

Karakteristika ovih propusta je da, osim u fazama projektovanja i programiranja, oni mogu nastati i u fazama povezivanja različitih komponenti složenih sistema. Bilo da je reč o neipravnoj primeni protokola povezivanja, upotrebi neispravnih protkola ili nekim drugim propustima, za većinu propusta koji nastaju u ovoj fazi je zajedničko da predstavljaju posledicu neispravne, nepotpune ili neispravno upotrebljavane dokumentacije.

14.1.4 Upravljanje propustima

Različiti razvojni timovi imaju različita iskustva sa propustima. U nekom timu se propusti češće prave, u nekom drugom se lakše pronalaze, a u nekom trećem se lakše otklanjaju. Ozbiljnost propusta takođe nije ujednačena. Ako analiziramo

različite timove, načine rada u njima, kao i način na koji svaki pojedinačan član tima pristupa svom poslu, onda možemo da uočimo mnogo razlika u različitim posmatranim oblastima. Razlike postoje u kadrovskoj politici (sastav timova, način odabira članova tima,...), načinu upravljanja timom (organizaciona struktura, podela odgovornosti,...), primenjenoj razvojnoj metodologiji (koncepcija rada, fokus, skup metoda i tehnika,...), konkretnim primenjenim alatima i drugim aspektima koji utiču na svakodnevni rad u timu. Čak i različiti članovi istog tima imaju najčešće sasvim različita iskustva u vezi sa greška i propustima. Neko greši manje a neko više, neko pravi ozbiljnije a neko bezbolnije propuste, neko ih pronalazi i ispravlja relativno lako, a nekome to predstavlja veliki napor.

Zbog toga postoji potreba da se rad sa greškama i propustima sistematizuje i unapređuje na nivou čitavog tima. Sve one aktivnosti koje za predmet bavljenja imaju propuste ili neke teme koje su povezane sa propustima, čine tzv. *upravljanje propustima*. Značaj upravljanja propustima je često presudan za kvalitet razvojnog procesa i konačnog proizvoda i predstavlja deo aktivnosti koje se šire nazivaju *obezbeđivanjem kvaliteta*.

Poslovi koji čine upravljanje propustima mogu da se, prema vremenu odvijanja, podele na dva velika skupa poslova – na prevenciju propusta i otklanjanje propusta. Prevencija propusta se zasniva na stalnom staranju o okolnostima koje imaju neposredan ili posredan uticaj na nastajanje propusta i njihovo kasnije otklanjanje. Odvija se tokom čitavog razvoja, od prvih koraka na definisanju i preciziranju ciljeva i zahteva, pa sve do puštanja sistema u rad. Prevencijom propusta ćemo se baviti u završnom delu ovog poglavlja. Otklanjanje propusta se odvija tokom čitavog perioda razvoja, ali i kasnije, tokom perioda održavanja softvera. Obuhvata sve aktivnosti koje imaju za cilj pronalaženje uzroka uočenih problema i otklanjanje pronađenih uzroka problema.

14.2 Otklanjanje grešaka

Otklanjanje grešaka iz programskog koda se među programerima uobičajno naziva *debagovanje* (engl. *debugging*). Doslovni prevod engleskog termina na srpski jezik bi mogao da bude *dezinsekcija* ili *uklanjanje buba*, ali su ipak uobičajeni termini *otklanjanje grešaka* i *debagovanje*.

Kao i svaki drugi složen posao, debagovanje je mnogo lakše ako mu se pristupi sistematično. Iako ponekad može da se stekne utisak da je neko „pravi talenat“, a neko „potpuni antitalenat“ za debagovanje, te da sam proces debagovanja u velikoj meri zavisi od intuicije i osećaja, stvari ipak stoje malo drugačije. Tačno je da lične sposobnosti mogu imati nezanemarljiv uticaj na kvalitet i efikasnost debagovanja, ali je značaj sistematičnog pristupa i obučenosti razvijalaca ipak daleko veći.

Osobe kojima debagovanje ide od ruke su obično one koje su (svesno ili nesvesno) usvojile i primenjuju osnovna pravila debagovanja.

Dejvid Agans

Debagovanje je veoma složen proces, koji od programera zahteva visok nivo stručnosti i dobro poznavanje problema, primenjenih algoritama, programskog jezika, svih povezanih delova sistema, a uz sve to još i sistematičnost, sposobnosti dobrog koncentrisanja svoje pažnje na problem, sagledavanja problema iz različitih uglova, kritičkog razmišljanja i sklonost kako analtičkom tako i konstruktivnom razmišljanju. Na sve to valja dodati i poznavanje pravila i tehnika debagovanja, kojima ćemo u ovom delu teksta posvetiti posebnu pažnju. Najvažnije aktivnosti u okviru debagovanja su:

- uočavanje da propust postoji;
- analiziranje i razumevanje propusta;
- lociranje propusta i
- ispravljanje propusta.

Uočavanje neispravnosti se najčešće dešava u okviru testiranja softvera, kao deo širih aktivnosti u oblasti prevencije propusta ili obezbeđivanja kvaliteta, ali se neispravnosti često uoče i tek pri upotrebi završenog i isporučenog softvera. Uočavanje neispravnosti u suštini nije predmet debagovanja, ali se tokom debagovanja elementi softvera veoma pažljivo posmatraju, pa se često dešava da se tokom rada na otklanjanju jednog problema uoče i evidentiraju i neki drugi problemi. Zato se uočavanje i evidentiranje propusta ponekad svrstava u aktivnosti u oviru debagovanja, a ponekad ne.

U idelanom slučaju, pri uočavanju propusta se on detaljno evidentira. Navode se tačne okolnosti koje su dovele do uočene neispravne manifestacije, što obuhvata tačnu verziju softvera, ulazne podatke, niz koraka koji je doveo do ispoljavanja problema, mesto i vreme i sve drugo što bi moglo da pomogne. Međutim, ako imamo u vidu da uočavanje problema često dolazi od strane osoba koje ne poznaju internu strukturu softvera i tokove podataka, već se prvenstveno bave pitanjima upotrebe i funkcionalnosti, veoma često će opis manifestacije problema biti nedovoljno detaljan.

Naredni korak, koji se često smatra i za prvi korak *pravog* debagovanja, čine analiziranje i razumevanje problema. Osnovni cilj analize problema je da se prikupi dovoljno informacija da bi problem mogao da se reprodukuje. Sve dok nismo u stanju da ponovimo problem, teško možemo da kažemo da razumemo kako on nastaje.

Da bi propust mogao da se razume, potrebno je da se poznaju šire okolnosti manifestovanja propusta, kao i struktura i povezanost podsistema u kome se propust manifestuje. Razumevanje propusta obuhvata povezivanje njegove manifestacije sa stvarnim dešavanjem u sistemu. Obično se pri uočavanju propusta dokumentovanje zadržava na informacijama koje se dobijaju kroz korisnički interfejs, dok je za puno razumevanje problema neophodno da se posmatraju i interni elementi sistema, koji učestvuju u širenju infekcije od mesta nastajanja do mesta ispoljavanja. Rezultat analiziranja i razumevanja propusta bi trebalo da bude skup pretpostavki o onome što se interno dešava u sistemu i dovodi do odgovarajuće manifestacije.

Lociranje propusta je postupak pronalaženja tačnog segmenta programskog koda (ili više takvih segmenata) u kome postoje greške koje za posledicu imaju opisan problem. Za uspešno lociranje propusta je neophodno njegovo razumevanje, kao i dobro poznavanje i razumevanje posmatranog programskog koda. Ako razumevanje propusta nije izvedeno dovoljno dobro, onda mnogo vremena može da se utroši na neuspešno traženje greške u delovima programskog koda koji su samo prividno povezani sa problemom.

Razumevanje propusta i lociranje konkretne greške u programskom kodu čine nerazdvojivu celinu, koja predstavlja najsloženiji i najteži deo procesa debagovanja. Analiza i razumevanje problema se bave delom puta infekcije koji je bliži tački ispoljavanja, dok se lociranje greške bavi delom puta infekcije koji je bliži uzroku. U nekim slučajevima, ako je put infekcije kratak, a mesto ispoljavanja vrlo blizu uzroku problema, može da se dogodi da se tokom analize i pokušaja razumevanja problema otkrije i tačnan uzrok, ali to je pre izuzetak nego pravilo.

Ta dva koraka debagovanja se veoma često odvijaju naizmenično (nakon pokušaja razumevanja sledi pokušaj lociranja, a u slučaju neuspeha se ponavlja pokušaj razumevanja, pa tako sve dok se greška ne locira) ili čak do te mere objedinjeno da imaju karakteristike jednog složenog koraka.

Kada se greška pronađe, njeno ispravljanje je najčešće daleko jednostavniji korak nego što je bilo razumevanje i lociranje. Težina ispravljanja propusta najviše zavisi od nivoa na kome je on načinjen. Greške u implementaciji programskog koda se ispravljaju relativno lako, ali prevazilaženje propusta pronađenih u poslovnim pravilima ili formalnim zahtevima nekada može da bude veoma zahtevno i skupo – čak i do te mere da se više isplati da se delovi programskog koda ponovo razvijaju.

Navedeni koraci debagovanja mogu da se izvode uz primenu odgovarajućih strategija ili različitih skupova principa i pravila. Prema tome kako se odvijaju, možemo da razlikujemo i različite pristupe debagovanju.

14.3 Neformalno debagovanje

Praktično svaki programer se, makar u početnom periodu svog programerskog stvaralaštva, oprobao u tzv. *neformalnom debagovanju*. Neformalno debagovanje predstavlja najjednostavniji, ali istovremeno i najpovršniji metod debagovanja. Počiva na pretpostavci da je propust jednostavan i da može brzo i lako da se pronađe i ispravi.

Osnovno sredstvo za rad su znanje o rešavanom problemu i napisanom programskom kodu i intuicija. Opis postupka čine samo dva koraka:

1. Pokušati sa sasvim jednostavnom[50] popravkom.
2. Sve dok program ne proradi, ponavljati korak 1.

Važno je da znamo da neformalnim putem može da se dođe do rešenja samo u sasvim ograničenom prostoru slučajeva. Jednostavne instant popravke mogu da se pronalaze „zatvorenih očiju" samo u slučajevima:

- jednostavnih malih programa;
- programa ili delova programa u kojima je prostor problema veoma uzak i lako sagledliv;
- programa ili delova programa koje piše jedan programer za najviše nekoliko dana;
- kada je domen problema izuzetno dobro poznat programeru.

U ovu grupu slučajeva u prvom redu spada rešavanje problema koji su uočeni već tokom pisanja koda. Tada programer vrlo dobro sagledava zadatak i kod koji piše, pa može neposredno i lako da ispravlja jednostavne previde u kodiranju. Manje sintaksne greške, zaboravljanje nekih promena podataka, pogrešno postavljanje granica blokova, neispravni indeksi i slične greške su primeri propusta koji se često mogu jednostavno rešiti. Ali nisu svi previdi koji se prave pri kodiranju *dovoljno* jednostavni. Neke greške nastaju zbog neispravnog koncepta rešenja ili pogrešnog zaključivanja. One uglavnom ne mogu da se reše primenom neformalnog metoda.

Kod naknadno ustanovljenih grešaka stvari obično stoje sasvim drugačije. U savremenom razvoju softvera se koriste testovi jedinica koda, kao i testovi ponašanja i prihvatljivosti, pa ako nijedan od tih testova ne prepozna problem, nego se on ispolji tek kasnije, pri manuelnom testiranju ili eksploataciji sistema, onda takav

[50] Ovde možemo dodati još neke atribute uz popravku, bilo da smo uvereni u sopstvene sposobnosti ili u jednostavnost problema, kao na primer: očigledna, originalna, genijalna i drugo.

problem najverovatnije nema jednostavan uzrok i zaslužuje ozbiljniji tretman nego što neformalni metod pruža.

Iako neformalan pristup u nekim slučajevima može da radi, on vlo često može da bude kontraproduktivan. Nipošto ne smemo da se zavaravamo da takvo debagovanje može da doprinese rešavanju svih problema. Naprotiv, ako neuspešna primena velikog broja pokušaja popravljanja nije praćena i potpunim brisanjem neuspelih „popravki", onda će nam po svoj prilici kod postati bogatiji za čitavo brdo novih problema. U svakom slučaju, ako rešavanje nekog problema neformalnim putem ne dovede do rešenja iz nekog manjeg broja pokušaja, obično je to znak da je vreme za ozbiljniji pristup problemu.

14.4 Empirijski naučni metod debagovanja

Razumevanje i lociranje propusta su analitičke aktivnosti. Počivaju na posmatranju velike količine informacija i izvođenju zaključaka na osnovu tog posmatranja. To se u osnovi ne razlikuje od istraživačkog pristupa u prirodnim (i nekim drugim) naukama. Uobičajen empirijski[51] naučni metod počiva na sledećem algoritmu:

1. Posmatrati prostor problema (ili nečije zapise o ranije obavljenom posmatranju);
2. Oblikovati (izmisliti) neprovereno tvrđenje (*hipoteza*) koje je u skladu sa rezultatima posmatranja;
3. Na osnovu hipoteze napraviti predviđanje ponašanja u nekim novim slučajevima;
4. Eksperimentalno (ili daljim posmatranjima) proveriti da li je predviđanje bilo ispravno;
5. Ponavljati korake 3 i 4, a po potrebi i 2, uz popravljanje hipoteze ili oblikovanje nove hipoteze, sve dok postoje neslaganja između predviđanja na osnovu hipoteze i odgovarajućih eksperimenata i posmatranja.

Opisan emprijski metod može da se primeni i na problem razumevanja i lociranja propusta u procesu debagovanja. Takav pristup se naziva *naučno debagovanje* ili *sistematsko debagovanje*:

1. Posmatrati uočen problem;

[51] Empirijski naučni metod je uobičajen za većinu prirodnih i društvenih nauka. Odlikuje se izvođenjem opštih zaključaka na osnovu pojedinačnih posmatranja.

2. Postaviti hipotezu o uzroku problema;
3. Na osnovu hipoteze napraviti predviđanje ponašanja sistema;
4. Eksperimentalno proveriti ispravnost predviđanja;
5. Ponavljati korake 3 i 4, a po potrebi i 2, uz popravljanje ili zamenjivanje hipoteze, sve dok se ne potvrdi ispravnost hipoteze ili ne iscrpe mogućnosti za njeno dalje unapređivanje.

Rezultat uspešne primene opisanog postupka bi trebalo da bude vrlo jasno i precizno definisana hipoteza. Jedan deo hipoteze mora da se odnosi na razumevanje uzroka problema, a drugi na tačnu lokaciju greške (ili grešaka) u kodu.

Primetimo da naučni metod zahteva da se pri debagovanju eksplicitno prave i zapisuju hipoteze. To je jedan veliki kvalitet, zato što implicitno rezonovanje o problemu ima tendenciju da stvara više teškoća nego koristi. U debagovanju je u igri ogromna količina raznovrsnih informacija i implicitno rukovanje njima vodi veoma krivudavim putem, samo u nove probleme. Jedna od posledica implicitnog pristupa je da je programeru teško da odgovara na pitanja svojih kolega o problemu. Kada se zaključci eksplicitno prepoznaju i evidentiraju kao uočene činjenice i hipoteze, umesto da postoje samo u neformalnom obliku kao zamisli i ideje, onda je rezonovanje na osnovu njih mnogo lakše i pouzdanije, a i lakše se o njima razmenjuju mišljenja.

Pri proveravanju važenja hipoteze, postoji određena fleksibilnost u pogledu mogućnosti menjanja sistema radi olakšavanja posmatranja i proveravanja. Međutim, pri tome mora dobro da se pazi da se neodgovarajućim ili preteranim izmenama ne dovede do *efekta posmatrača*. Kao i u prirodnim naukama, tako i pri debagovanju, svakim korakom koji načinimo prema detaljnijem i obuhvatnijem posmatranju, svakim zahvatom u programskom kodu ili primenom nekog spoljašnjeg alata, mi neminovno menjamo stanje sistema i okolnosti događaja koji posmatramo. Efektom posmatrača se nazivaju sve one posledice posmatranja koje dovode do netrivijalnih promena u ponašanju posmatranog sistema. Kao posledica načinjenih izmena može da nastupi promena u načinu manifestovanja problema, pa čak i potpuno maskiranje problema koji pokušavamo da posmatramo.

Da ne bi došlo do efekta posmatrača, moramo da imamo u vidu dinamičku prirodu problema, koju smo upoznali na početku ovog poglavlja. Sva naša posmatranja moramo da sprovodimo tako da ne remetimo prirodan tok ispoljavanja greške. U suprotnom može da se dogodi da nekim neopreznim zahvatima privremeno prekinemo ili promenimo način prenošenja ili ispoljavanja infekcije, a time i izmenimo ili sprečimo nastupanje neuspeha. To nije ništa drugo do ostvarivanje efekta posmatrača.

Najveći problem u primeni naučnog metoda je u tome što nam on daje neke opšte naznake kako i šta da radimo da bismo se približili rešavanju problema, ali nam ne

pomaže mnogo u konkretnim slučajevima. Najteži deo posla u praktičnom empirijskom debagovanju je *oblikovanje hipoteza*. Metod nas uči da to moramo da radimo, ali nam ne daje čak ni približne naznake kako bi to trebalo da radimo a da bude i dobro i što efikasnije.

14.5 Heurističko debagovanje

Heuristički metod debagovanja počiva na primenjivanju određenih pravila i principa, koji se oblikuju na osnovu ranijeg iskustva. Različiti autori predlažu različite skupove pravila, koji su uglavnom oblikovani prema njihovom konkretnom načinu razmišljanja. U nekim knjigama se navodi i po više različitih skupova pravila[52].

Kao jedan od praktičnijih i boljih skupova pravila može da se izdvoji onaj koji je predstavio Dejvid Agans [Agans2006]. Dalji tekst odeljka o heurističkom metodu debagovanja se u velikoj meri oslanja na pravila izložena u toj knjizi. Dejvid Agans ističe devet osnovnih pravila za debagovanje. Njegova pravila su dobro oblikovana i sistematično i dobro izložena. Pojedinačna pravila se najviše odnose na razumevanje i lociranje grešaka, ali celovit skup pravila pokriva i ostale aspekte debagovanja. Pravila su u osnovi jednostavna, što olakšava njihovo razumevanje, ali ne i primenu, koja u slučaju kompleksnih sistema može biti prilično teška.

Kada nam je trebalo mnogo vremena da pronađemo neku grešku,
to je bilo zato što smo zanemarili neko od osovnih pravila debagovanja
– jednom kada smo ga primenili, brzo smo pronašli problem.

Dejvid Agans

Njihova primena zahteva da se čitavom procesu pristupi veoma sistematično, uz redovno preispitivanje urađenog i istovremeno posvećivanje i detaljima i celini posmatranog problema. Posmatranje i formalno opisivanje problema na različitim nivoima astrakcije je upravo jedna od najvažnijih krakteristika procesa programiranja, pa tako i proces debagovanja mora da se odlikuje sličnim svojstvima, kako bi uspeo da obuhvati i prevaziđe sve potencijalne propuste koji su načinjeni pri kodiranju.

Pravila debagovanja su veoma jednostavno i razumljivo formulisana:

[52] Na primer, u [Metzger2004] se neka pravila navode u vidu heuristika, dok se neka druga pravila navode u vidu *taktika*. Iako su taktike uglavnom navedene u formi manjih postupaka koji se obavljaju u okviru šire oblikovanih heuristika, i tako definisana pravila imaju preklapanja sa skupovima pravila koje predstavljaju drugi autori. Pri kraju odeljka o heuristikama ćemo ukratko razmotriti i neke predložene taktike.

1. Razumeti sistem
2. Navesti sistem na grešku
3. Najpre posmatrati pa tek onda razmišljati
4. Podeli pa vladaj
5. Praviti samo jednu po jednu izmenu
6. Praviti i čuvati tragove izvršavanja
7. Proveravati naizgled trivijalne stvari
8. Zatražiti tuđe mišljenje
9. Ako je nismo ispravili, onda greška nije ispravljena

U narednim odeljcima ćemo prodiskutovati najvažnije aspekte ovih pravila i pokušati da objasnimo zašto su toliko važna za uspešno otklanjanje grešaka.

Pravilo 1 – Razumeti sistem

Da bi se neki problem prevazišao, potrebno je razumeti ga, pronaći gde je načinjen propust i otkloniti propust. Ali da bi problem mogao da se razume, najpre mora da se razume okruženje čiji je problem deo, a to je, najšire posmatrano, računarski sistem na kome se ispoljava posmatrani problem. Pojam računarskog sistema se u ovom kontekstu upotrebljava u svojoj najširoj definiciji, koja obuhvata razvijani softver, operativni sistem, sve ostale softverske komponente sistema (kako one sa kojima razvijani softver ostvaruje komunikaciju, tako i one koje naizgled sa njim nemaju ama baš nikakve veze), hardver računarskog sistema (uključujući sve hardverske uređaje, pa i one koji se naizgled ne upotrebljvaju), mrežne i distribuirane komponente sistema i korisnike.

Najčešće se pri debagovanju posmatranje sistema ograničava na najuži mogući deo – na razvijani softver ili neku njegovu pojedinačnu komponentu. Ovde moramo da istaknemo da je nesistematična primena takvog pristupa veoma loša i potencijalno kontraproduktivna. Iako u praksi, srećom, za rešavanje većine problema ne moramo da sagledavamo doslovno sve elemente posmatranog računarskog sistema, ograničavanje posmatranja ipak mora da se radi sistematično. Detaljnije objašnjenje ostavljamo za odeljak o pravilu *4 - Podeli pa vladaj*.

Razumevanje sistema počiva na poznavanju njegovih opštih i specifičnih karakteristika. Opšte karakteristike se odnose na uobičajene hardverske i softverske komponente sistema, zvaničnu dokumentaciju i formalno definisana pravila korišćenja sistema i razvijanog softvera. Specifične karakteristike obuhvataju sve elemente sistema koji se mogu manje ili više razlikovati između različitih sistema na kojima se razvijani softver koristi. Na primer, ako je softver razvijan za jedan konkretan operativni sistem, onda taj operativni sistem predstavlja opštu karakteristiku sistema, a jezik korisnika i druga regionalna podešavanja spadaju u specifične karakteristike sistema.

Uobičajeno je da se uputstva i tehnička dokumentacija ne čitaju u punom obimu, ili ako se to ipak čini, da se onda čitaju relativno površno. Kada se upoznajemo sa nekim sistemom, to je najčešće sasvim dovoljno, ili bar težimo da tako sebi predstavimo, zato što je puna dokumentacija za ozbiljne sisteme danas preobimna da bi neko mogao da je prouči od reči do reči pre nego što počne da ih koristi. Razvojna dokumentacija savremenih operativnih sistema, sistema za upravljanje bazama podataka i drugih složenih softverskih sistema može da obuhvata po nekoliko hiljada stranica prepunih detaljnih specifikacija načina upotrebe ili opisa ponašanja u različitim situacijama. Iluzorno je očekivati da neko može da drži u glavi sve informacije iz tolike dokumentacije. U krajnjem slučaju, dok bi početnik proučio svu tu dokumentaciju, vrlo verovatno bi bila objavljena neka novija verzija softvera sa novom i još obimnijom pripadajućom dokumentacijom.

Sa druge strane, kada dođe do nekog problema, čitanje dokumentacije je ipak nezamenljivo. Najveći deo dokumentacije upravo tome i služi. Načešće je dovoljno da se pri čitanju usmerimo na određene delove dokumentacije, posvećne nekim konkretnim pitanjima, koja su bliska problemu. Koliko god da je to teško i neprijatno, za neke problema se može prepoznati uzrok ili pronaći rešenje tek posle čitanje više različitih knjiga dokumentacije o različitim komponentama sistema (operativni sistem + sistem za upravljanje bazama podataka + razvojni alat + neke biblioteke + ...). Posebno je važno da se odgovarajući delovi dokumentacije prouče detaljno, tako da se ne propuste neke „sitnice“ koje mogu da značajno utiču na ponašanje sistema i razvijanog softvera.

Poseban problem sa dokumentacijom je da ona često ima tendenciju da ne bude ažurna. Uobičajena je praksa da se dokumentacija pravi paralelno sa izradom softvera ili da blago kaska za izradom softvera. Tako je veoma čest slučaj da se aktuelna javno dostupna verzija dokumentacije zapravo odnosi na prethodnu verziju softvera. U takvim slučajevima obično postoji tzv. „beta“ verzija dokumentacije, koja se odnosi na aktuelnu verziju softvera, ali nije do kraja „ispeglana“, pa se u njoj mogu naći neke „omaške“. Čak i kada je dokumentacija deklarisana kao ažurna, ona često ima različite vrste slabosti: objašnjenja nekih novih elemenata softvera ili novih oblika ponašanja softvera nisu dovoljno detaljna ili sasvim nedostaju, postojeća objašnjenja nisu usklađena sa poslednjim izmenama usoftveru ili čak postoje neki viškovi, koji opisuju elemente softvera koji su izbačeni u odnosu na prethodnu verziju ili iz nekog razloga ipak nisu ušli u aktuelnu verziju iako su bili planirani.

Sastavni deo razumevanja sistema je i poznavanje očekivanog ponašanja razvijanog softvera. Da bi se mogao uočiti, analizirati i razumeti neki problem, neophodno je znati kako bi sistem trebalo da se ponaša u idealnim uslovima. Nije retkost da se prijavi postojanje nekog problema i da se na njegovu analizu utroši mnogo vremena, a da se na kraju ispostavi da je u pitanju ponašanje koje je u

potpunosti u skladu sa planiranim ponašanjem. Neke stvari mogu neupućenom korisniku izgledati neintuitivno, čudno, prekomplikovano ili čak pogrešno, a da u širem kontekstu predstavljaju dobro zamišljeno i implementirano rešenje. U takvim slučajevima se mnogo vremena u fazama testiranja i analize prijavljenih nedostataka može uštedeti poznavanjem očekivanog ponašanja sistema.

Savremeni razvoj softvera počiva na upotrebi različitih softverskih i hardverskih komponenti i njihovoj međusobnoj komunikaciji i saradnji. U takvim okolnostima poznavanje ponašanja razvijanog softvera obuhvata i poznavanje tokova podataka između različitih podistema, a često i poznavanje formata tih podataka. Potrebno je znati kako je organizovana raspodela odgovornosti između komponenti sistema – koja komponenta je zadužena za koje aktivnosti, šta su ulazne informacije na osnovu kojih ona obavlja svoj posao, koje izlazne informacije proizvodi i kojim putem ih izdaje. U razumevanju ovog segmenta sistema mogu da budu od velike pomoći dobro napravljeni dijagrami. Tu je danas nezamenljiva uloga *UML*-a, kao široko prihvaćenog jezika za modeliranje softvera.

Sve što je rečeno u vezi sa čitanjem dokumentacije u vezi sa komponentama sistema, odnosi se i na razvojne alate. Razvojni alati su svi oni programi koji članovim razvojnog tima pomažu u proizvodnji softvera. Uobičajeno je da se u razvojne alate svrstavaju prevodioci, editor teksta, debageri i/ili integrisani razvojni alati (engl. *Integrated Development Environment – IDE*), ali tu spadaju i mnogi drugi alati, kao sistemi za praćenje verzija, sistemi za praćenje poslova i problema, alati za rad sa bazama podataka i drugo. Štaviše, u savremenom razvoju softvera je uobičajeno da se za pisanje različitih komponenti sistema koriste i različiti programski jezici, pa i različiti sistemi za upravljanje bazama podataka, a nije retko ni da se različite komponente prilagođavaju različitim operativnim sistemima.

Dokumentacija služi da se čita i to je potrebno raditi što je ranije moguće. Ako smo iz bilo kog razloga propustili čitanje dokumentacije već pri započinjanju pisanja softvera, onda se njome moramo pozabaviti već na početku debagovanja.

Mnoge greške u kodiranju su posledica površinskih sličnosti između razvojnih alata. Iako se može činiti da su neki elementi različitih programskih jezika međusobno ekvivalentni, njihova slična sintaksa može da prikrije sasvim različitu semantiku, što je idealna podloga za greške. Slično važi i za druge vrste razvojnih alata, pa je dobro poznavanje svih alata i jezika koji se koriste u razvoju veoma važno kao preventiva pravljenja grešaka. A kada je greška već načinjena, za njeno razumevanje i lociranje je neophodno da se poznaje jezik progamskog koda, kao i specifičnosti alata koje eventualno mogu da utiču na ponašanje.

Savremeni softverski sistemi su veoma složeni, što značajno otežava njihovo razumevanje. No, niko nije rekao da je programerski posao lak – a ako želimo da ga radimo dobro, tj. da pišemo programe sa malo grešaka i da te greške dobro i efikasno otklanjamo, neizbežno smo prinuđeni da poznajemo različite aspekte sistema koje

pravimo i računarskih sistema u okviru kojih će oni da se koriste. Pri tome nije dovoljno da se zadržimo na površnom upoznavanju, već moramo da se bavimo i detaljima.

Neophodna pretpostavka za dobro razumevanje računarskog sistema je dovoljno široko i temeljno poznavanje teorijskih i praktičnih osnova računarstva. Iskustvo pokazuje da u ovom kontekstu „dovoljno“ nikada nije dovoljno.

Pravilo 2 – Navesti sistem na grešku

Da bi neki problem mogao da se reši, najpre je potrebno da bude pažljivo posmatran i analiziran. U slučaju propusta u razvoju softvera, teškoća je što izvršavanje programa nije jedna stabilna statička slika koja se može proizvoljno dugo posmatrati, već dinamički proces čiji tok nije uvek jednostavno predvideti. U tom kontekstu je posmatranje nekog problema prilično otežano. Zbog toga je veoma važno da se pronađe tačan redosled koraka posle koga se ispoljava uočen problem. Ako problem može da se ponovi, onda on može lakše da se posmatra iz više uglova i u različitim okolnostima, što je neophodno za njegovo razumevanje.

Omogućavanje posmatranja problema je jedan od najvažnijih razloga za ponavljanje greške, ali ne i jedini. Drugi važan razlog za ponavljanje greške jeste omogućavanje posmatranja uzroka njenog ispoljavanja. Uzrok problema je obično povezan sa nekim od koraka koji su doveli do manifestovanja greške. Ako ne znamo koji su to koraci, onda smo veoma daleko od razumevanja problema i ustanovljavanja uzroka. Tek kada poznajemo mehanizam za ponavljanje greške, možemo da razmatramo i različite potencjalne uzroke i da pristupimo postepenoj lokalizaciji problema.

Postoji i treći razlog za određivanje niza koraka koji dovode do greške – provera da li je greška otklonjena ili ne. Ako znamo kako da ponovimo grešku, onda imamo relativno jednostavno sredstvo za proveravanje da li je ispravljanje greške bilo uspešno – nakon sprovođenja pokušaja njenog otklanjanja, možemo da pokušamo da ponovimo grešku i posmatramo ishod. Ako greška više ne može da se ponovi, onda možemo da smatramo da je otklonjena, a ako se i dalje ponavlja, onda nije.

Uobičajen metod za ponavljanje manifestovanja greške je isprobavanje uz dokumentovanje različitih nizova koraka i rezultata. I ovde vidimo značaj sistematičnosti pri debagovanju. U najboljem slučaju, kada neko po prvi put uoči problem, znaće kako je do toga došlo i dokumentovaće sasvim precizno odgovarajući niz koraka. Naravno, to nije uvek tako. Vrlo često je specifikacija uočenog problema nedovoljno dobra da bi se on mogao ponoviti, pa je potrebno da se eksperimentiše sa različitim aktivnostima, koje se ne udaljavaju previše od navedenog niza koraka. Kada god mora da se eksperimentiše, neophodno je da dokumentujemo sve pokušaje i ishode, zato što u suprotnom vrlo lako može da se

dogodi da se isti pokušaji ponavljaju po više puta i dragoceno radno vreme troši uzalud.

Jednom kada ponavljanje problema uspe, ne bi trebalo da nas iznenadi ako na nekom drugom sistemu, ili na istom sistemu ali u drugim okolnosima, ista sekvenca koraka ne dovodi do ispoljavanja problema. Za uspešno ponavljanje greške često nije dovoljno dokumentovati samo niz koraka koji do nje dovode nego i detaljan opis uslova, na primer u obliku niza koraka koji pripremaju za rad čitavo okruženje. Na ponavljanje greške mogu da utiču neke očekivane stvari, kao npr. različiti podaci koji se obrađuju, postojeći sadržaj baze podataka i sl. ali i neke naizgled sasvim nebitne stvari kao npr. kodni raspored tastature, vreme i datum na računarskom sistemu, vremenska zona, da li je korisnik levoruk ili desnoruk, da li istovremeno na istom računaru radi još neki program i drugo.

Neke probleme je potrebno da ponovimo mnogo puta da bi se izvršilo dovoljno različitih analiza i ustanovio uzrok. Ako je postupak ponavljanja greške složen i/ili dugotrajan, a mora da se ponovi više puta, onda može biti korisno da se napravi skript koji simulira aktivnosti korisnika i automatizuje ponavljanje greške. Simuliranje omogućava i da se programski simulira ponašanje sistema za različite vrednosti parametara, pa se može upotrebiti kao značajna pomoć pri lokalizovanju greške. Grafički korisnički interfejsi savremenih operativnih sistema uglavnom počivaju na arhitekturi upravljanoj događajima, tako da se simulacija aktivnosti korisnika svodi da simuliranje događaja koje korisnik proizvodi, tj. simulaciju upotrebe miša i tastature. Elementarni alati za simuliranje aktivnosti korisnika su sadržani u nekim distribucijama operativnih sistema. Obično su koncipirani tako da omogućavaju snimanje aktivnosti korisnika i njihovo kasnije ponavljanje. Naprednije verzije ovih alata omogućavaju i manuelno održavanje sačuvanih zapisa aktivnosti, pa i upotrebu kontrolnih struktura, tj. pune programske mogućnosti.

Ako se pri debagovanju naprave neki pomoćni alati, čija je namena da omoguće ponavljanje greške, takve alate nakon otklanjanja konkretne greške nije dobro odbacivati. Preporučljivo je da se sačuvaju, zato što se lako može dogoditi da nam ponovo zatrebaju. U prirodi grešaka je da su reko usamljene. Greške nastaju u trenucima umora ili smanjene pažnje programera, pa ako u nekom delu koda imamo grešku, to znači da je taj deo koda pisan (ili menjan) kada je programer bio u „nepažljivoj fazi“, a to opet znači da je sasvim moguće da su u istom delu koda načinjene i još neke greške. Čuvanje pomoćnih alata za ponavljanje grešaka nam može uštedeti mnogo vremena u slučajevima naknadog ispoljavanja „srodnih“ problema.

Ako smo u nekom trenutku uspeli da rekonstruišemo niz koraka koji dovode do greške, tu ne treba stati. Poželjno je da razmotrimo mogućnost da se do istog problema može doći i na neki drugi, ali relativno sličan način. Posmatranje je potrebno da se usmeri na razliličite srodne postupke, koji služe za izvođenje iste ili

sličnih operacija i da se poveri da li je istu ili sličnu manifestaciju moguće proizvesti i u delimično izmenjenim okolnostima. Primarna motivacija za ovakvu analizu je u tome što vezivanjem za sasvim specifičan niz koraka možemo da se usmerimo na otklanjanje samo jednog dela propusta, pa da neispravljeni ostatak nastavi da se manifestuje u izmenjenim okolnostima. Suviše precizno posmatranje može da nas navede da otklonimo samo simptom, a ne i uzrok problema. Alternativni postupci za ponavljanje greške mogu da nam prošire pogled na grešku i njene uzroke, ali i da omoguće pouzdaniju proveru nakon otklanjanja.

Pri traženju alternativnih postupaka za ponavljanje greške ne treba ići predaleko. Ako se neka dva niza koraka do iste greške veoma značajno razlikuju, onda vrlo verovatno uopšte nije reč o dva puta do ispoljavanja istog problema, već o dve potpuno različite greške sa sličnim manifestacijama. Naravno, nije loše da pri popravljanju jednog problema proverimo i što više srodnog koda, ali, na žalost, to nije uvek prihvatljivo iz poslovnog ugla i to iz bar dva razloga: najpre zato što može da bude veoma važno da se konkretan problem otkloni u što kraćem periodu, a zatim i stoga što redovnim proveravanjem široke okoline grešaka možemo doći do toga da neke delove koda nepotrebno poveravamo po više puta.

Traženje više alternativnih nizova koraka koji dovode do greške može dalje da se uopšti ponavljanjem (ili simuliranjem) ne same greške nego uslova u kojima se ona ispoljava.

Neke greške je teško ponoviti. Za to može biti više razloga, od složenih okolnosti u kojima se pojavljuje, do zahtevane visoke preciznosti specifičnog niza koraka. Složene okolnosti podrazumevaju veliki broj razlčitih elemenata koji moraju da se usklade da bi se greška manifestovala. Visoka preciznost je, na primer, ako se problem ispoljava samo u slučajevima kada se pokazivač miša pomeri sa jedne tačno određene pozicije na neku drugu tačno određenu poziciju.

Veoma je važno da se neuspešno ponavljanje greške ne tumači kao njeno odsustvo. Moramo imati određeno poverenje u osobu koja je prijavila postojanje problema – zašto bi neko gubio vreme da prijavljuje problem koji ne postoji? Sasvim je moguće da prijavljene okolnosi nisu dovoljno tačne, da možda i samo ispoljavanje nije dovoljno dobro opisano, ili da je u pitanju ispravno ponašanje koje je korisniku izgledalo kao greška, ali činjenicu da problem postoji ne smemo olako da dovodimo u pitanje. Najlakše je da na prijavljenu grešku odgovorimo komentarom da je „opisan problem nemoguć“, ali moramo da imamo na umu da greška koju nismo otklonili nije otklonjena (videti pravilo 9).

Ako smo prepoznali niz koraka koji dovodi do greške, ali samo u nekim slučajevima, onda to znači da nam okolnosti ispoljavanja greške nisu do kraja poznate. Najbolji put je da se daljim analiziranjem okolnosti i eksperimentisanjem popravi niz koraka tako da se učestalost ispoljavanja greške poveća. Ipak, neke greške će se uporno pojavljivati samo povremeno, npr. jedanput u 10, 100 ili čak i

više hiljada ponavljanja niza koraka. Da bi se ponovile takve greške, potrebno je da se razmotre svi oni uslovi koji do tada nisu kontrolisani. To može da obuhvata praćenje i podešavanje:

- neinicijalizovanih podataka;
- slučajno generisanih podataka;
- ulaznih podataka;
- trenutka ili trajanja izvršavanja;
- međusobne sinhronizacije niti i procesa i
- spoljašnjih uređaja.

U najvećem broju slučajeva će određivanje fiksiranog ponašanja za neki od ranije nekontrolisanih parametara uticati na učestalost ponavljanja greške. Ako se učestalost poveća, to znači da smo se približili tačnim uslovima. Ako se smanji, to znači da smo se udaljili, ali da uslov koji je kontrolisan ipak utiče na pojavljivanje greške. Tada je potrebno da nastavimo kontrolisanje istog uslova, ali sa nekim drugim vrednostima.

Nekada svi pokušaji uvođenja dodatne kontrole uslova imaju za rezultat smanjivanje učestlosti ponavljanja problema. To su posebno neugodni slučajevi, zato što ukazuju na to da se greška ispoljava samo u vrlo strogo definisanim uslovima, te da je takve uslove veoma teško ponoviti. Primarni cilj u takvim slučajevima jeste da se prepozna koji su to parametri koji utiču na ponavljanje greške, a tek sekundarni cilj je da se ustanove i odgovarajuće vrednosti tih parametara. Pri prepoznavanju značajnih parametara može da bude od pomoći upravo suprotan metod od do sada pominjanog – povećavanje stepena slučajnosti vrednosti parametra. Time što neki parametar konfigurišemo da pri svakom ponavljanju uzima neku drugu slučajnu (tj. pseudo-slučajnu) vrednost i posmatramo da li se i u kojim uslovima greška ponavlja, možemo da postepeno dođemo do skupa prametara čije menjanje utiče na ispoljavanje problema. Takav postupak nije ni jednostavan ni brz, ali ako uobičajen analitički pristup ne dovede do rezultata, onda nam je to jedna od najprihvatljivijih alternativa. Problem sa ovakvim pristupom je što postavljanje slučajnih vrednosti parametara može da proizvede neke druge greške, što ima za posledicu brojne potencijalne stranputice tokom rešavanja jednog konkretnog problema. Od velike je važnosti da se slučajne vrednosti parametara primenjuju uz primenu još jednog pravila debagovanja – *6. Praviti i čuvati tragove izvršavanja*“. Bez toga je praktično nemoguće analizirati ponašanje sistema u brojnim ponovljenim izvršavanjima sa različitim vrednostima parametara.

Ako uradimo sve opisano, a problem se i dalje ponavlja samo povremeno, onda je potrebno promeniti redosled aktivnosti. Najpre, ne smemo da zapadnemo u apatiju zato što je problem „svojeglav“. Problem nikada nije svojeglav. Uvek se ispoljava u

nekim sasvim preciznim ulovima, a to što ne umemo da ih ponovimo je samo otežavajući faktor. U takvim slučajevima možemo da odustanemo od striktnog ponavljanja problema i da se neposredno posvetimo upravo ciljevima ponavljanja. Ciljevi ponavljanja su, podsetimo se, omogućavanje posmatranja problema i njegovih uzroka, a kasnije i provera da li je problem otklonjen. Za proveru nam može poslužiti čak i procedura koja samo povremeno proizvodi problem – ako se problem pojavljuje jedanput u 1000 izvršavanja, možemo da smatramo da smo ga otklonili sa određenom verovatnoćom, ako se ne pojavi nijedanput u npr. 50.000 izvršavanja. Takav metod proveravanja nije savršen, ali je često prihvatljiv.

Što je neki problem teže ponoviti, to je važnije da ga pažljivije posmatramo onda kada se ponovi. Potrebno je da prikupljamo sve relevantne informacije i da njihovim analiziranjem pokušamo da ustanovimo koji su to uzroci problema. Da bi to bilo moguće, često je potrebna intenzivna primena pravila *6 – Praviti i čuvati tragove izvršavanja*.

Pravilo 3 – Najpre posmatrati pa tek onda razmišljati

Videli smo da je problem potrebno ponoviti da bismo ga mogli posmatrati. Posmatranje je neophodno radi prikupljanja što veće količine informacija o ispoljavanju problema i uslovima njegovog ispoljavanja. Informacije su nam neophodne zato što samo uz dovoljno informacija možemo razmišljanjem da dođemo do dobrog zaključka. Razmišljanje kome bismo pristupili bez relevantnih informacija nam ne bi bilo od velike pomoći, već bi moglo da nas zatrpa neispravnim ili beskorisnim zaključcima. Pravilo „najpre posmatrati pa tek onda razmišljati“ ima za cilj da nam uštedi vreme na razmatranju i implementiranju pogrešnih zaklučaka.

Iskusni programeri mogu u nekim slučajevima da na osnovu stečenog iskustva donose neke zaključke i sa mnogo manje raspoloživih informacija nego početnici. To je dobro – uostalom, upravo tome i služi iskustvo. Međutim, za dobre zaključke „mnogo manje“ ne sme biti i „nedovoljno“. Ne smemo da se zalećemo i da nudimo rešenje pre nego što sagledamo okolnosti. Ne smemo da donosimo zaključke pre nego što se greška ponovi. Izuzetno, ako imamo veliko poverenje u onoga ko je prijavio problem i u njegov izveštaj, onda informacije sadržane u tom izveštaju mogu biti dovoljne, ali takve prečice smeju da se prave samo u iskusnim timovima u kojima se članovi tima odlično sporazumevaju. Čak i tada je preporučljivo da se problem ponovi pre njegovog rešavanja.

Pri posmatranju moraju da se uzimaju u obzir i detalji i celina. Svaka pojedinačna informacija može da nam ukaže na potencijalan uzrok ili neki element mehanizma koji dovodi do problema. Sa druge strane, međusoban odnos uočenih detalja je često mnogo rečitiji nego svaki detalj za sebe.

Da bi posmatranje moglo da se obavlja dovoljno dobro, u mnogim razvojnim projektima se već u fazi projektovanja softvera isplanira obezbeđivanje alata za

pomoć pri posmatranju. Svi alati za posmatranje se dele na spoljašnje i unutrašnje. Spoljašnji alati su različite vrste softverskih debagera, ali i hardverski alati koji, na primer, omogućavaju da se posmatrani softver izvršava na jednom, a alati za posmatranje na drugom računarskom sistemu ili uređaju. Savremeni spoljašnji alati raspolažu solidnim mogućnostima i veoma su korisni u svakodnevnom debagovanju. Značajan broj grešaka se može razumeti i locirati njihovom upotrebom. Međutim, nisu savršeni. Za veliki broj problema je neophodno razvijanje i korišćenje unutrašnjih alata.

Unutrašnji alati su oni koji se ugrađuju u programski kod softvera koji se razvija i debaguje. Takvi alati mogu da obuhvataju mehanizme za pravljenje tragova (videti pravilo 6), ali i procedure za zapisivanje podataka u fajlove u nekom čitljivom obliku. Danas se za zapisivanje složenih struktura podataka često upotrebljaju *XML*, *JSON* i druge notacije, koje mogu da dobro predstve strukturu i međusobni odnos podataka, a istovremeno se u osnovi radi o tekstualnim formatima koji su čitljivi čoveku. Uobičajeno je da se tragovi izvršavanja prave tako da se olakša njihova automatska analiza, pa se zato u njih zapisuju i obaveštenja, parametri, vrednosti promenljivih i druge potencijalno korisne informacije. Unutrašnjim alatima će biti posvećeno nešto više pažnje kasnije u ovom poglavlju.

Pri posmatranju grešaka često se primenjuje privremeno isključivanje delova koda. Cilj isključivanja je da se istovremeno suzi prostor oko greške (pravilo 4), skrati vreme potrebnog za ponavljanje i posmatranje greške i smanji količina informacija koju je potrebno anlizirati.

Jedna od opasnosti pri upotrebi unutrašnjih i spoljašnjih alata je softverska verzija efekta posmatrača: „Svako posmatranje menja sistem, zato što su i posmatranja sastavni deo sistema“. Čak i sasvim sitne promene programskog koda, koje se uvode radi olakšavanja posmatranja, mogu da imaju uticaja na ponašanje sistema, a time i na uslove i načine ispoljavanja posmatranih grešaka. Ovo ne znači da kod ne treba aktivno posmatrati, već da je potrebno da se to čini veoma pažljivo. Preporučljivo je da se programski kod menja radi posmatranja samo toliko koliko je neophodno da se uoče posmatrani elementi, ali ne i više od toga. Pravljenjem suvišnih izmena se nepotrebno povećava verovatnoća menjanja ponašanja sistema, a bez značajnih pozitivnih posledica.

Put do pouzdanih zaključaka o uzroku greške vodi preko pretpostavki (hipoteza). Pretpostavke se donose kako bi se dalje lokalizovali uzroci ispoljenih problema. Neproverene pretpostavke se ne smeju koristiti kao osnova za otklanjanje grešaka –neproverene hipoteze su ništa drugo do nagađanje. Svi zaključci moraju (1) da počivaju na dovoljnim i ispravnim informacijama i (2) da budu nedvosmisleno potvrđeni.

Pravilo 4 – Podeli pa vladaj

Osnovna ideja pravila „podeli pa vladaj“ je da se postepenim aproksimacijama sužava oblast traženja greške. Najpre se postavljaju hipoteze o lokaciji greške (naravno, u skladu sa pravilom 3). Takve hipoteze ne bi trebalo da pokušavaju da sasvim precizno lociraju grešku, već da podele posmatranu oblast na dve podoblasti približnih veličina. Zatim se testovima ustanovljava tačnost ili netačnost hipoteze i u skladu sa rezultatima testiranja se sužava oblast posmatranja. Ponavljanjem ovih koraka se oblast postepeno sve više sužava, da bi se, u idealnom slučaju, došlo do sasvim precizno lociranog uzroka greške.

Motivacija za ovakav pristup potiče iz realnih ograničenja posmatrača i velikog obima i složenosti kompletnog sistema. Iako bi bilo idealno da se uvek posmatra ceo sistem, to u praksi uglavnom nije moguće. Ako bi neko čak mogao da isprati sva dešavanja u složenom sistemu, to bi onda sigurno bilo isuviše sporo. Znači, već pri posmatranju greške postoji potreba da se posmatranje usmeri na one delove sistema, koji povezuju uzrok problema i uočene posledice.

Jedna od najčešćih grešaka u primeni ovog pravila se pravi već na samom početku, izborom nedovoljno širokih okvira polazne posmatrane oblasti. Od presudne važnosti za uspešnu primenu pravila „podeli pa vladaj“ jeste da svi uzroci problema budu u okviru polazne posmatrane oblasti. Ako to nije slučaj, onda se nikakvim hipotezama i smanjivanjima ne mogu u usuženoj oblasti naći svi uzroci problema, pa i čitava primena pravila može da ispadne samo protraćeno vreme.

Nije uvek jednostavno nedvosmisleno utvrditi da je posmatrana oblast dovoljno široka, tj. da obuhvata sve uzroke. To je posebno teško u slučajevima kada se posmatra mrežno ili distribuirano okruženje. Neophodan preduslov za pravljenje dobrih početnih procena je dobro poznavanje sistema (pravilo 1).

Pri izboru hipoteza potrebno je voditi računa o veličini potprostora na koje deli posmatrani prostor, ali i o tome da se hipotezom nedvosmisleno ukazuje na jedan od tih potprostora kao na lokaciju greške. Ako jedan od ova dva uslova nije zadovoljen, tada je i hipoteza verovatno beskorisna.

Jedan način za postavljanje hipoteza je da se pođe od greške prema potencijalnim uzrocima. Mesto ispoljavanja greške je obično relativno poznato. Pretpostavljanjem da je uzrok ne dalje od nekih hipotezom određenih okvira omogućava se postepeno sužavanje posmatrane oblasti bez posvećivanja pažnje svakom pojedinačnom potencijalnom uzroku. Mesta ispoljavanja problema obično ima mnogo manje nego potencijalnih uzroka. Zbog toga je kretanje od greške prema uzrocima najčešće efikasnije od kretanja u suprotnom smeru i postepenog odbacivanja jednog po jednog potencijalnog uzroka. Od presudne važnosti je da se počne od *svih* oblika ispoljavanja problema. Ako se zanemarivi neki od aspekata ispoljavanja problema, onda posledica može da bude „rešavanje“ nekih simptoma, a ne njihovog uzroka.

Pri traženju uzroka mogu postojati problemi u vidu šumova. Šumovi su različiti dinamički delovi sistema, koji svojim promenljivim ponašanjem mogu da prikriju stvarne uzroke problema, ili navedu posmatrača da traži uzroke u pogrešnom delu sistema. Poseban problem je što jedna greška može biti uzrok druge greške. Zbog toga bi i posle pronalaženja i lociranja greške trebalo nastaviti dalje sužavanje posmatrane oblasti, kako bi se proverilo da ne postoji još neki uzrok problema.

Šumovi se mogu otkloniti privremenim isključivanjem delova koda za koje se pretpostavlja da ne sadrže uzrok greške. U duhu primene ovog pravila, isključivanje dela koda predstavlja upravo vid testa odgovarajuće hipoteza o lokaciji greške.

Neke vrste grešaka se teško uočavaju na „živim" podacima. Zbog toga se često prave veštački uzorci ulaznih podataka (ili širih uslova izvršavanja programa), za koje je poznat ispravan rezultat. Veštački uzorci mogu da olakšaju proveru nekih hipoteza, kao i da dopuste formulisanje nešto slobodnijih hipoteza, kakve ne bi imale smisla sa realnim podacima.

Pored šumova i teškoća sa živim podacima, najozbiljniji problem sa kojim se suočavamo pri sužavanju oblasti posmatranja su slučajevi sa više uzroka koji se nalaze u raličitim segmentima sistema. Tada se sužavanje na uobičajen način obično završava na relativno širokoj oblasti koja obuhvata sve uzroke i još poneki dodatni element softvera. Kao dodatni elementi su obično obuhvaćene komponente koje povezuju delove softvera koji sadrže uzroke, ili su u relativno složenim odnosima sa takvim delovima.

Problem širokih oblasti, koje obuhvataju više uzroka, se može prevazići pomoću ciljano oblikovanih veštačkih uzoraka ulaznih podataka. Biranjem ulaznih podataka sa različitih granica opsega se obično može neutralisati uticaj nekih uzroka, pa time i omogućiti dalje sužavanje posmatrane oblasti. Biranjem različitih veštačkih uzoraka obično mogu da se lokalizuju neki od uzroka problema. Po otklanjanju uočenih uzroka, izborom drugačijih ulaznih podataka, mogu se potražiti drugi uzroci i tako nastaviti rešavanje problema.

Pravilo 5 – Praviti samo jednu po jednu izmenu

Svaki put kada menjamo postojeći kod, moramo da imamo u vidu da naknadne izmene predstavljaju plodnije okruženje za nastajanje grešaka nego što je to slučaj sa pisanjem sasvim novog koda. Razlika je u tome što kada pišemo novi kod, onda obično imamo široku sliku o tome šta se zbog čega piše i zašto se nešto piše na konkretan izabran način, dok pri menjanju postojećeg koda postoji povećana opasnost da neki deo konteksta, u kome se nalazi ili funkcioniše programski kod koji se menja, nije u potpunosti shvaćen ili čak uopšte nije ni uzet u razmatranje.

Naravno, menjanje postojećeg koda je neophodno. Štaviše, debagovanje podrazumeva učestalo menjanje postojećeg programskog koda. U fazi lokalizacije greške kod se menja da bi se privremeno isljučilo funkcionisanje nekog dela koda, ili

da bi se proverila ispravnost hipoteze. U fazi otklanjanja greške kod se menja da bi se otklonili lokalizovani uzroci greške. U uslovima kada moramo da pravimo izmene, a one predstavljaju opasnost u vidu potencijalnog pravljenja novih grešaka, potrebno je da sagledamo načine za što bezbednije pravljenje izmena. U tome je osnovni smisao pravila „Praviti samo jednu po jednu izmenu". Izmene nikada ne bi trebalo da se prave u paketu, već samo pojedinačno[53], da bi se na taj način olakšalo uočavanje eventualno napravljenih propusta.

Među potencijalne probleme pri pravljenju paketa izmena spada mogućnost da jedna izmena reši problem, a da je druga suvišna. U tom slučaju, druga izmena uopšte nije poželjna, zato što može da napravi nepotrebne nove probleme. Pri menjanju koda je potrebno da se uvek zadržimo na najmanjem mogućem broju izmena, te da pravimo samo one izmene koje su neophodne za rešavanje problema. Tim pre se te izmene mogu i moraju praviti svaka za sebe, jedna po jedna, uz učestalo proveravanje njihovih posledica.

Može da se dogodi i da nijedna od načinjenih izmena nije dobra. Naravno, to možemo da ustanovimo proveravanjem, ali će biti mnogo teže da se pokaže da načinjene izmene nisu dobre nego kada bi se to radilo sa jednom po jednom izmenom. Nije retkost da se načinjenim izmenama istovremeno rešava jedan i pravi neki drugi novi problem. I u tom slučaju je uočavanje neispravnosti i njihovo povezivanje sa konkretnim izmenama jednostavnije i pouzdanije ako se izmene prave postepeno, jedna po jedna.

Pre nego što načinimo prvu izmenu, najpre je potrebno da prepoznamo ključni element koda koji se menja, pa zatim da najpre menjamo baš taj element koda, a tek posle, kao vid finog podešavanja, da popravljamo i druge delove koda. Svaka izmena bi trebalo da bude dobro izolovana i lokalizovana. Ako je zaista uočeno u čemu je problem, onda je jedna izmena uglavnom dovoljna. Ako postoji utisak da je potrebno napraviti više izmena da bi se problem rešio, onda je obično potrebno da nastavimo posmatranje koda, kako bismo bolje lokalizovali problem.

Česta greška pri debagovanju je tzv. „zaletanje" u izmene, tj. menjanje koda na osnovu neutemeljnih i neproverenih pretpostavki. Takav pistup obično ima za rezultat veći broj potpuno nepotrebno načinjenih izmena, što stvara uslove za pravljenje novih problema. Zbog toga je pri debagovanju potrebno da budemo *uzdržani*. Pre svake izmene je neophodno da dobro razmislimo:

- Da li je ta izmena neophodna?
- Da li ona sigurno rešava problem?

[53] Kada govorimo o „jednoj izmeni" ili o „paketu izmena", to se, pre svega, odnosi na određen broj izmena koje se prave u okviru jednog „atomičnog" koraka rešavanja nekog problema, u toku koga se ne proveravaju posledice načinjenih izmena.

- Da li postoji alternativa?

Ukoliko se ispostavi da načinjena izmena, suprotno očekivanjima, ne rešava problem, to znači da je lokalizacija problema loše ili nepotpuno sprovedena. Tada je najpre potrebno da se izmenjen deo koda vrati u početno stanje, da ne bi uticao na ponašanje sistema i nastavak procesa lokalizacije, kao i da ne bi ostao kao potencijalna nova greška.

Pravljenje privremenih izmena koda ima značajnu ulogu u procesu lokalizacije problema, pre svega u okviru testiranja hipoteza. U jednom broju slučajeva se pri lokalizaciji mogu uočiti potencijalne izmene koje otklanjaju posmatrani problem, ali koje se ne smeju praviti zbog toga što bi predstavljale suviše velike izmene ponašnja. Iako mogu da predstavljaju uzrok uočenog problema, takva mesta u kodu su obično samo posledica nekih manjih grešaka, koje bi trebalo pronaći. Ako sprovedena lokalizacija ipak ne može da dovede do drugih, manje invazivnih izmena, onda se i ovakve uočene izmene mogu upotrebiti. Ideja je da se ovakve „velike" izmene načine iako je jasno da nisu odgovarajuće, pa da se onda posmatra ponašanje sistema nakon menjanja i uporedi sa ponašanjem pre menjanja koda. Pri tome je potrebno porediti programski kod, ulazne i izlazne podatke, tragove izvršavanja, dnevnike izvršavanja i sve ostale raspoložive informacije. Analizom prikupljenih informacija se nekad mogu izvesti novi zaključci o lokaciji greške u kodu, koji omogućavaju njeno preciznije lokalizovanje.

Od velikog značaja pri lokalizovanju grešaka, a u vezi sa pravljenjem izmena, jeste čuvanje istorije verzija koda. Kada se uoči neki problem, može biti veoma korisno da se proveri da li je problem postojao u prethodnim verzijama koda. Pronalaženjem poslednje verzije koda u kojoj problem nije postojao i prve verzije koda u kojoj se on pojavljuje omogućava nam da poređenjem programskog koda tih verzija dođemo do značjnih informacija o uzrocima problema.

Posledice načinjenih izmena se proveravaju i logičkom analizom i testiranjem. Svaki od metoda ima svoje prednosti i mane. Logička analiza je pouzdanija od ograničenog broja testova, ali ona najčešće nije efektivno primenjiva pri debagovanju, zbog velikog broja potencijalno značajnih uslova. Sa druge strane, testovi zahtevaju veliku opreznost, zato što svaki pojedinačan test proverava ponašnje sistema samo u jednom vrlo specifičnom kontekstu i nikako ne može da predstavlja pouzdanu garanciju da će se softver uvek ponašati na ispravan način.

Neki programeri se relativno olako odlučuju da u okviru debagovanja preduzimaju korake refaktorisanja, kako bi popravili dizajn nekih delova koda, a radi olakšavanja pronalaženja greške. Koliko god da je refaktorisanje poželjno, ono ne bi trebalo da se odvija paralelno sa debagovanjem, ili bar ne dok nismo sasvim sigurni da smo locirali grešku. Ako bismo pokušavali da refaktorišemo tokom razumevanja i lociranja greške, onda bismo morali da se zapitamo kako da budemo

sigurni da refaktorisanjem nećemo promeniti ponašanje, kada mi uopšte i ne znamo kako se softver ponaša?

Koraci refaktorisanja i debagovanja moraju da budu jasno razdvojeni, zato što je njihova priroda potpuno drugačija – refaktorisanjem popravljamo *dizajn* koda (tj. njegovu strukturu) i nipošto ne menjamo ponašanje, a debagovanjem težimo da popravljamo *neispravno ponašanje*. Različita priroda ovih procesa ima za posledicu da je istovremeno preduzimanje debagovanja i refaktorisanja veoma rizično i teško proverivo. Refaktorisanje ne sme da menja rezultate postojećih (i eventualno novih) testova, a debagovanju je upravo cilj da promeni rezultate testova koji potvrđuju postojanje problema (ne menjajući rezultate drugih testova). Zbog toga što su koncepti na kojima počivaju ovi procesi potpuno različiti i čak prilično suprostavljeni, razlikuju se i mehanizmi njihovog sprovođenja. Njihovim eventualnim spajanjem bi se dobila nekonzistentna masa pravila i postupaka, koja u praksi donosi više problema (čitaj „novih grešaka“) nego što ih rešava.

U nekim slučajevima ispravljanje greške može da zahteva pravljenje „širokih“ izmena u različitim delovima koda. Takve greške su najčešće posledica propusta na nivou projekta, ili čak na nivou analize ili definisanja zahteva. U takvim slučajevima moramo da podelimo izmene na dve potpuno razvdvojene celine: refaktorisanje i debagovanje. Pre nego što se pristupi refaktorisanju, potrebno je da se postave jasni ciljevi i da se nedvosmisleno potvrdi tačna lokacija greške. Ciljevi se oblikuju na osnovu uočenih ograničenja postojeće strukture koda, tako da po njihovom ispunjavanju bude moguće rešavanje problema menjanjem relativno malog izolovanog i lokalizovanog dela koda. Prema tako postavljenim ciljevima se odabiru koraci refaktorisanja kojima će se ciljevi ispuniti. Svaki od koraka refaktorisanja se pravi u skladu sa pravilima refaktorisanja, sa testiranjem jedinica koda i ponašanja koda kao celine. Tek kada se refaktorisanje privede kraju, tj. kada su postavljeni ciljevi ispunjeni, a da ponašanje pri tome *nije izmenjeno*, onda možemo da pristupimo menjanju ponašanja, tj. otklanjanju greške.

Pravilo 6 – Praviti i čuvati tragove izvršavanja

Tragovi izvršavanja (engl. *trace*) su jedan od najvažnijih unutrašnjih alata za rešavanje problema. Oni su nezamenljivo sredstvo za obezbeđivanje informacija o toku izvršavanja u prostornoj i vremenskoj okolini greške. tragovi izvršavanja su posebno korisni u slučajevima kada je neophodno da se izvršavanje ponovi veliki broj puta da bi se greška ispoljila, ali i u slučajevima kada je praćenje sistema otežano iz bilo kojih razloga. Na primer, interaktivno praćenje izvršavanja distribuiranih sistema je veoma teško, pa se u distribuiranim okuženjima problemi po pravilu rešavaju uz pravljenje detaljnih tragova izvršavanja.

Kada se govori o tragovima izvršavanja, obično se pod time podrazumevaju delovi programskog koda koji automatski zapisuju informacije o stanju programa i

toku njegovog izvršavanja u pomoćnim izlaznim datotekama (tzv. *tragovi*, engl. *trace file*). Delovi koda koji služe za pravljenje tragova se obično pišu i upotrebljavaju samo u fazi debagovanja.

Drugi značajan oblik tragova predstavljaju tzv. manuelni tragovi izvršavanja, tj. dnevnici aktivnosti korisnika koji testira softver. Kao što im ime kaže, manuelni tragovi se ne prave automatski, iako mogu da se koriste alati koji olakšavaju njihovo pravljenje. Manuelni tragovi mogu da budu od presudnog značaja za razumevanje problema. Ako se redosled koraka zapisuje tek nakon ispoljavanja propusta, onda postoji mogućnost da korisnik slučajno zaboravi neku od prethodno načinjenih aktivnosti ili previdi neki od aspekata stanja sistema koje je prethodilo izvođenju zapisanog niza koraka. Može da se zameri da zapisivanje svih aktivnosti tokom testiranja predstavlja usporavajući faktor pri testiranju, ali je važnije što se na taj način štedi mnogo više vremena pri kasnijoj analizi i otklanjanju problema. Manuelni tragovi izvršavanja su posebno korisni u slučajevima kada neki naizgled beznačajan korak ili spoljašnji faktor utiče na ispoljavanje problema.

Pri pravljenju manuelnih tragova potrebno je da se zapisuju:

- detaljan opis stanja koje prethodi nizu aktivnosti;
- tačan redosled preduzetih aktivnosti;
- način izvođenja svih aktivnosti, detaljno;
- sve uočene posedice preduzetih aktivnosti.

Detljan opis stanja koje prethodi nizu aktivnosti može da se izostavi samo ako se testiranje nadovezuje na neki prethodno zapisan niz koraka, zato što tada stanje može da se rekonstruiše na osnovu tog prethodnog zapisa. Ipak, ako se tokom radnog dana obavlja veći broj različitih testiranja, dobro je da se bar povremeno reinicijalizuje i ponovo opiše trenutno stanje sistema.

Jedan od najvažnijih ciljeva upotrebe tragova izvršavanja je uspostavljanje relacija između simptoma (tj. ishoda izvršavanja) i potencijalnih uzroka (tj. uslova koji su doveli do ispoljavanja problema). Povezivanje simptoma i uzroka na osnovu tragova izvršavanja je moguće samo ako su i simptomi i uzroci navedeni u tragu izvršavanja. Ako bismo unapred znali simptome ili uzroke, onda bismo mogli da ograničimo sadržaj tragova samo na potrebne informacije. Međutim, onda nam tragovi izvršavanja verovatno ne bi ni bili potrebni. Teškoće pri debagovanju i nastaju zato što ne možemo da unapred pouzdano znamo ni da li smo uočili sve simptome ni koji su sve njihovi stvarni uzroci. Zbog toga se obično kaže da pisani trag izvršavanja (bilo automatski ili manuelni) nikada nije previše detaljan.

Zaista, ako je neki trag prevelik za manuelno iščitavanje i analiziranje, uvek se može pronaći (ili napraviti) neka alatka za automatsko pretraživanje, filtriranje i analiziranje teksta, a koja može da se koristi za obradu tragova izvršavanja. Već smo

više puta naglasili da su svi detalji važni i da je razlog pravljenja tragova upravo u tome da sprečimo da nam neki važan detalj promakne. I pored toga, ipak postoje relativno retki slučajevi kada trag izvršavanja može da bude preobiman. Tipičan primer su automatski tragovi izvršavanja kojima se zapisuje tok nekog složenog iterativnog računanja. Složenost izračunavanja i veliki broj iteracija mogu da proizvedu ogromne tragove, čije zapisivanje može da značajno uspori sam tok izračunavanja, a to usporavanje može da prikrije posmatrani problem ili čak da obesmisli konkretno izvršavanje.

Pravilo 7 – Proveravati naizgled trivijalne stvari

Uzroci i rešenja problema su često sasvim jednostavni, ali je to često veoma teško da se ustanovi. Da bi se razumelo o kom nivou jednostavnosti se ovde radi, možda je najbolje da navedemo nekoliko trivijalnih primera:

- Ako se pita nije ispekla, da li je rerna uopšte bila uključena?
- Ako automobil ne može da se pokrene, pre nego što ga rastavimo, možda bi bilo dobro da proverimo da li ima goriva?

Da, ovi primeri zaista izgledaju kao da je u pitanju neka šala – ali nije. Čitaocima koji nemaju značajnija iskustva u debagovanju, možda je teško da poveruju da takve greške u programiranju uopšte postoje. Onima koji takva iskustva imaju, verovatno ih je izlišno predočavati. Zajedničko za obe populacije je da se pri analiziranju i rešavanju nekog problema veoma često zanemaruju potencijalno jednostavni uzroci i rešenja.

Suština problema je u spontanom podrazumevanju nekih očiglednih stvari. Kada se suočimo sa nekim problemom, obično smo spremni na težak i ozbiljan rad na njegovom rešavanju. Samim tim, pokušavamo da razmišljamo istovremeno sveobuhvatno i apstraktno i da sagledavamo čitav sistem i njegove elemente. Pri tome smo veoma često skloni da zanemarimo mogućnost da neka od osnovnih pretpostavki za uspešno funkcionisanje nije ispunjena.

Da ne bismo dolazili u priliku da jednostavne i „očigledne“ uzroke i rešenja uočavamo tek posle više sati napornog rada, potrebno je da poštujemo ovo pravilo i na samom početku posmatranja problema najpre dovedemo u pitanje sve „podrazumevane“ pretpostavke. Ako smo iz bilo kog razloga pretpostavili da je neka komponenta ispravna, to ne znači automatski da ona zaista *jeste* ispravna. Zbog toga je analizu potrebno da *uvek* započnemo proveravanjem da li su ispunjeni svi preduslovi za izvršavanje problematičnog posla. Na primer, neki od uobičajenih preduslova su:

- Da li je korisnik pokrenuo ispravnu verziju softvera?

- Da li je softver ispravno instaliran?
- Da li je odgovarajući deo koda uopšte ugrađen u softver?
- Da li su ulazno izlazni uređaji ispravni?
- ... i razni drugi preduslovi.

Naravno, svaki razvojni projekat ima neke specifičnosti, pa i specifične preduslove koje im smisla proveravati na samom početku traženja grešaka. Na primer, ako se sistem sastoji od više distribuiraih komponenti softvera, onda provere ispunjenosti preduslova za obavljanje posla mogu da obuhvate i sledeća pitanja:

- Da li je računar *R1* uključen?
- Da li je na računaru *R1* pokrenuta komponenta softvera *K1*?
- Da li funkcioniše mreža?
- Da li su računar *R1* i komponenta *K1* dostupni na mreži?
- ... i druge slične specifične provere.

Osim provere ispunjenosti početnih uslova za obavljanje posla, u trivijalne probleme spadaju i različiti aspekti pogrešne upotrebe razvojnih alata. U ovom kontekstu u razvojne alate valja ubrojiti i programske jezike, prevodioce i različite biblioteke koje se koriste u projektu. Nije retkost da programeri misle da alat nešto radi na jedan način, a da se to zapravo odvija na neki sasvim drugi način. Na primer, početnici često greše pri upotrebi makroa u programskom jeziku *C*, zato što očekuju da se oni ponašaju kao funkcije, a njihovo stvarno ponašanje je sasvim drugačije.

Naravno, alata ima mnogo i relativno su složeni, pa je i pretpostavki o njhovom ponašanju mnogo. Zbog toga je praktično nemoguće dovoditi u pitanje doslovno svaku pretpostavku o njihovom ponašanju, posebno ne kao prvu stvar u debagovanju. Kada se radi o alatima, suština primene ovog pravila je u tome da se naglasi da (1) sve pretpostavke o alatima ne predstavljaju mrtvo slovo na papiru i *mogu* da se dovode u pitanje, kao i da je (2) neke pojedinačne pretpostavke *potrebno* dovoditi u pitanje, ali tek kada je problem dovoljno lokalizovan da takvih pretpostavki ima razumno mnogo.

Drugi aspekt ovog pravila se odnosi na proveravanje čak i onih uslova koji nam *izgledaju* nemoguće ili neverovatno. Sve dok nam nešto *izgleda* nemoguće, to znači da se ne radi o potvrđenoj pretpostavci već samo o *hipotezi*. A već smo videli (pravilo 3) da svaka hipoteza *mora* da se potvrdi, pre nego što počnemo da je koristimo kao činjenicu. Koliko god da nam je „očigledno“ da nešto ne može da bude uzrok problema, ne smemo tu mogućnost da odbacimo dok je ne proverimo. Tek kada različitim hipotezama i proverama hipoteza pouzdano isključimo mogućnost da

nešto pripada prostoru potencijalnih uzroka (pravilo 4), možemo to da isključimo iz daljeg razmatranja.

Pravilo 8 – Zatražiti tuđe mišljenje

U razvoju softvera, kao ni drugde uostalom, ne bi trebalo da bude egoizma, sujete ili ponosa. Ali ih, kao i drugde, ima i mogu da sprečavaju programera da zatraži savet od saradnika, prijatelja ili nekog nepoznatog eksperta. Argumentacija obično liči na pravdanja poput: „Ovo je moj deo softvera. Ja sam ga pisao i samo ja znam šta se tu dešava. Niko osim mene ne može da reši ovaj problem." Ali, trebalo bi da se vratimo korak nazad: „Da, ovo jeste moj deo softvera i niko drugi ga nije pisao osim mene. Ako problem postoji, onda sam ga *ja napravio*." A greške izvesno postoje, zato što se inače tim delom softvera ne bismo ni bavili u procesu debagovanja. Zašto bismo verovali da onaj ko je napravio grešku jeste pozvaniji i sposobniji da je otkloni nego neko drugi?

Naravno, deo argumentacije je sasvim ispravan – niko drugi ne poznaje neki deo koda bolje od njegovog autora. Ali u nekim slučajevima to nije *prednost* autora već njegov *hendikep*. U onim brojnijim slučajevima, kada je to prednost, autor je obično u stanju da uglavnom samostalno pronađe i otkloni grešku. Ali u onim malobrojnijim, u kojima ga traženje uzroka greške pošteno namuči, sva je prilika da bi neko sa strane mogao da bude od velike pomoći. Zašto?

Problem poznavanja nekog dela koda, ili čak i dugotrajnog bavljenja tim delom koda, na primer kroz pokušaje debagovanja, je u tome da *poznavanje* dovodi do *pretpostavki*. Da to malo apstrahujemo, svako *iskustvo* stvara nove *pretpostavke*. Hendikep je u tome što se takve pretpostavke prećutno, često i nesvesno, stvaraju i zatim više ne dovode u pitanje. Time se vraćamo na pravilo *7 - Proveravati i naizgled trivijalne stvari*. Ali, za razliku od uobičajene primene tog pravila, ovde postoji značajna otežavajuća okolnost – programer u praksi veoma teško uočava trivijalne pretpostavke koje su nastale kao rezultat njegovog sopstvenog iskustva u radu sa posmatranim delom koda. Jednostavno, stepen podrazumevanja je preveliki.

„Spoljašnji posmatrač" koji ne poznaje mnogo posmatrani deo koda, mogao bi da svojim „pogledom sa strane" bolje sagleda potencijalne slabosti koje promiču „unutrašnjem posmatraču". Neko ko nije opterećen suvišnim pretpostavkama može mnogo bolje da rasuđuje o posmatranom problemu. To je ujedno i jedna od najznačajnijih motivacija za traženje pomoći. Međutim, čak i kada zatražimo pomoć, ponovo možemo napraviti grešku. Iste lične slabosti koje nas mogu sprečavati u traženju pomoći, mogu da se ispolje i kada se pomoć zatraži, u vidu pravdanja i objašnjavanja šta smo to sve učinili bez uspeha. Suvišna objašnjenja mogu imati samo negativne posledice, zato što se kroz njih na saradnika prenose iste one pretpostavke koje nas ometaju u radu.

Druga grupa motiva za traženje pomoći se odnosi na ograničene mogućnosti pojedinca u pogledu poznavanja brojnih tehnika i tehnologija koje se prepliću u savremenom razvoju softvera. Za svaku oblast postoje eksperti. Istovremeno, niko nije dovoljno dobar ekspert za sve oblasti koje su zastupljene u razvoju softvera. Koliko god da mislimo da dobro poznajemo neku konkretnu tehnologiju, uvek će postojati eksperti koji se bave *samo njom* i koji je poznaju mnogo bolje od nas. Umesto da gubimo sate i dane pokušavajući da razumemo neki problem, može biti mnogo jednostavnije, efikasnije i jevtinije da se zatraži pomoć od eksperta. Sasvim je moguće da je neko već rešavao problem koji nas muči, pa nema razloga da ga ponovo rešavamo. Može da se kaže da eksperti znaju gde da udare mašinu čekićem, pa da ona proradi. Naravno, većina nas bi čekićem samo nanela nova oštećenja.

Od presudnog značaja je da se saradniku pri opisivanju problema, navode *samo simptomi*. Pretpostavke se ne smeju saopštavati! Umesto da pozvanog saradnika zatrpamo svim bitnim i nebitnim, proverenim i neproverenim informacijama kojima raspolažemo, mnogo je produktivnije da sve te informacije sačuvamo za sebe sve dok nam ne bude postavljeno odgovarajuće pitanje, pa i tada moramo biti „štedljivi". Čak i ako mislimo da smo neke pretpostavke dokazali i da izvesno važe, ne smemo ih prenositi pomagaču. Potrebno je pustiti pomagača da uči o sistemu *neopterećen* našim pogledom na problem. Naravno, pomagač će postavljati pitanja, ali bi odgovori kojemu dajemo trebalo da budu prvenstveno informativne prirode, tako da nude informacije a ne zaključke. Na primer, na pitanje „Da li je proveren taj i taj uslov?", odgovor ne bi nikako smeo da bude u obliku „Da, to je sigurno u redu." već u opisnom obliku poput „Provereno je to i to na taj i taj način i rezultat je bio takav i takav." Ako smo od nekoga tražili pomoć, važno je da ga pustimo da proba da pronađe odgovore, umesto da mu ih sami nudimo – prisetimo se, da su naši odgovori tačni, pomoć nam ne bi ni bila potrebna.

Danas je pristup informacijama mnogo lakši nego u prethodnim decenijama. Uloga Interneta u razvoju softvera je jednostavno nezamenljiva. Dok je ranije „pasivna pomoć" bila ograničena na različita uputstva, tutorijale, vodiče kroz probleme (engl. *troubleshooting guides*) i slične kolekcije iskustava, danas se mnogo informacija može pronaći na ličnim veb lokacijama i blogovima, gde programeri zapisuju svoja zanimljiva iskustva. Diskusione grupe pružaju istovremeno sasvim specifičan vid aktivne i pasivne pomoći. Aktivnu pomoć možemo zatražiti postavljanjem pitanja i očekivanjem da nam neko ponudi odgovor, a pasivna nam je na raspolaganju u obliku uvida u već vođene diskusije.

Pasivna pomoć je posebno korisna u slučajevima kada pretpostavljamo da smo naišli na neispravno ili nedokumentovano ponašanje neke komponente ili biblioteke koju koristimo. Princip je jednostavan – ako problem zaista postoji, vrlo je velika verovatnoća da ga je neko drugi već uočio pre nas, kao i da ga je negde zapisao.

Pravilo 9 – Ako je nismo ispravili, onda greška nije ispravljena

Moglo bi se reći da je ovo pravilo toliko jasno da je nepotrebno, no to ovde nije slučaj. Izvesno ga nije ga teško razumeti. Osnovno značenje je sasvim jasno. Ipak, ovo pravilo ima nešto šire implikacije, koje u složenim okolnostima razvoja softvera ne smeju ni da se zanemare ni da se podrazumevaju.

Nije nikakva retkost da neki problem ne može da se ponovi. Pri tome se ne misli na problem čije ponavljanje ne umemo da izvedemo zbog nepoznavanja odgovarajućeg postupka (pravilo 2), nego na slučajeve kada procedura koja je proizvodila problem naprasno prestane da ga proizvodi. Razlozi za to su brojni: moguće da je neka druga izmena programskog koda uticala na pojavljivanje posmatranog problema, ili je možda neka komponenta zamenjena novom verzijom pa se ona ponaša drugačije, ili su neke okolnosti u okviru računarskog sistema izmenjene pa se zato problem ne ispoljava. Ipak, nama je važnije da je sasvim moguće da se problem i dalje ispoljava, ali na neki drugi način, pa mi to ne uočavamo.

Jedan od razloga da ne zanemarimo ovo pravilo je što praksa potvrđuje (a i čuveni Marfijev zakon nas uverava) da će problem koji nije rešen, a koji je naprasno prestao da se ispoljava, sasvim verovatno isto tako naprasno ponovo da se ispolji onda kada nam to bude najmanje odgovaralo. Ako je problem „nestao“ sam od sebe, to bi trebalo da nam sugeriše da zapravo nismo dvoljno dobro upoznali okolnosti u kojima se problem pojavljuje. Promena tih nepoznatih okolnosti je uticala na ispoljavanje greške. Ako se okolnosti ponovo promene, problem može ponovo da počne da se ispoljava.

Ako je promena nekih delova sistema proizvela posledicu da se posmatrani problem ne ispoljava, a pouzdano znamo da se niko nije bavio njegovim rešavanjem, to znači da je neka od načinjenih promena proizvela relativno široke efekte, a to opet može da znači i da je proizvela i neke druge, potencijalno loše posledice. Sasvim je moguće da su rešeni samo neki aspekti problema i da već neka minimalna promena niza koraka može ponovo da dovede do ispoljavanja problema.

Prestanak ispoljavanja problema često može da znači da je samo prestalo ispoljavanje uočenih simptoma, ali da problem i dalje postoji, ali se ispoljava na drugi način. Ovaj slučaj možemo da uporedimo sa zamenjivanjem pregorelog električnog osigurača – uzroci pregorevanja i nakon zamene nastavljaju da postoje i mogu da se ispolje pregorevanjem novog osigurača, ali i na neki drugi, mnogo neugodniji način. Zbog toga se svim „nestalim“ problemima mora posvetiti pažnja na isti način i u istoj meri kao problemima koji „nisu nestali“.

Uobičajeno je da prvi korak u ovakvim slučajevima predstavlja ozbiljna provera da li je problem zaista popravljen. Jedan od načina je da se vratimo do poslednje verzije koda u kojoj se problem ispoljavao i pokušamo da ga lokalizujemo. Zatim

lokalizovan deo koda uporedimo sa novijim verzijama koda, u kojima se problem ne ispoljava. Ako uspemo da prepoznamo da je neka konkretna promena izmenila ponašanje na odgovarajući način onda je i moguće da je problem sada rešen.

Ovde je neophodno da budemo posebno oprezni – slučajno ili usputno rešavanje drugih problema je često nedovoljno temeljno. Zato proveru da li je greška popravljena moramo da izvodimo veoma pažljivo i ozbiljno. Nekada je korisno da posle lokalizacije pristupimo i rešavanju problema na staroj verziji koda, pa da tek posle toga upoređujemo naše izmene sa onima koje je neko drugi napravio. Na taj način možemo da smanjimo uticaj već načinjenih izmena na naše rasuđivanje o problemu, što važno zato što pod takvim uticajem možemo da pomisimo da je rešenje potpunije i kvalitetnije nego što zaista jeste.

Jedan značajan aspekt ovog pravila se odnosi na pitanja kvaliteta razvojnog procesa. Ako imamo neke greške, to znači da razvojni proces ima nekih slabosti. Pri rešavanju problema smo često u prilici da uočimo uzroke njihovog nastajanja i da na taj način postepeno popravljamo i unapređujemo razvojni proces. Ako bismo poverovali da je neki problem nestao sam od sebe, ili da je „slučajno“ rešen kroz neke druge popravke, time bismo propustili da sagledamo propuste u razvojnom procesu koji su doveli do njegovog nastajanja. Samim tim, mogli bismo očekivati da će se pojaviti neki drugi problem, kao posledica istih slabosti u procesu razvoja, a što bi moglo da se izbegne posvećivanjem pažnje „nestalom“ problemu.

Dodatna pravila

Većina skupova pravila debagovanja, sa kojima je autor ovog teksta imao dodira, su ekvialentni sa ovim navedenim Agensovim skupom pravila ili nekim njegovim podskupom. Postoje i neka pravila koja nisu doslovno obuhvaćena Agensovim skupom, na primer[54]:

- Nacrtati dijagram
- Opisati problem nekom standardnom metodologijom
- Redukovati ulazne podatke koji su neophodni za ponavljanje problema
- Razumeti zahteve
- Pojednostaviti test-slučaj koji dovodi do ponavljanja problema
- Pročitati tačnu poruku o grešci,
- Koristiti odgovarajuće alate
- Ispratiti ispravljenu grešku novim testom
- i mnoga druga

[54] Navedena pravila predstavljaju podskup od 13 pravila iz [Grotker2008] i 9 pravila iz [Metzger2004], koja nisu trivijalno ekvivalentna sa Agensovim pravilima.

Ako malo bolje pogledamo navedene primere pravila, možemo videti da, iako nisu neposredno obuhvaćena Agensovim pravilima, većina njih može da se izvede kao posledica ili specifičan vid primene njegovih pravila. Na primer, crtanje dijagrama i opisivanje problema nekom standardnom metodologijom su samo vrlo konkretne primene pravila 6 „praviti i čuvati tragove izvršavanja“, razumevanje zahteva je samo podskup šireg razumevanja sistema i sl.

Dodavanje novih dobrih pravila u našu svakodnevnu praksu ne može da škodi. Naprotiv, može da unapredi svakodnevnu praksu debagovanja. Sa druge strane, detaljnije posvećivanje većem broju pravila pri učenju heurističkog metoda debagovanja može da nepoželjno raširi priču i oteža razumevanje suštine metoda. Zbog toga Agensov skup predstavlja idealnu polaznu osnovu, a svako od nas je slobodan da proširuje i prilagođava taj skup pravila prema sopstvenim merilima i potrebama.

Heuristički metod nije u potpunosti suprotstavljen neformalnom metodu, već predstavlja njegovu prirodnu nadgradnju. Neki autori čak sugerišu da najpre isprobamo sve one potencijalne popravke koje nam izgledaju „očigledno“, pa tek ako to ne uspe da se posvetimo sporijem i pouzdanijem postupku debagovanja.

14.6 Tehnike i alati

Sve tehnike i alate za prevenciju i otklanjanje grešaka u programskom kodu delimo na *spoljašnje* i *unutrašnje*.

Spoljašnje tehnike i alati su različiti programski alati koji su napravljeni za uopštenu primenu, a ne za neke specifične slučajeve. Takve tehnike i alati se koriste u svakodnevnom radu pri debagovanju različitih projekata.

Unutrašnje tehnike i alati su elementi koji se ugrađuju u programski kod a nemaju nikakvu svrhu u kontekstu izvršavanja posla kome je program namenjen, već služe isključivo radi pomoći u prevenciji i otklanjanju grešaka. Često predstavljaju tzv. *ad hoc* sredstva, koja se razvijaju za rešavanje jednog specifičnog problema. Neki unutrašnji alati mogu poslužiti i za rešavanje više različitih problema, ali vrlo retko van relativno ograničenog dela posmatranog programskog koda.

14.6.1 Spoljašnje tehnike i alati

Debager

Najvažniji spoljašnji alat za debagovanje je debager. Debager (engl. *debugger*) je alat koji omogućava detaljno posmatranje stanja računarskog sistema i izvornog koda programa u vreme izvršavanja programa. Primenom različitih tehnika se korisniku (tj. programeru koji otklanja neki problem) pruža pomoć u praćenju

izvršavanja i posmatranju stanja programa, a sa osnovnim ciljem pružanja što informativnijeg uvida u rad programa i pomoći pri lociranju uzroka problema.

Debageri mogu da budu sastavni deo većih razvojnih alata, tzv. integrisanih razvojnih okruženja (engl. *Integrated Development Environment - IDE*) ili samostalni proizvodi. Prednost itegrisanih debagera je u tome što su obično vrlo dobro uklopljeni u razvojni alat i omogućavaju povezanost delova okruženja za razvoj i debagovanje. Sa druge strane, prednost samostalnih debagera je u mogućnosti njihovog povezivanja sa različitim razvojnim okruženjima i programskim jezicima. Primeri razvojnih okruženja sa integrisan debagerom su *MS Visual Studio* i *Eclipse*, dok je najpoznatiji primer samostalnog debagera *GNU* debager *GDB*, koji se koristi iz komandne linije ili pomoću neke od brojnih dodatnih implementacija vizualnog interfejsa.

Sve ono što može da se uradi pomoću debagera moglo bi (uz malo ili malo više truda) da se uradi i pomoću odgovarajućih unutrašnjih alata. Prednost debagera u tome što je daleko jednostavnije koristiti gotove alate nego ih praviti svaki put iznova kada nam zatrebaju. Navešćemo neke od osnovnih alata kojima raspolažu savremeni debageri:

Izvršavanje programa „korak-po-korak“. Programer ima priliku da prati izvršavanje programa od samog početka njegovog izvršavanja, tako što eksplicitno zahteva da se izvrši jedna po jedna naredna naredba. U slučaju svakog poziva potprograma može da bira da li će ući u telo potprograma i pratiti njegovo izvršavanje korak-po-korak (obično se takva komanda naziva „korak-unutra“, engl. *Step Into*) ili će izvršiti pozivanje potprograma kao jedan veći korak (komanda „korak-iznad“, engl. *Step Over*). Većina debagera ima i opciju „izlazak iz potprograma“ kojom se automatski izvršavaju svi koraci tekućeg potprograma, zaključno sa povratkom na mesto odakle je on pozvan (komanda „korak-van“, engl. *Step Out* ili *Step Return*). Neki debageri imaju i opciju „izvršavanje do kursora“ (engl. *Run To*), kojom se nalaže da se nastavi izvršavanje dok se ne dođe do naredbe koju je programer izabrao.

Postavljanje tačaka prekida. Izvršavanje velikih programa korak-po-korak može da bude dugotrajno. Umesto toga, mnogo je efikasnije pustiti da se program izvršava dok ne dođe do neke konkretne naredbe. Tačke prekide su oznake koje se postavljaju na naredbe u programu i koje sugerišu debageru da mora da stane kada dođe do neke od njih. Po dostizanju neke od tačaka prekida izvršavanje programa se privremeno zaustavlja i prepušta se kontrola programeru. Izvršavanje može da se nastavi korak-po-korak ili nastavljanjem izvršavanja do naredne tačke prekida (komanda „nastavak“, engl. *Resume* ili *Continue*).

Uslovne tačke prekida. Često je potrebno da se tačka prekida postavi na naredbi koja se izvršava veliki broj puta, ali da je poželjno da se program ne zaustavlja svaki put kada se dođe do tačke prekida, već samo u nekim specifičnim slučajevima.

Uslovne tačke prekida omogućavaju da se uz tačku prekida definiše uslov koji će automatski da se proverava kada se ta tačka dostigne i na osnovu koga će debager da odlučuje da li je potrebno da se program zaustavi ili da se nastavi izvršavanje. Alternativa je da se broje prolasci kroz tačku prekida i da se program zaustavi tek nakon zadatog broja prolazaka.

Tačke prekida na podacima. Osim na naredbama, tačke prekida mogu da se postavljaju i na podacima. Tačka prekida na podatku se vezuje za neku oblast u memoriji i nalaže debageru da svaki put kada program pristupi toj zoni memorije mora da prekine njegovo izvršavanje. Kao i u slučaju tačaka prekida u programu, i tačke prekida na podacima mogu da budu uslovne. Uslov može da obuhvati i vrstu pristupa, tako da se, na primer, program zaustavi samo ako se pokuša pisanje.

Tačke prekida u slučaju izbacivanja izuzetaka. Većina debagera omogućava da se automatski prekine izvršavanje programa neposredno pre izbacivanja izuzetka. Na taj način se programeru omogućava da posmatra stanje programa koje je dovelo do izbacivanja izuzetka.

Lokalne promenljive. Razlog za zaustavljanja programa u nekom trenutku ili izvršavanja programa korak-po-korak je to što programer želi da posmatra stanje programa. Jedan od osnovnih elemenata stanja programa predstavljaju lokalne promenljive. Dok je program privremeno zaustavljen, debager omogućava detaljan uvid u lokalne promenljive.

Globalne promenljive. Lokalne promenljive predstavljaju samo uzak lokalni deo stanja programa. Za dobijanje šire slike moraju da se prate i posmatraju i vrednosti globalnih promenljivih. Globalnih promenljivih je često mnogo, pa nije praktično da se sve prikazuju, ali debageri obično pružaju korisnicima mogućnost da sami navedu koje su to promenljive čije stanje žele da prate.

Pregledanje podataka. Složene strukture podataka ne mogu lako da se sagledaju samo posmatranjem vrednosti promenljivih, već je potrebno da se pažljivo pregledaju neki elementi podataka ili povezani podaci (atributi objekata, objekti na koje pokazuju pokazivači, atributi objekata na koje pokazuju pokazivači,...). Na primer, neki debageri umeju da prepoznaju i na odgovarajući način prikažu strukture podataka poput niza ili liste.

Praćenje stanja steka. Programski stek služi za prenošenje argumenata i rezultata potprograma i za čuvanje adrese povratka iz potprograma. Analizom stanja steka može se utvrditi kako je tok programa došao od glavnog programa (funkcije `main` u jeziku C/C++) do potprograma koji se trenutno izvršava. Prikazivanjem sadržaja steka u preglednom obliku, debager omogućava programeru da vidi odakle je pozvan tekući potprogram, odakle je pozvan prethodni potprogram i tako dalje sve do nivoa glavnog programa. Štaviše, debager može da pruži programeru uvid u stanje lokalnih promenljivih u svakom od tih potprograma. Analiza sadržaja steka je

jedan od najvažnijih aspekata upotrebe debagera. Praćenjem steka i stanja lokalnih promenljivih na različitim nivoima pozivanja može da se ustanovi odakle potiču neke eventualno neispravne vrednosti argumenata potprograma.

Praćenje stanja niti. Savremeni razvoj softvera podrazumeva pisanje programa koji se izvršavaju konkurentno, u više niti. Osnovni problem pri debagovanju programa koji se izvršavaju u više niti je u otežanom praćenju izvršavanja i otežanoj lokalizaciji problema. Debager mora da omogući praćenje stanja svake od niti. Obično debageri omogućavaju da se izabrana nit privremeno suspenduje, kako bi se smanjio obim istovremenih promena stanja i olakšalo fokusiranje posmatranja na preostale niti.

Praćenje toka programa na nivou mašinskih isntrukcija. Obično je mnogo lakše, bolje i brže analizirati izvršavanje pograma uz posmatranje izvornog koda programa, u obliku u kome je originalno napisan na konkretnom programskom jeziku, ali u nekim specifičnim slučajevima može da bude potrebno da se program posmatra i izvršava na nivou mašinskih instrukcija procesora. Većina debagera nam pruža tu mogućnost. Ona je korisna u slučajevima kada je potrebno proveriti da li je optimizacija izvedena na odgovarajući način ili da li je neki deo koda preveden tako da može da se izvršava neatomično. Posebno, mnogi savremeni prevodioci omogućavaju da se u okviru teksta programa na višem programskom jeziku, između konstrukcija tog jezika, eksplicitno navodi asemblerski zapis mašinskih instrukcija. To se naziva „umetnuti asemblerski kod“ (engl. inline assembler) i omogućava da se i programi koji zahtevaju maksimalne performanse razvijaju najvećim delom na višem programskom jeziku, a da se samo oni delovi koji su posebno osetljivi dodatno optimizuju primenom asemblera. Debagovanje delova programa koji su pisani sa umetnutim asemblerskim kodom je obično nemoguće bez praćenja izvršavanja programa na nivou mašinskih instrukcija. Sve što je navedeno za izvršavanje programa korak-po-korak i postavljanje tačaka prekida, važi i kada se program izvršava na nivou mašinskih instrukcija, tim da se kao jedinice izvršavanja ne posmatraju naredbe i konstrukcije višeg programskog jezika nego mašinske instrukcije procesora.

Praćenje stanja procesora. Ako se tok izvršavanja programa prati na nivou mašinskih istrukcija, onda se stanje programa, makar na lokalnom nivou, mora pratiti posmatranjem stanja procesora. Stanje procesora čine vrednosti registara, uključujući i stanje registara zastavica, koje opisuje režime rada procesora i privremene podatke veličine jednog bita. Debageri, koji omogućavaju praćenje izvršavanja programa na nivou mašinskih instrukcija, po pravilu omogućavaju i praćenje stanja procesora.

Udaljeno debagovanje. U distribuiranim okruženjima, ili kada se razvija softver za neki specifičan uređaj, često nije moguće interaktivno upotrebljavati debager na sistemu na kome se izvršava program koji se debaguje. Mnogi savremeni debageri

podržavaju tzv. „udaljeno debagovanje“ (engl. *remote debugging*), kada se jedna komponenta debagera izvršava na istom sistemu gde i debagovani program, a druga na radnoj stanici koju interaktivno upotrebljava korisnik. Prva komponenta obuhvata sve sistemske i funkcionalne elemente debagera, a druga samo predstavlja korisnički interfejs. Naravno, između ove dve komponente mora da postoji komunikacija. Najčešće se upotrebljava uobičajena mrežna infrastruktura, ali se mogu upotrebljavati i neke druge posvećene veze, npr. putem serijskog ili USB interfejsa. U slučaju udaljenog debagovanja, arhitekture sistema koji se debaguje i radne stanice mogu biti potpuno nezavisne. Jedino ograničenje je da konkretna verzija debagera mora da podržava obe vrste računarskih sistema. Ako se ima u vidu da se kao radne stanice obično koriste standardizovane platforme (npr. PC + Linux ili PC + Windows ili Apple), koje su dobro podržane u pogledu alata za debagovanje, onda je potencijalan problem samo obezbeđivanje udaljene komponente debagera za konkretan računarski sistem ili uređaj.

Simulirano debagovanje. Predstavlja podvrstu udaljenog debagovanja. U slučaju simuliranog debagovanja se debagovani program i funkcionalna komponenta debagera izvršavaju u okviru odgovarajućeg simulatora sistema. Arhitektura simulatora je vrlo slična virtualnim mašinama, s tim da se pretpostavlja veća nezavisnost u arhitekturi matičnog i gostujćeg sistema. Iz ugla debagovanja, stvari na konceptualnom nivou funkcionišu skoro potpuno isto kao u slučaju udaljenog debagovanja.

Profajler

Osnovna namena profajlera (engl *profiler*) je da olakša uočavanje delova programa koji predstavljaju usko grlo u pogledu zauzeća nekog od resursa, a pre svega u pogledu zauzeća procesora ili memorije. Iako im nije osnovna namena debagovanje, oni mogu da budu od pomoći u slučajevima kada su problemi vezani upravo za preterano zauzeće resursa.

Neke od osnovnih tehnika upotrebe profajlera su brojanje učestalosti izvršavanja određenih potprograma ili merenje vremena provedenog tokom njihovog izvršavanja. Neke od osnovnih tehnika implementacije su ugradnja brojača u programski kod (što se u poslednje vreme relativno retko praktikuje, zato što zahteva ugradnju posebnih biblioteka i često značajno usporava izvršavanje osetljivih delova koda) i učestalo zaustavljanje programa i snimanje stanja. Na primer, profajler može da zaustavlja izvršavanje programa na svakih 100ms i da zapisuje šta se tom prilikom izvršavalo i koje je bilo stanje steka. Na taj način se omogućava relativno brojanje učestalosti upotrebe potprograma – ne zna se tačan broj pozivanja, ali se zna učestalost pozivanja u odnosu na upotrebu drugih delova programa.

O upotrebi profajlera će biti nešto više reči u poglavlju posvećenom optimizaciji softvera.

Alati za dinamičku analizu

Alati za dinamičku analizu programa prave posebno okruženje za pokretanje programa, koje omogućava da se prate različiti parametri programa tokom njegovog izvršavanja. Obično simuliraju standardne biblioteke operativnog sistema ili čak stvaraju okruženje koje podseća na virtualnu mašinu ili kutiju sa peskom (engl. *sandbox*), tako da izvršavani program *ima osećaj* da se izvršava u realnim uslovima, a da je okruženje zapravo potpuno kontrolisano od strane alata za dinamičku analizu.

Jedan od najpoznatijih alata za dinamičku analizu je *Valgrind*. *Valgrind* je kolekcija alata za dinamičku analizu koja obuhvata programe za praćenje svih operacija sa memorijom (alokacija, dealokacija, upotreba) i detekciju neispravnih pristupanja, curenja memorijei drugih problema; za praćenje upotrebe keša i uočavanje delova programa koji potencijalno rade sporije zato što ne koriste keš na pravi način; za praćenje pozivanja funkcija, slično profajlerima ali na egzaktnom nivou; za praćenje upotrebe globalne memorije (hip); detekciju grešaka u komunikaciji i sinhronizaciji niti i drugo.

Čistači

Čistači (engl. *sanitizer*, moglo bi se prevesti i kao *alati za dezinfekciju*) su alati koji se isporučuju u okviru prevodilaca i rade sličnu stvar kao alati za dinamičku analizu, ali tako što se fizički ugrađuju u prevedenu izvršnu verziju programa. Namena čistača je da prilikom izvršavanja programa evidentiraju sve aktivnosti koje su od značaja (u skladu sa upotrebljenim opcijama prevodioca) i izveštavaju programera o uočenim neispravnostima. Na primer, prevodioci `Clang` i `g++` imaju veliki broj opcija oblika `-fsanitize=...`, kojima se uključuju odgovarajući čistači.

Čistači mogu da se uporede sa alatima za dinamičku analizu i imaju i neke prednosti i neke slabosti. Razlike uglavnom potiču otud što čistači imaju neposredan pristup izvornom kodu programa i deo pripreme za svoj rad obavljaju još u vreme prevođenja programa, dok programi za dinamičku analizu znaju samo za izvršni oblik programa. Kao osnovne razlike bismo mogli da istaknemo:

- čistači su obično daleko brži;
- čistači mogu precizno da se podešavaju opcijama prevodioca;
- čistači su novija tehnologija i slabije su dokumentovani;
- čistači ne mogu da pomognu ako je problem u nekoj upotrebljenoj biblioteci a ne u našem kodu;
- neke opcije čistača ne mogu da se koriste zajedno, pa za neke složene provere može da bude potrebno da se program više puta izgrađuje sa različitim opcijama.

14.6.2 Unutrašnje tehnike i alati

Unutrašnje tehnike i alati su različiti elementi koji se ugrađuju u programski kod ali nemaju svrhu u kontekstu izvršavanja posla kome je program namenjen, već služe isključivo radi pomoći u prevenciji i otklanjanju grešaka. U najvažnije unutrašnje tehnike spadaju:

- proveravanje pretpostavki:
  - o stanju ulaznih argumenata na početku potprograma;
  - o stanju promenljivih u toku algoritma;
  - o stanju promenljivih na kraju potprograma;
- pravljenje tragova izvršavanja:
  - o prolasku kroz neku tačku programa;
  - o stanju promenljivih u nekim tačkama programa;
- umetanje specifičnih delova koda
  - radi sprovođenja složenijih analiza stanja;
  - radi izvođenja istog posla na drugi način;
  - radi pravljenja čitljivog zapisa stanja ili dela stanja programa;
- dodavanje testova jedinica koda:
  - u skladu sa proverenim pretpostavkama;
  - u skladu sa neproverenim pretpostavkama;
- pisanje jasnog i razumljivog koda:
  - dobro imenovanje promenljivih i potprograma;
  - komentarisanje programskog koda u skladu sa proverenim pretpostavkama
  - i drugi elementi.

Testovima jedinica koda je posvećeno posebno poglavlje (*9 - ...Razvoj vođen testovima*, na stranici 180), a pisanju jasnog i razumljivog koda je posvećena praktično cela knjiga. Ovde ćemo da posvetimo pažnju preostalim navedenim tehnikama.

Proveravanje pretpostavki

Proveravanje pretpostavki (engl. *assert*) u programskom kodu je tehnika koja omogućava da se na odgovarajućim mestima ugrade automatske provere nekih uslova, za koje se očekuje da na tim mestima moraju da važe. U slučaju evenualne neispunjenosti proveravanog uslova, uobičajeno je da se prekine rad programa i na

određeni način saopšti odgovarajuća poruka. U retkim slučajevima se reaguje ispisivanjem poruke i nastavljanjem rada programa.

Pretpostavke se proveravaju nekim oblikom funkcije (ili makroa) `assert`.

U narednom primeru je predstavljeno kako u kodu pisanom na programskom jeziku C++[55] može da se proverava pretpostavka da je argument funkcije `fact` u odgovarajućem opsegu:

```
#include <cassert>
...
int fact( int n )
{
    assert( n>=0 );
    assert( n<100 );
    int r = 1;
    while( n > 1 )
        r *= (n--);
    return r;
}
```

Uobičajeno je da se proveravaju sasvim jednostavne pretpostavke. Ako je potrebno proveravati neke složene uslove, preporučljivo je da se oni podele na manje delove pa da se svaki od njih posebno proverava. Razlog za to je jednostavan – ako neki složen uslov nije ispunjen, pri analizi problema se neće znati koji konkretan deo tog složenog uslova nije ispunjen. Sa druge strane, ako nije ispunjen neki jednostavan uslov, onda je jasnije o čemu se radi. Zato su u prethodnom primeru navedene dve odvojene pretpostavke za proveravanje uslova `n>=0` i uslova `n<100`.

U najčešće proveravane uslove spadaju provere ispravnosti opsega argumenata ili međusobne uskađenosti vrednosti različitih argumenata potprograma. Takve provere same za sebe ne govore mnogo, ali ako se pokaže da neki potprogram pozivamo sa neispravnim argumentima, onda znamo da je potrebno da problem tražimo na mestima gde se taj potprogram upotrebljava.

U složenijim potprogramima može da bude potrebno da se proverava stanje promenljivih u toku izvođenja nekog algoritma. Provere međurezultata se najčešće implementiraju kao provere ispravnosti grananja, provere ispravnosti iteracija i provere ispravnosti spajanja. Provere ispravnosti grananja se ugrađuju na početku svake od grana pri uslovnom grananju i predstavljaju dodatnu proveru važenja uslova koji u svakoj od grana moraju biti ispunjeni. Provera ispravnosti iteracije služi da se prveri da li su na početku iteracije ispunjeni uslovi koji moraju važiti svaki put pre početka narednog ponavljanja. Provera ispravnosti spajanja se koristi na mestima spajanja različitih grana (npr. iza bloka `if-then-else`) ili iza naredbi ponavljanja, da

[55] Skoro isto je i u programskom jeziku C, s tim da je se uključuje zaglavlje `<assert.h>` umesto `<cassert>`.

bi se proverilo da li je odgovarajuća celina koda ispravno uradila posao za koji je zadužena.

Često može biti korisno da se proveravaju i rezultati potprograma. Ako izlazne vrednosti potprograma ne zadovoljavaju određene uslove u odnosu na ulazne vrednosti, onda znamo da u konkretnom potprogramu imamo grešku.

Proveravanje ispravnosti ulaznih podataka predstavlja deo pisanja robusnih programa i vid prevencije grešaka. Za razliku od toga, proveravanje ispravnosti izlaznih podataka i stanja promenljivih u toku potprograma obično se preduzima tek u fazi debagovanja. Međutim, kada se proveravanje pretpostavki jedanput napiše, dobro je da se one ostave u programu, tj. da se ne brišu nakon pronalaženja i ispravljanja tražene greške.

U većini programskih jezika i prevodilaca, kao i u nekim od alata ili biblioteka, postoji način da se prevodiocu naglasi da li je proveravanje pretpostavki potrebno uključiti u prevedeni izvršni program ili ne. Na taj način se omogućava da se proveravanje pretpostavki ugradi u razvojne verzije, koje se upotrebljavaju u početnim fazama testiranja i pri debagovanju, a da se iz konačne izvršne verzije softvera one isključe da ne bi umanjivale performanse sistema. U slučaju programskih jezika C i C++, za proveravanje pretspostavki se obično koristi makro `assert`, koji proverava stanje datog uslova i prekida rad programa ako uslov nije zadovoljen (kao u prethodnom primeru). Pretpostavke se proveravaju samo u fazi debagovanja programa, dok je u slučaju produkcione verzije ovaj makro definisan kao prazan kod i ne proizvodi nikakvo dejstvo. Produkciona verzija se prepoznaje po predefinisanom makrou `NDEBUG`, koji nije definisan u verziji za debagovanje i testiranje.

U mnogim projektima se prave dve vrste pretpostavki, koje se razlikuju samo po tome što se jedna od njih zadržava i u izvršnim verzijama prevedenog softvera, a druga ne. Provere pretpostavki koje ostaju u izvršnim programima mogu da se naprave i tako da ne prekidaju rad programa, već da samo izdaju odgovarajuće obaveštenje, zapisuju informacije u dnevnik događaja i zatim omogućavaju nastavak rada programa. Takve provere omogućavaju prikupljanje informacija koje mogu da olakšaju otkrivanje i prepoznavanje problema uočenih u fazi eksploatacije softvera. Nije dobra praksa da se sve provere pretpostavki ostavljaju u izvršnom kodu, zbog potencijalnog ugrožavanja performansi. Čak i jednostavne provere, ako se nalaze u petljama i potprogramima koji imaju jednostavna tela a izvršavaju se veoma često, mogu da veoma negativno utiču na efikasnost izvršavanja.

U programskom jeziku C++ (od verzije 11) postoji i mogućnost *statičkog* proveravanja pretpostavki. Radi se o pretpostavkama čije važenje se proverava u fazi prevođenja programa i prijavljuje kao greška već u toj fazi, tako da ne uzima vreme u fazi izvršavanja. Takve pretpostavke mogu da se navedu kako u bloku programa tako i van njega. Na primer:

```
template<typename T, unsigned N>
class Point {
    static_assert( N < 100, "Dimension too high" );
    ...
};

static_assert( sizeof(Point<double,70>) <= sizeof(buffer) );
```

U zavisnosti od korišćenih alata, ali i konkretnih razvojnih projekata, mogu da se definišu i koriste različiti alati za pisanje pretpostavki. Na primer, ako se ispostavi da detaljno proveravanje pretpostavki u toku nekog potprograma može da unese neprihvatljivo usporenje u program (iako samo u fazi debagovanja), onda se opcijama prevođenja i specifičnim makroima upravlja uključivanjem i isključivanjem proveravanja pretpostavki u različitim delovima programa.

Pravljenje tragova izvršavanja

Pravljenje tragova izvršavanja programa se obično implementira kroz upotrebu različitih makroa ili potprograma koji zapisuju odgovarajuće informacije u nekoj datoteci ili u izlaznom toku za greške. Tragovi izvršavanja programa mogu da budu veoma obimni. Zbog toga, da bi se lakše pratili i analizirali, uobičajeno je da svaki zapis sadrži tačno vreme i mesto izdavanja. Mesto izdavanja služi za lakše lociranje mesta u kodu na kome je poruka izdata i obično obuhvata naziv potprograma i/ili naziv izvornog fajla programa i broj reda u njemu.

Slično proveravanju pretpostavki i pravljenje tragova može da opterećuje performanse softvera. Štaviše, pravljenje tragova je obično mnogo intenzivnije i obimnije, pa i posledice po performanse mogu da budu mnogo veće. Zbog toga je uobičajeno da se tragovi izvršavanja ne uključuju u izvršne verzije programa, a ako je neophodan neki vid praćenja rada radi kontrole kvaliteta, onda se to radi u sasvim ograničenom obimu.

U velikim projektima se često koriste ili čak implementiraju posebne biblioteke za pravljenje tragova, koje sadrže različite potprograme, kao i različite opcije detaljnosti ili formata ispisivanja tragova. Tako se postiže da potrebne informacije mogu da se izdaju u odgovarajućim prilagođenim formatima, ali i da tragovi izvršavanja mogu da se analiziraju, pretražuju ili upoređuju primenom različitih alata.

Na primer, na programskom jeziku C++ bi makro `DEBUG_TRACE_CERR` za pravljenja traga izvršavanja mogao da se implementira i koristi na sledeći način:

```
#ifdef NDEBUG
   #define DEBUG_TRACE_CERR(msgs)
#else
   #define DEBUG_TRACE_CERR(msgs)                    \
      std::cerr                                      \
      << " * TRACE: " << __FUNCTION__ << std::endl   \
      << " *        " << __FILE__                    \
```

```
      << " [" << __LINE__ << "]" << std::endl       \
      << " *        " << msgs
#endif

...{...
   DEBUG_TRACE_CERR( "[x=" << x << "]" )
...}...
```

Umetanje specifičnih delova koda

Greške obično nastaju u složenim delovima programa ili su bar povezane sa složenim stanjem koje se prenosi (implicitno ili eksplicitno) kroz različite delove programa. Zbog toga ispisivanje jednostavnih tragova izvršavanja, koji sadrže samo lokaciji ili pojedinačne vrednosti promenljivih lesto može da bude nedovoljno informativno.

U takvim slučajevima je obično neophodno da se dodaju posebni delovi programa koji služe isključivo za olakšavanje postupka debagovanja i ne bi trebalo da imaju bilo kakvog uticaja na produkcionu verziju softvera. Takvi delovi programa mogu da imaju različitu formu i namenu.

Po formi razlikujemo potprograme i isečke koda. Ako pišemo posebne potprograme, koji služe samo za debagovanje, onda nas oni ne brinu previše, zato što njihovo postojanje samo po sebi ne utiče na produkcionu verziju – najpre zato što ako se neki potprogram ne koristi, onda on ne troši vreme i resurse osim mesto u prevedenoj verziji programa, ali zapravo ne troši ni to zato što se u fazi povezivanja programa obično odbacuju delovi koji se nigde ne koriste. Sa druge strane, ako u postojeće potprograme ubacujemo specifične isečke koji nam služe pri debagovanju, onda bi bilo dobro da imamo sredstvo za njihovo isključivanje iz produkcione verzije programa.

Najčešća namena umetnutih isečaka koda jeste pravljenje čitljivog zapisa stanja programa, obično prevođenjem neke složene strukture u tekst. Takav zapis zatim može da se koristi u okviru tragova izvršavanja ali može i da se posmatra u okviru primene nekog spoljašnjeg alata. Na primer, ako privremeno zaustavimo izvršavanje programa, onda možemo da vidimo stanje svih lokalnih promenljivih, pa tako i stanje tako pripremljenog čitljivog zapisa stanja.

U takvim slučajevima možemo da koristimo delove koda poput:

```
std::string opis1 = ...;
std::string opis2 = ...;
```

Da bismo takav isečak koda imali samo u verziji za debagovanje, možemo da napravimo i da koristimo makroe nalik na naredni primer:

```
#ifdef NDEBUG
   #define DEBUG_RUN(x)
```

```
#else
   #define DEBUG_RUN(x) x
#endif

...{...
   DEBUG_RUN(
      std::string opis1 = ...;
      std::string opis2 = ...;
   )
...}...
```

Pored ovakve namene, umetnuti segmenti programa mogu da imaju i mnogo složenije namene. Na primer, mogu da obavljaju dodatna izračunavanja ili analize da bi se automatski proverila ispravnost stanja programa na konkretnom mestu i u konkretnom trenutku. Zbog lakšeg snalaženja u programu, preporučljivo je da sve složenije analize i obrade budu implementirane u dodatnim pomoćnim potprogramima, a da se u potprograme koje debagujemo ubacuju samo njihovi pozivi.

14.6.3 Strategije i taktike debagovanja

Opisivanjem neformalnog, naučnog i heurističkog debagovanja nismo iscrpli sve pristupe debagovanju. Na primer, Čarls Mecger u knjizi [Metzger2004] predlaže da se postupak debagovanja veže za matematički metod i u okviru toga oblikuje strategije, heuristike i taktike debagovanja. Heuristikama smo već posvetili pažnju, pa ćemo ovde ukratko prikazati strategije i taktike.

Strategije predstavljaju viši nivo organizovanja i planiranja postupka debagovanja. Uglavnom su oblikovane analogno sa različitim pristupima pretraživanju složenih struktura podataka. Izražene su u vidu algoritama, gde ulaz predstavlja skup svih segmenata programskom koda u kome se pretpostavlja da bi trebalo da je greška, a izlaz je redukovan skup segmenata (poželjno jednočlan). Pomenimo neke od njih:

- strategija binarnog pretraživanja – delimo skup na dva dela i proveravamo u kom skupu je greška;
- strategija pohlepnog pretraživanja – bira se jedan po jedan segment iz ulaznog skupa i proverava se da li je u njemu greška, pri čemu metod izbora daje prioritet onim segmentima koji na osnovu nekog kriterijuma izgledaju kao verovatnija lokacija greške;
- strategija pretragom u širinu – polazeći od najšireg poziva u kome se detektuje greška, proveravati svaki poziv potprograma, pa ako se u njemu detektuje greška, onda sve njegove pozive potprograma dodati kao kandidate na kraj liste;

- strategija pretragom u dubinu – slično kao prethodni, ali se potprogrami dodaju na početak liste;
- strategija programskih isečaka – pravljenje i testiranje isečaka programa, tako da oni obuhvataju delove izračunavanja ili promene stanja povezanih sa manifestacijom greške;
- strategija deduktivne analize – u osnovi počiva na uopštenom postavljanju hipoteza deduktivnim zaključivanjem i
- strategija induktivne analize – u osnovi počiva na uopštenom postavljanju hipoteza induktivnim zaključivanjem;

Taktike su manji postupci (često elementarni), koji se preduzimaju u okviru primene širih heuristika i strategija, za efektivno proveravanje da li je greška u nekom posmatranom delu programskog koda. U knjizi je u vidu kataloga predstavljena 21 taktika. Veći broj taktika predstavlja primenu već obrađenih unutrašnjih i spoljašnjih tehnika, ili može da se svede na savete za upotrebu alata za dinamičku analizu i čistača. Ovde ćemo navesti samo neke od preostalih taktika:

- čitati izvorni kod i tražiti probleme;
- praviti izveštaje o kompletnom stanju memorije;
- zameniti lokalne promenljive globalnim;
- prevoditi program do nivoa asemblerskog koda;
- probati da li greška postoji i kada se koristi prevodilac drugog proizvođača;
- isprobati prevođenje i izvrašavanje programa na drugom operativnom sistemu.

14.7 ...Prevencija propusta

U uvodnom delu ovog poglavlja smo videli da upravljanje propustima obuhvata aktivnosti na prevenciji propusta i aktivnosti na otklanjanju propusta. Do sada smo se uglavnom bavili otklanjanjem propusta. Videli smo da je debagovanje složen i zahtevan proces, koji uz to i nije posebno prijatan. Sada je vreme da se posvetimo prevenciji propusta i da vidimo da li postoji naki način da doprinesemo smanjenju broja grešaka koje se prave tokom rada na razvoju softvera.

U radnom okruženju mogu da postoje neke okolnosti koje posebno doprinose nastanku propusta. Takođe, postoje i okolnosti koje doprinose smanjivanju učestalosti i ozbiljnosti propusta. Veliki značaj imaju i okolnosti koje ne utiču

presudno na nastajanje propusta, ali mogu kasnije da doprinesu njihovom lakšem uočavanju, lociranju i otklanjanju.

Osnovni cilj prevencije propusta je stvaranje uslova za nastajanje što manjeg broja propusta i njihovo blagovremeno uočavanje. Redovnim sagledavanjem i proveravanjem uslova rada i odnosa u timu obično može da se blagovremeno uočiti prisustvo ili odsustvo poželjnih ili nepoželjnih okolnosti. Zatim odgovarajućim koracima uslovi i način rada mogu da se unapređuju ili prilagođavaju, kako bi se stanje popravilo.

Okolnosti koje pogoduju nastajanju propusta

Među najvažnijim okolnostima koje doprinose nastajanju propusta su nedovoljna stručnost tima i nepotrebno povećan nivo stresa u timu.

Nedovoljna stručnost se različito ispoljava na različitim nivoima organizacije tima. Primer nestručnog ponašanja na poziciji pogramera je primena pristupa „kodiraj pa razmišljaj“. Brzopleto pisanje koda stvara uslove za nastajanje mnogih propusta, od pogrešnog tumačenja specifikacije, pa sve do pravljenja grešaka u algoritmima i njihovoj implementaciji. Na višem nivou, nestručnost i žurba često dovode do lošeg razumevanja zahteva, nedovoljno dobre analize problema i na kraju izrade neispravnog projekta, koji nije u skladu sa postavljenim zahtevima. Konačno, ako stručnost izostane na nivou rukovodstva tima, onda se kao posledica dobija odsustvo posvećenosti kvalitetu, a time i prostor za brojne propuste u svim fazama razvoja i svim delovima softvera koje tim razvija.

Na rukovodstvu razvojnog tima leži najveća odgovornost za stvaranje dovoljno velikog i dovoljno stručnog tima. Ovaj aspekt problema je obično dobro prepoznat, što često nije dovoljna prepreka da se zapostavlja.

Nepotrebno povećanje nivoa stresa u razvojnom timu je najčešće posledica lošeg planiranja ili loših međuljudski odnosa u timu. Loše planiranje obuhvata sve vidove lošeg raspoređivanja resursa, uključujući i nedovoljan broj kadrova i njihovu obučenost. Svi oblici nestručnosti u timu na neki način, pre ili kasnije, negativno utiču i na međuljudske odnose i na rokove za dovršavanje razvoja, pa time posredno i na povećavanje nivoa stresa u timu.

Loši međuljudski odnosi neminovno vode lošem razumevanju između članova tima, što stvara posebno pogodne uslove za različita tumačenja zahteva, specifikacija i drugih vidova dokumentacije. Pored toga, članovi tima koji koji nemaju dobru komunikaciju sa svojim saradnicima najpre postaju nezadovoljni, a zatim često i dodatno opterećeni činjenicom da moraju da sarađuju sa osobama sa kojima se ne razumeju. To podiže nivo stresa kod pojedinaca, što se neminovno preslikava i na tim kao celinu.

Osnovni problem sa povećanim nivoom stresa je u dugotrajnom izlaganju tima otežanim uslovima rada. Dok je većina ljudi je u stanju da na kratkotrajan stres

odreaguje pozitivno i u takvim uslovima pruži možda i više nego što bi inače pružila, dotle su posledice izlaganja dugotrajnom stresu upravo suprotne – čak i veoma sposobni i stručni članovi tima će u slučaju produženog stresa početi da prave neuobičajeno mnogo propusta. Različite su vrste stresa koje mogu da opterećuju članove tima i ceo tim. Neke od njih mogu biti prisutne u praktično svakom razvojnom okruženju, kao, na primer: prekovremeni rad, različite vrste pritisaka (obećavanje nagrada, pretnje kaznama, učestalo ponavljanje i insistiranje na neostvarivim planovima i drugo), loši međuljudski odnosi, loši ambijentalni uslovi (neprovetrenost, manjak svetla, loša oprema i slično), neprilagođene radne procedure (komplikovani i spori formalni postupci, mnogo dokumentacije,...) i drugo. Sa druge strane, svako konkretno radno okruženje ima neke specifične karakteristike, što sa sobom nosi i mogućnosti za nastanak nekih specifičnih vrsta stresa.

Na rukovodstvu razvojnog tima leži najveća odgovornost za stvaranje dobrih uslova za rad i prepoznavanje i suzbijanje nepotrebnog stresa i loših međuljudskih odnosa. Međutim, rukovodstvo to ne može da postigne bez dobre saradnje svih članova tima. Važno je imati na umu da je dobar tim od dobrih pojedinaca često daleko uspešniji nego loš tim od izuzetnih pojedinaca. Usklađenost članova tima i kvalitet njihovih međusobnih odnosa često su važniji od sposobnosti pojedinačnih članova tima.

Okolnosti za izbegavanje propusta

Kao što postoje okolnosti koje pogoduju nastajanju propusta, tako postoje i one koje smanjuju mogućnost nastajanja propusta. Neke od njih neposredno ili posredno sprečavaju ispoljavanje ili smanjuju značaj već pominjanih okolnosti koje pogoduju nastajanju propusta. Druge donose u razvojno okruženje neke nove kvalitete, koji doprinose uspešnom radu.

Neke od najznačajnijih okolnosti za izbegavanje (ili bar smanjivanje verovatnoće nastajanja) propusta su:

- dobri odnosi i komunikacija u timu;
- stalno usavršavanje članova tima;
- dobra komunikacija sa klijentom;
- relativno opušteni uslovi rada;
- sistematičnost i obezbeđivanje kvaliteta;
- i druge.

U prethodnom odeljku smo videli da je nedovoljna stručnost članova tima (uključujući i rukovodioce) jedan od glavnih faktora koji utiču na nastajanje grešaka.

u skladu sa tim, neke od navedenih okolnosti za izbegavanje propusta se, posredno ili neposredno, odnose na problem stručnosti. Naravno, osnovni put prema dobroj osposobljenosti članova tima vodi preko pažljivog odabira kadrova i njihovog stalnog usavršavanja. Ako imamo u vidu da je tim već formiran, onda nema mnogo prostora da se utiče na odabir kadrova. Eventualno može da se sugeriše nadređenima da je potrebno obezbediti pojačanje timu, ako se ustanovi da među članovima tima postoji značajan nedostatak osposobljenosti u nekoj od oblasti.

Sa druge strane, uvek ima i mora da bude prostora za stalno usavršavanje članova tima. To je najpouzdaniji način za redovno i sistematično podizanje nivoa stručnosti i pojedinaca i celog tima. Ako govorimo o bagovima, onda se „stručnost" najpre odnosi na poznavanje alata koji se upotrebljavaju u svakodnevnom radu, a pre svega na dobro poznavanje principa programiranja, programskih jezika, prevodilaca i drugih alata. Međutim, ako se problem posmatra malo šire, onda se „stručnost" odnosi i na principe projektovanja, na šire razumevanje sistema koji se razvija, kao i okruženja u kome bi on trebalo da funkcioniše, kao i na različite aspekte problema čijem je rešavanju tim posvećen.

Često se prenebregava da je za usavršavanje članova tima potrebno vreme, a da bi se to vreme obezbedilo, neophodno je da se smanji angažovanje na nekoj drugoj strani. Ne može da se očekuje da će programeri, koji rade prekovremeno na nekim veoma složenim problemima, pa možda i u drugim stresnim uslovima, imati dovoljno vremena, motiva i energije da se dodatno usavršavaju. Zato je dobro da se planom rada predvidi neko vreme za čitanje i učenje, makar to bilo i svega par sati nedeljno. U radu tima povremeno može da nastupi period smanjene aktivnosti, kada iz nekog razloga postoji smanjeni pritisak rokova i obima posla (na primer, ako se čeka na neku značajnu odluku klijenta). Takve periode je dobro iskoristiti za usavršavanje članova tima, kako kroz pojedinačni rad, tako i kroz zajedničk rad na nekim temama za koje je ustanovljeno da zahtevaju bolje poznavanje.

Kao drugi značajan faktor uticaja na nastajanje bagova smo naveli stresne uslove rada. Nezadovoljstvo i stres mogu da nastaju kao posledica mnogih dešavanja na radnom mestu, pa i suočavanje sa tim problemom mora da ima odgovarajuću širinu. Uticaj međuljudskih odnosa na nivo stresa je među najvažnijim faktorima, čak do te mere da predstavlja jedan od glavnih faktora zbog kojih se zaposelni odlučuju da ostanu u timu i preduzeću ili da se trude da ih promene. Neki od elemenata radnog okruženja mogu da neposredno utiču na opuštenij radnu atmosferu, na primer udobna radna stolica, odgovarajuća visina radnog stola, kvalitetan interfejs računara (monitor, tastatura, miš), prijatan ambijent i drugo.

Sistematičnost i rad na obezbeđivanju kvaliteta imaju višestruk uticaj na nastajanje grešaka. Sa jedne strane, tehnike koje se tu koriste (testovi jedinica koda, testovi prihvatljivosti, pisanje robusnog softvera, upravljanje rizicima i druge) imaju za osnovni cilj olakšavanje blagovremenog uočavanja problema, o čemu je već bilo

reči. Ali sa druge strane, sistematičnost i konkretne pojednačne tehnike značajno doprinose ugodnosti na radnom mestu. Praktično sve tehnike koje se uvode sa ciljem uređivanja procesa razvoja istovremeno imaju i pozitivan uticaj na smanjenje stresa u timu.

Dobra komunikacija sa klijentom može da se posmatra i u okviru staranja o kvalitetu, ali i u okviru razmatranja kvaliteta međuljudskih odnosa. Ona najpre u velikoj meri utiče na kvalitet specifikacija i jasnoću ciljeva i planova, ali može da ima značajan ticaj i na motivaciju članova tima.

Okolnosti za lociranje propusta

Samo u trivijalnim projektima može da se govori o tome da propusti *mogu* da nastanu. U svakom iole složenijem projektu, koliko god da se pažnje posveti sprečavanju nastajanja propusta, oni *svakako* nastaju, samo je pitanje kada, gde i kako. Zbog toga se u oblasti razvoja softvera veliki značaj pridaje ne samo stvaranju radnog okruženja u kome će do propusta ređe dolaziti, nego i do stvaranja preduslova da se nastali propusti lakše uočavaju, lociraju i otklanjaju.

Jedan od najlakših načina da se locira greška u programskom kodu jeste poređenje programskog koda verzije u kojoj greška postoji sa programskim kodom neke prethodne verzije, u kojoj greška nije postojala. Zbog toga je jedna od najvažnijih i nezamenljivih tehnika u razvoju programa čuvanje svih prethodnih verzija programskog koda. To se radi pomoću alata za praćenje verzija, kao što su *GIT*, *SVN* i drugi. U slučaju većih projekata može da bude relativno teško preuzimati i prevoditi različite sačuvane verzije programa, zbog čega je dobro da se čuvaju različite verzije izgrađenih izvršnih verzija programa, kako bi lakše moglo da se proverava da li se u nekoj od njih ispoljava neki problem ili ne.

Osim programskog koda, drugi značajan izvor informacija predstavlja prepiska između članova razvojnog tima, kao i prepiska sa klijentom. Za ovaj vid komunikacije može da se upotrebljava elektronska pošta, ali je pretraživanje daleko jednostavnije i uspešnije ako se upotrebljava neki od alata za praćenje poslova i problema. Savremeni alati za praćenje poslova i problema omogućavaju da se sve relevantne informacije o različitim elementima razvoja zapisuju sistematično i centralizovano. U praksi je upotreba ovakvih sistema daleko pouzdanija i efikasnija nego upotreba elektronske pošte, posebno ako se ima u vidu potencijalno ogroman obim cirkulacije poruka u iole većem razvojnom timu.

Da bi se greške ispoljile (što je neophodno da bi se uočile i zatim locirale), nekada nije dovoljno prevoditi samo izmenjene delove koda. Zbog toga se preporučuje često i plansko redovno *građenje koda*. Terminom *građenje koda* se označava postupak prevođenja čitavog projekta, bez korišćenja ranije prevedenih međurezultata. U slučaju velikih projekata ovo može biti relativno dugotrajan proces, ali je njegov značaj dovoljan da ga ima smisla automatizovati tako da se odvija planski. Da

problem bude još veći, često nije dovoljno testirati prevedeni program na računarima na kojima se razvija, već je potrebno izvršiti potpunu instalaciju softvera, kako bi okolnosti upotrebe u potpunosti odgovarale planiranoj ciljnoj platformi. U ozbiljnim testiranjima (koja se ne odnose samo na uske delove sistema i pojedinačne operacije) uobičajeno je da se koristi samo softver koji je preveden punim građenjem koda i instaliran na ciljnoj platformi, zato što u drugim slučajevima neke greške mogu lakše da ostanu neprimećene.

Dodatna korist od redovnog građenja koda se ostvaruje ako se čuvaju sve (ili izabrane) izgrađene izvršne verzije programa. Na taj način se olakšava proveravanje kako su se ponašale prethodne verzije i da li se problem ispoljavao i u nekoj od njih.

Potencijalan problem u vezi sa prevođenjem programa može da bude relativno površan odnos programera prema *upozorenjima* koja se dobijaju od prevodilaca. Dok programeri moraju da se posvete porukama o *greškama* u prevođenju, zato što se program inače ne bi uspešno preveo, sa druge strane *upozorenja* uglavnom bezbolno mogu da se ignorišu. To je veoma loša praksa. Upozorenja prevodilaca se obično odnose na delove programa koji su potencijalno dvosmisleni i sadrže potencijalne greške, ili ostavljaju prostor za kasnije nastajanje grešaka. Na primer, upozorenja se dobijaju u slučaju implicitnih konverzija tipa, naredbi koje se nikada ne izršavaju, zapisivanja podatka veće dužine u promenljivoj manje dužine i slično. Da se ne bi ostavio prostor za greške, poželjno je pročistiti programski kod tako da se ne dobija nijedno upozorenje. To je obično sasvim jednostavan posao, ako se radi redovno, tako da je korist daleko veća od neophodnog napora.

Veoma čest problem pri lociranju grešaka može da predstavlja nečitak programski kod. Čitljivost programskog koda može da se ostvari na više načina, ali uobičajeno je da se u timovima uvode i poštuju pravila o načinu pisanja i formatiranja i razumnom komentarisanju programskog koda. Pravila o načinu pisanja koda obuhvataju načine imenovanja podataka i potprograma, kao i vizualno oblikovanje programskog koda. Dosledno poštovanje ovih pravila značajno olakšava programerima uočavanje i razumevanje delova programskog koda, što je preduslov za uspešno pronalaženje grešaka i održavanje koda. Savremeni programerski editori omogućavaju da se programski kod automatski vizualno uređuje i to konfigurabilno, u skladu sa pravilima koja su određena u timu.

Komentarisanje programa

Razumno komentarisanje programa podrazumeva uvođenje i poštovanje pravila o komentarisanju celina u programskom kodu (moduli, klase, potprogrami) i pojedinačnih elemenata koda. U savremenom razvoju je uobičajena upotreba alata za automatsko pravljenje dokumentacije na osnovu programskog koda i komentara koji su napisani u skladu sa pravilima konkretnog alata. Ako dokumentacija već mora da se pravi (a obično mora), onda je to verovatno najbolji način da se ostvari ažurnost programske dokumentacije. Alat za generisanje dokumentacije *Doxygen* je u toj

oblasti postao nezvanični standard [Doxygen]. Razvijen je 1997. godine i vrlo brzo je stekao zasluženu popularnost. Danas se koristi u velikom broju projekata, a postoji i veliki broj pomoćnih alata koji dodatno nadgrađuju ono što *Doxygen* napravi.

Obično se uvode pravila da bi pojedinačni elementi koda (naredbe, izrazi, blokovi) trebalo da se komentarišu samo onda kada nije sasvim očigledno šta se na nekom mestu radi ili izračunava, ili zašto je nešto urađeno na možda malo neuobičajen način. Obično se sugeriše da se komentarima ne objašnjavaju elementi programskog koda, nego motivacija za njihovo pisanje, razlozi zašto su neophodni i slično. Programeri bi trebalo da dovoljno dobro poznaju programski jezik i alate da mogu da razumeju šta neki element programskog koda radi (ili opisuje) i bez dodatnih komentara, a ako ipak nije očigledno čemu taj element služi u konkretnom kontekstu, onda bi komentarima trebalo opisati upravo njegovu ulogu i smisao, a ne njegovo tehničko značenje u konktekstu programskog jezika.

Komentari u programskom kodu predstavljaju veoma značajan vid dokumentacije. Postoji mnogo korisnih informacija koje mogu da se navedu u programskom kodu, ali da bi komentari imali smisla oni ne smeju da budu preterani. Trivijalne komentare bi svakako trebalo izbegavati, zato što nepotrebno opterećuju programera tokom čitanja koda.

Često je bolje da se nejasni delovi koda izdvoje u posebne potprograme, zato što onda na osnovu imena potprograma i argumenata i možda jednostavnog komentara, uloga tog dela koda može da bude znatno jasnija nego kada je detaljno prokomentarisan u sklopu neke veće celine.

Neke od uobičajenih vrsta komentara obuhvataju:

- **Zaglavlje datoteke.** Na početku svake datoteke sa izvornim kodom programa je poželjno da se vrlo krakto (sa svega par reči) navede kom pojektu ona pripada i šta je njen sadržaj.
- **Opis namene definicije.** Pre svake definicije (klasa, funkcija, metod, podatak) je poželjno da se navede kratak opis namene elementa koji se definiše. U slučaju klase se opisuje njena funkcija, možda ukratko glavni deo interfejsa ili uobičajen model upotrebe. Možda neka važnija napomena o mestu u hijerarhiji ili ulozi u nekom obrascu ili delu projekta. U slučaju funkcije (metoda) ili podatka (atributa) navode se uloga i celina kojoj pripada. U slučaju funckije (metoda) navode se i aspekti menjanja stanja, ako nisu očigledni.
- **Opis ulaznih i izlaznih podataka.** U slučaju funkcija i metoda mogu da se opisuju ulazni ili izlazni argumenati, kao i rezultat funkcije. Ne opisuju se ako su očigledni.

- **Opis promene stanja.** Ako funkcija (metod) proizvodi bočni efekat, odnosno menja stanje nekog podatka (objekta), a da to nije očigledno naziva funkcije i argumenata, onda je to potrebno da se opiše.
- **Vlasništvo.** Ako funkcija preuzima vlasništvo nad nekim argumentom ili ne predaje vlasništvo nad rezultatom, to je poželjno da se objasni. Na primer, u programskom jeziku C++ je uobičajeno je da se vlasništvo ne menja kada se radi sa referencama, ali nije uvek jasno šta se dešava kada se prenose pokazivači.
- **Algoritam.** Algoritam se objašnjava samo ako nije očigledan. Može da se objašnjava u celosti, u okviru jednog uvodnog komentara, ili po segmentima u okviru tela funkcije, na mestima gde se implementiraju pojedinačni koraci algoritma. Umesto opisa algoritma, češće je potrebno da se objasni zašto je izabran baš taj algoritam a ne neki drugi. Takvo objašnjenje može da spreči da neko u budućnosti eventualno izgubi vreme na razmatranju drugog algoritma, a što u datom kontekstu možda nema smisla.
- **Objašnjenja nelogičnih elemenata koda.** Ponead u programski kod moraju da se ugrade neki naizgled nepotrebni ili besmisleni elementi. To je obično slučaj kada neka specifična konstrukcija, za koju bi se očekivalo da nema značajno dejstvo, ima posledice po način prevođenja i optimizovanja delova koda od strane neke verzije prevodioca. Takođe, i kada se radi manuelna optimizacija, vrlo često može da izgleda da je napisan kod besmislen, a da on ipak ima neku važnu ulogu u kontekstu optimizacije.

Kao što postoje stvari koje je dobro da se objašnjavaju komentarima, tako postoje i loši i neprimereni komentari. Kao što je već navedeno, komentari ne smeju da budu trivijalni, tj. da obuhvataju očigledne informacije. Komentari koji opisuju nešto što je i bez njih sasvim jasno, predstavljaju opterećenje po korisnike izvornog koda. Svaki suvišan red komentara troši vreme čitaoca i istovremeno nepotrebno skreće pažnju sa bitnih detalja. Ako napišemo 20 rečenica komentara, od kojih su samo dve važne, a ostale su očigledne, onda je sasvim moguće da neki čitalac ne uoči te dve važne rečenice u masi beznačajnih. Tipični primeri trivijalnih komentara su oni poput „ova povećavamo brojač za 1“ ili „argument `brojElemenataNiza` određuje veličinu niza mereno brojem elemenata“. Iako je sasvim očigledno da od takvih komentara niko ne može da ima koristi, komentari tog tipa se iznenađujuće često koriste, pa se čak u razvojnim timovima često insistira na navođenju bar po jedne rečenice objašnjenja za svaki argument potprograma.

Neki od primera loših komentara su:

- **Trivijalni komentari.** To su obično komentari koji nepotrebno prevode program sa programskom jezika na govorni jezik ili objašnjavaju očigledne stvari.
- **Dobri komentari za loš kod.** Ako se jednostavnim komentarom opisuje jednostavan a nerazumljiv kod, onda to obično znači da bi taj kod trebalo napisati tako da bude razumljiviji. Nekada je dovoljno upotrebiti razumljivija imena promenljivih.
- **Dugački i nejasni komentari.** Ako se neki deo koda opisuje dugačkim komentarom, to obično znači da tu postoji neki problem sa implementacijom.
- **Neprecizne i nejasne reference.** Kada se u komentaru referiše na neki dokument ili veb lokaciju, onda je to potrebno da bude kratko ali nedvosmisleno. Neprecizne i nejasne reference prave više štete nego koristi.
- **Objašnjenja koja nisu primerena kontekstu.** Neprimereno je da se u komentaru izlaže nešto što se ne odnosi na konkretan segment programskog koda. Na primer, nije dobro da se arhitektura sistema opisuje u okviru datoteke izvornog koda koja implementira samo jedan njen manji deo. Umesto toga je bolje da se navede samo kratka rečenica sa referencom na deo dokumentacije gde može da se pronađe odgovarajući opis. Ako neko već gleda izvorni kod, onda verovatno već zna ponešto o arhitekturi, ali je još veći problem što se takvi komentari retko ažuriraju pri promeni šireg konteksta na koji se odnose, pa mogu da postanu dezinformacija.

Komentari relativno često mogu da budu naznaka da nešto u programu nije u redu. Martin Fauler to lepo opisuje, kada kaže da se komentari često koriste kao dezodorans i da je propručljivo da se pre pisanja komentara uvek najpre pokuša sa refaktorisanjem [Fowler1999]. Zaista, suviše se često dešava da bi komentar mogao potpuno da se odstrani ako bi se odgovarajući segment koda preuredio. Kako se to često kaže, programski kod bi trebalo da govori za sebe, a ne da ga komentari zastupaju.

Iako se u oblasti agilnog razvoja može naići na ekstremne stavove, koji predlažu potpuno eliminisanje komentara i insistiranje na promeni strukture koda umesto toga, ipak bismo zaključili da je najbolje pronaći dobar balans i pisati komentare, ali samo tamo gde su zaista potrebni i u obimu u kome su korisni.

14.8 Umesto zaključka

Kada započinjemo novi projekat, često podstaknuti svojim i tuđim velikim pa i revolucionarnim planovima, poslednje o čemu želimo da razmišljamo su greške koje ćemo tokom rada na projektu napraviti. Prirodno je da želimo da se fokusiramo na sve ono lepo i dobro što taj projekat nosi, a ne na ono što bi moglo da nam pokvari dan ili čak čitav projekat. Međutim, kako stvari stoje, ispada da je jedini način da nam greške ne pokvare previše dana (a možda čak i ceo projekat) upravo da im se od samog početka vrlo temeljno i strpljivo posvetimo – najpre kroz prevenciju, a kasnije kroz sistematično pronalaženje i otklanjanje.

Što više naučimo o greškama i debagovanju, to će nam lakše biti ne samo da pronalazimo i rešavamo probleme, nego i da prepoznajemo okolnosti koje nas vode prema problemima. Zato je dobro da se tome na vreme posveti vreme.

Postoji mnogo knjiga o debagovanju i nije lako izabrati i preporučiti neke od njih. U ovom poglavlju su već pominjani neki izvori, a pre svih su istaknute knjige Andreasa Zelera [Zeller2006] i Dejvida Agansa [Agans2006]. U prvoj se temeljno i argumentovano izlažu razlčiti aspekti procesa debagovanja, pa se na kraju čak navodi i jedna formalizacija postupka debagovanja. U drugoj se na veoma praktičan način izlažu neka od najvažnijih pravila debagovanja. Zanimljiv pristup debagovanju primenom *matematičkog metoda* se izlaže u knjizi Čarlsa Mecgera [Metzger2004]. Knjiga „Vodič kroz debagovanje za razvijaoce“ [Grotker2008] je više praktično orijentisana i nudi opise i uputstva za primenu alata za debagovanje i detektovanje problema sa memorijom, kao i elemente upotrebe profajlera, a sve to na primeru programskog jezika C++.