Pouzdano inženjerstvo

MAGDALENA MIĆIĆ

1038/2014

Teme

- **Redundantnost i raznovrsnost**
  - Osnovni pristupi kojima se postiže tolerancija na greške
- **Pouzdani procesi**
  - Kako upotreba pouzdanih procesa vodi ka pouzdanim sistemima
- **Pouzdane arhitekture sistema**
  - Arhitektonski šabloni za toleranciju na greške u softveru
- **Pouzdano programiranje**
  - Smernice za izbegavanje grešaka prilikom programiranja

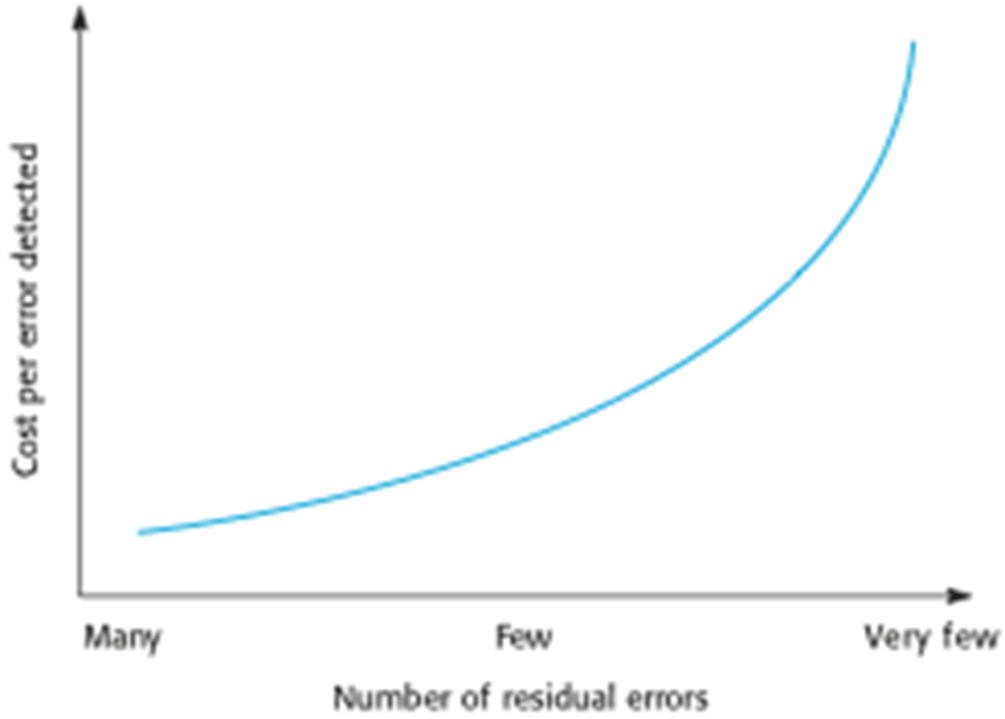
Pouzdanost softvera

- Uprkos upotrebi boljih inženjerskih tehnika, boljih programskih jezika i kvalitetnijeg razvojnog proces koji su doveli do značajnih napredaka u pouzdanosti softvera, padovi sistema se i dalje događaju.
- U nekim slučajevima ovi padovi izazivaju samo manje neugodnosti za korisnike sistema, dok u drugim mogu dovesti do gubitaka ljudskih života ili značajnih ekonomskih gubitaka.
- U kritične sistemi koji zahtevaju visok stepen pouzdanosti spadaju:
  - Medicinski sistemi
  - Telekomunikacioni sistemi i sistemi za električno napajanje
  - Sistemi za kontrolu letova

Tehnike za postizanje veće pouzdanosti sistema

- **Izbegavanje grešaka**
  - Prilikom projektovanja i implementacije softvera koriste se pristupi koji pomažu da se izbegnu greške i da se tako smanji broj potencijalnih uzroka koji mogu dovesti do pada sistema prilikom kasnije upotrebe.
- **Otkrivanje i korekcija grešaka**
  - Verifikacija i validacija služe za otkrivanje i otklanjanje grešaka iz programa, pre nego što se on pusti u upotrebu.
- **Tolerancija ne greške**
  - Sistem je projektovan tako da greške ili neočekivana ponašanja sistema koja se ispolje za vreme rada budu obrađeni tako da ne dođe do pada sistema.

Povećanje troškova prilikom otkrivanja preostalih grešaka



Redundantnost i raznovrsnost

- **Redundantnost** podrazumeva da su dodatne, ponovljene komponente uključene u sam sistem i da možemo da pređemo na njihovu upotrebu ukoliko dođe do otkazivanja primarnih komponenti.
- **Raznovrsnost** se odnosi na to da su dodatne komponente različitih tipova, što značajno smanjuje verovatnoću da će otkazati na isti način.
- Međutim, redundantnost i raznovrsnost dovode do dosta komplikovanijih sistema, kao i do većih ulaganja u pisanje koda i održavanje. U složenim sistemima teže je i pronaći greške.
- Kao posledica toga, neki zastupaju pristup da treba izbegavati ove tehnike, a umesto toga pisati što jednostavniji kod, uz izuzetno rigoroznu verifikaciju i validaciju.

Primeri redundantnosti i raznovrsnosti

- **Redundantnost**: U slučajevima u kojima je dostupnost sistema neophodna, koriste se dodatni serveri, tako da je omogućeno da se automatski pređe na njih ukoliko primarni server padne.

- **Raznovrsnost**: Kako bi se zaštitili od spoljnjih napada, serveri su obično različitih tipova, a često koriste i različite operativne sisteme.

Pouzdani procesi

- Za proizvodnju pouzdanih sistema važno je koristiti pouzdane procese. To su pravilno usvojeni i dokumentovani procesi, čijim se pridržavanjem postiže kvalitet i pouzdanost, kao i minimalan broj grešaka, a samim tim i smanjuje verovatnoća od otkazivanja sistem u toku rada.
- Dokaz da je pouzdan proces korišćen u implementaciji je često važan kako bi se naručioci sistema uverili da je primenjena najefikasnija softverska praksa prilikom razvoja.
- Za naručioce je od značaja i da je proces korišćen dosledno od strane svih učesnika u razvoju, kao i da se može primeniti u različitim razvojnim projektima.

Karakteristike pouzdanih procesa

Karakteristika	Opis
Dokumentabilan	Proces bi trebalo da ima definisan model, koji postavlja aktivnosti u procesu i dokumentaciju, koju treba proizvesti tokom ovih aktivnosti.
Standardizovan	Trebalo bi da postoji sveobuhvatan skup standarda i dokumentacije softverskog razvoja koji pokriva softverski proizvod.
Proverljiv	Proces bi trebalo da bude razumljiv i ljudima koji nisu učesnici u samom procesu, tako da i oni mogu da provere da li se prate odgovarajući standardi i da daju svoje predloge za unapređenje procesa.
Raznovrsnost	Proces bi trebalo da uključi redundantnost i raznovrsnost prilikom verifikacije i validacije.
Robustan	Proces bi trebalo da bude u stanju da se oporavi od padova u toku individualnih procesnih aktivnosti.

Primeri aktivnosti koje se mogu uključiti u pouzdan proces

- Razmatranje zahteva
- Upravljanje zahtevima
- Formalne specifikacije
- Modelovanje sistema
- Ispitivanje projekta i programa
- Statička analiza
- Planiranje i upravljanje testovima

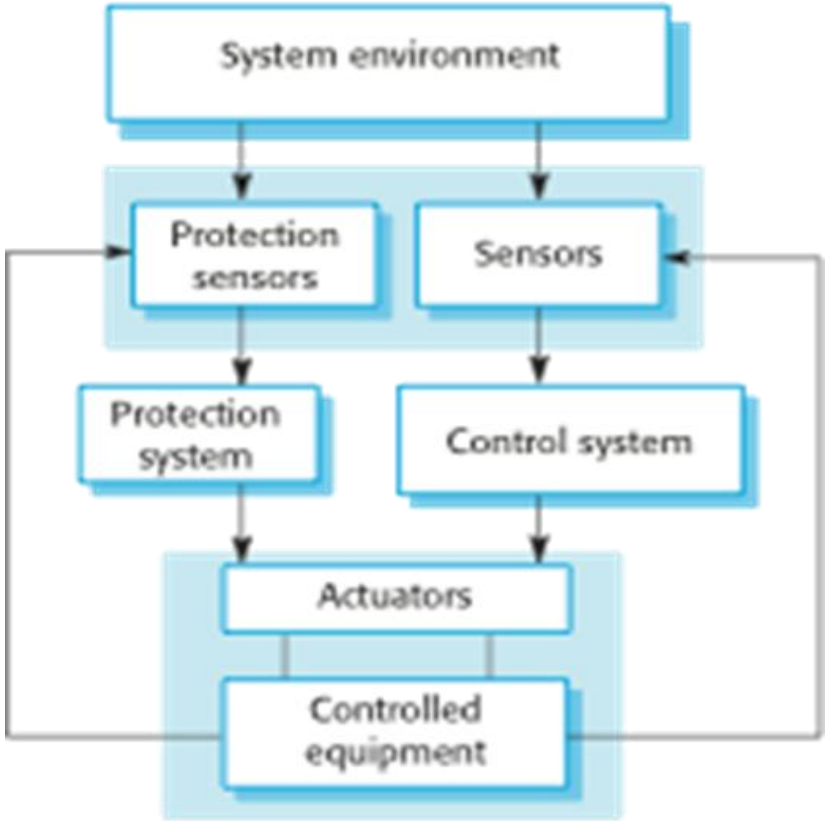
Pouzdane arhitekture sistema

- Pouzdan proces nije dovoljan da bi se stvorio pouzdan sistem. Potrebno je koristiti i pouzdanu arhitekturu sistema, pogotovo u situacijama u kojima je neophodna tolerancija na greške. Ovakvi sistemi su najčešće bazirani na redundantnosti i raznovrsnosti.
- Najjednostavnija realizacija pouzdane arhitekture je u repliciranim serverima, gde dva ili više servera imaju istu ulogu.
- Replicirani serveri obezbeđuju redundantnost, ali uglavnom ne i raznovrsnost. Hardver je obično identičan i svi koriste iste verzije softvera. Zbog toga, oni mogu da izađu na kraj sa softverskim ili hardverskim padovima koji su lokalizovani na jednoj mašini, ali ne i sa problemima u projektovanju koji dovode do toga da sve verzije softvera otkažu u isto vreme.

Sistemi za zaštitu

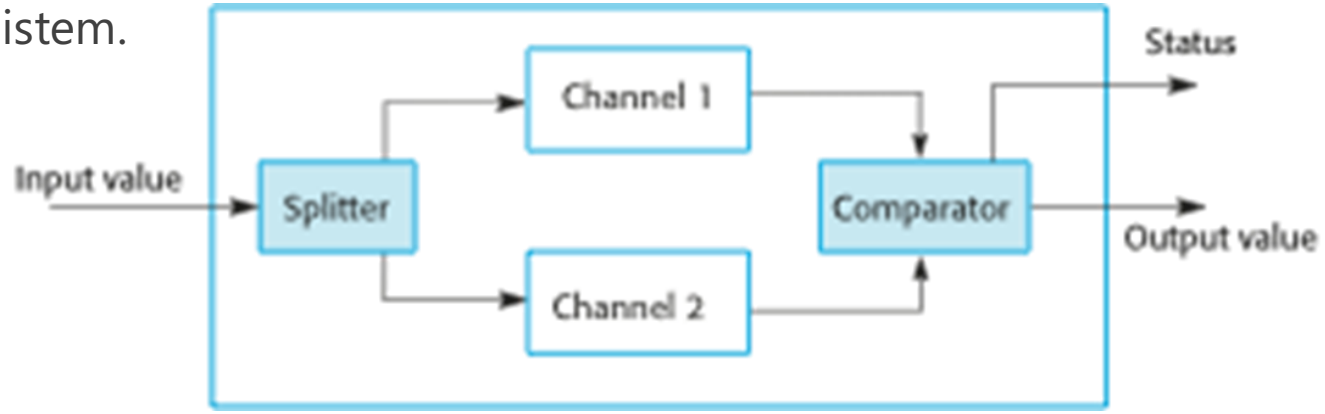
- Sistem za zaštitu je specijalizovan sistem koji je povezan sa nekim drugim sistemom (obično sistemom koji kontroliše neki proces). On obezbeđuje samo kritične funkcionalnosti neophodne da se kontrolisani sistem prevede iz potencijalno nebezbednog stanja u bezbedno.
  - Sistem koji zaustavlja voz ako prođe kroz crveno svetlo.
  - Sistem koji isključuje reaktor ako temperatura ili pritisak postanu previsoki.
- Sistem za zaštitu nadgleda i kontrolnu opremu i okruženje. Ukoliko je problem detektovan, izdaje komandu kojom se sistem isključuje ili preduzima neku drugu odgovarajuću meru.
- Prednost ovog sistema je što je značajno jednostavniji od sistema koji kontroliše zaštićeni proces. Njegova jedina uloga je da prati šta se dešava i da obezbedi da će sistem biti doveden u bezbedno stanje u vanrednoj situaciji. Zbog toga su moguća veća ulaganja u izbegavanje i otkrivanje grešaka.

Arhitektura sistema za zaštitu



Samoposmatrajuće arhitekture

- Samoposmatrajuća arhitektura je arhitektura sistema u kojoj je sistem projektovan tako da nadgleda sopstvene operacije i da preduzme odgovarajuće akcije ako se uoči problem.
- Ovo se postiže tako što se izračunavanja obavljaju na odvojenim kanalima, a zatim se porede rezultati. Ako su rezultati identični i dostupni u isto vreme, onda se zaključuje da sistem radi korektno. Inače, pretpostavlja se neuspeh. U tom slučaju, sistem baca izuzetak na izlaznu statusnu liniju, što dovodi do prebacivanja kontrole na drugi sistem.



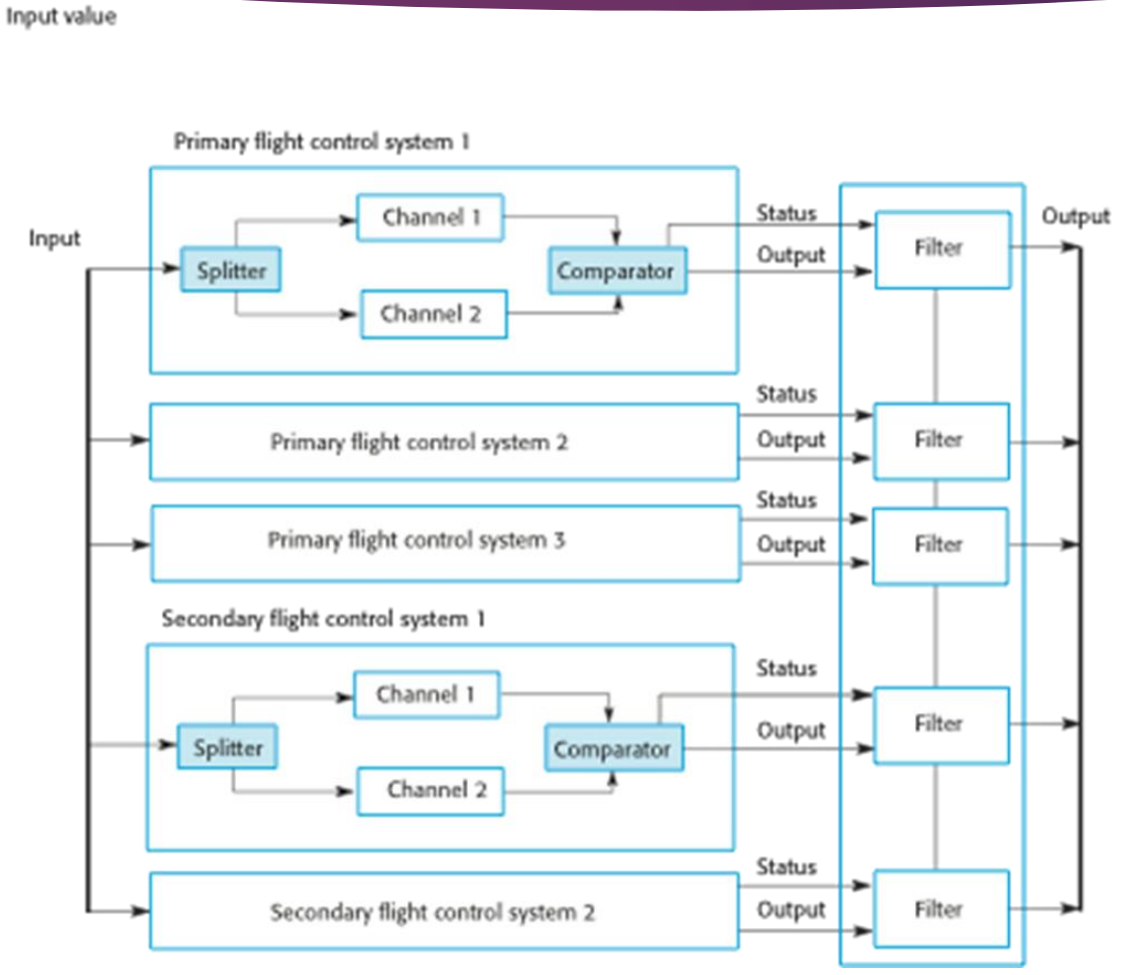
Samoposmatrajuće arhitekture (2)

- Efikasnost u detektovanju softverskih i hardverskih grešaka se postiže tako što:
  - Hardver koji se koristi u svakom kanalu je različt. U praksi, ovo može da znači da svaki kanal koristi procesore različitog tipa ili da je čipset koji se koristi dobijen od različitih proizvođača. Na taj način, smanjuje se verovatnoća da česte greške u projektovanju procesora utiču na izračunavanja.
  - Softver koji se koristi u svakom kanalu je različit. U suprotnom, ista softverska greška bi mogla da se javi na svim kanalima u isto vreme.
- Ova arhitektura se može koristiti u situacijama gde je neophodna tačnost u izračunavanju, ali ne nužno i raspoloživost. Ako se odgovori sa različitih kanala razlikuju, sistem se jednostavno isključuje. Za mnoge medicinske tretmane i dijagnostičke sisteme, pouzdanost je mnogo važnija od raspoloživosti, jer netačan odgovor sistema može voditi ka tome da pacijent dobije pogrešno lečenje.

Postizanje raznovrsnosti kod Airbus 340

- Primarni kompjuteri za kontrolu leta koriste različite procesore od sekundarnih sistema za kontrolu leta.
- Čipset koji se koristi u svakom od kanala primarnog i sekundarnog sistema je proizveden od strane različith proizvođača.
- Softver u sekundarnom sistemu za kontrolu leta obezbeđuje samo kritčne funkcionalnosti i zbog toga je manje složen od primarnog softvera.
- Softver u svakom od kanala u primarnom i sekundarnom sistemu je razvijen korišćenjem različitih programskih jezika i od strane različitih timova.
- Različiti programski jezici su korišćeni za primarni i sekundarni sistem.

Airbus arhitektura sistema za kontrolu letova



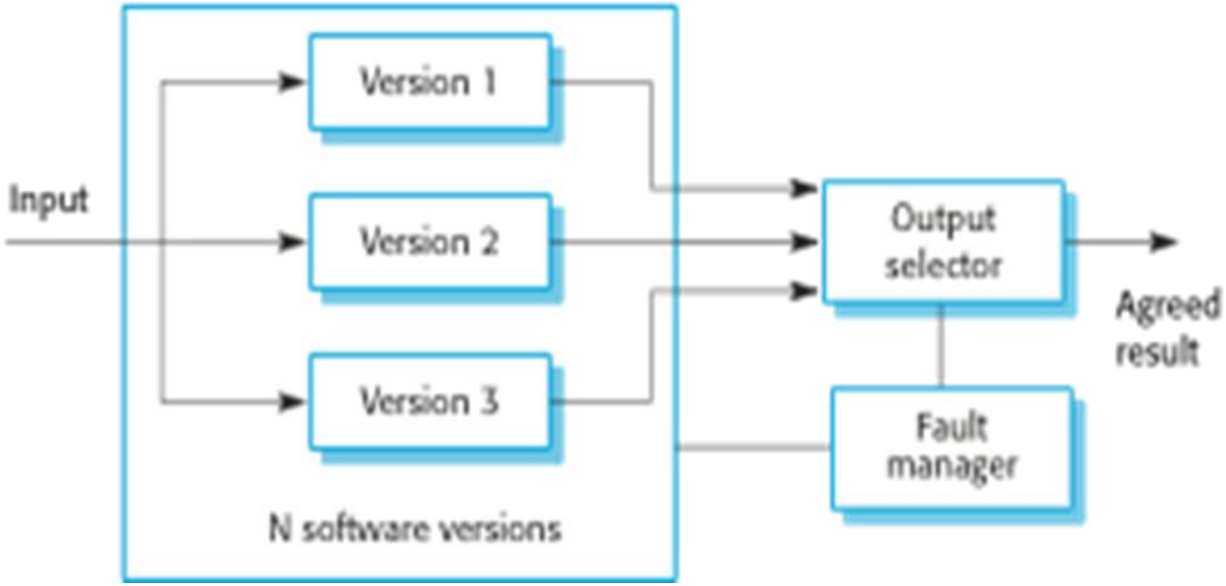
N-verziono programiranje

- Pojam multiverzionog programiranja je izveden iz hardverskih sistema i pojma trostruke modularne redundantnosti (TMR).
- Kod TMR sistema, hardverska jedinica je umnožena 3 (nekada i više) puta. Izlaz svake jedinice se prosleđuje izlaznom komparatoru koji se obično implementira kao sistem za glasanje. On poredi pristigle podatke i ako su dva ili više jednaki, onda ta vrednost predstavlja izlaz. Izlaz jedinice koja nije proizvela isti rezultat kao ostale se ignoriše. Upravljač greškama može da pokuša da popravi tu jedinicu, a ako ne uspe, sistem se automatski rekonfiguriše tako da tu jedinicu stavi van upotrebe.
- Ovaj pristup počiva na činjenici da je većina hardverskih padova rezultat otkazivanja neke komponente, a ne greške u projektovanju.

N-verziono programiranje (2)

- Triple modular redundancy

- N-version programming



N-verziono programiranje (3)

- Slično TMR-u, ovakav pristup se može primeniti na softverske sisteme, koji bi trebalo da budu otporni na greške, i to tako što se *N* različitih verzija softvera izvršava paralelno.
- Isti softverski sistem je implementiran od strane više timova. Ove verzije se izvršavaju paralelno, na različitim računarima i njihovi izlazi se porede korišćenjem sistema za glasanje. Nekonzistentni izlazi ili izlazi koji nisu pristigli na vreme se odbacuju.
- Uprkos mnogim prednostima, cena razvoja je visoka, jer je potrebno unajmiti više timova za obavljanje istog posla.
- Zbog toga se ovaj pristup koristi u sistemima gde nije praktično obezbediti sistem za zaštitu koji bi štitio od kritičnih, bezbednosnih padova.

Raznovrsnost softvera

- Raznovrsnost softvera podrazumeva da su različite implementacije iste specifikacije međusobno nezavisne. Zbog toga ne bi trebalo da imaju iste greške, a samim tim ni da otkažu na isti način, u isto vreme.
- Ovo zahteva da različiti timovi ne komuniciraju tokom razvojnog procesa.
- Kompanija koja sprovodi razvoj može da postavi neke eksplicitne politike raznovrsnosti koje treba primenjivati, npr:
  - Korišćenje različitih metoda projektovanja (npr. jedan tim treba da razvije objektno orjentisan model, a drugi funkcionalno orjentisan model)
  - Upotreba različitih programskih jezika
  - Upotreba različitih alata i razvojnih okruženja
  - Implementacija različitih algoritama u pojedinim delovima projekta

Potencijalni problemi sa raznovrsnosti softvera

- U idealnoj situaciji, različite verzije sistema ne bi trebalo da imaju međuzavisnosti i zbog toga bi se njihovi evenutalni padovi odvijali na potpuno drugačiji način.
- U praksi, postizanje potpune nezavisnosti je nemoguće, a za to postoji nekoliko razloga:
  - Članovi različitih timova su često iz iste sredine i obično imaju slično obrazovanje i poglede na problem.
  - Ako su zahtevi netačni, to će se odraziti u svakoj implementaciji.
  - Kod kritičnih sistema, specifikacija bi trebalo da bude dovoljno detaljna i da ne ostavlja prostora za bilo kakvu interpretaciju od strane razvojnog tima.
- Jedan od načina da se smanji verovatnoća čestih grešaka u specifikaciji je da se ona definiše na različite načine (npr. formalna, model sistema zasnovan na stanjima, napisana prirodnim jezikom itd.)

U praksi

- U eksperimentalnoj analizi, Hatton (1997) je zaključio da je sistem sa tri kanala između pet i devet puta pouzdaniji od sistema sa jednim kanalom.
- Međutim, postavlja se pitanje da li je ovaj metod vredan dodatnih troškova razvoja. Za mnoge sisteme, troškovi nisu opravdani i verzije sa jednim kanalom mogu biti dovoljno dobre.
- Ovaj pristup može naći primenu u situacijama gde su troškovi neuspeha veoma veliki (npr. sistemi za kosmičke letove).
- Čak i u tim situacijama može biti dovoljno obezbediti jednostavan rezervni sistem, sa ograničenom funkcionalnošću, dok se primarni sistem ne popravi.

Pouzdano programiranje

- Iako postoji veliki broj programskih jezika, neke smernice za bezbedno programiranje se mogu koristiti u svakom od njih.
  1. Ograničavanje vidljivosti informacija u programu
  2. Provera validnosti svih ulaznih podataka
  3. Obezbeđivanje obrade svih izuzetaka
  4. Minimizovanje upotrebe konstrukcija sklonih greškama
  5. Obezbeđivanje sposobnosti restartovanja
  6. Provera granica niza
  7. Uključivanje čekanja prilikom pozivanja spoljašnjih komponenti
  8. Imenovanje svih konstanti koje predstavljaju vrednosti u stvarnom svetu

1. Ograničavanje vidljivosti informacija u programu

- Komponente u programu bi trebalo da imaju pristup samo podacima neophodnim za svoju implementaciju. Na ovaj način, informacije ne mogu biti oštećene od strane komponenti koje ne bi trebalo da ih koriste.
- Ako interfejs ostane isti, reprezentacija podataka može da se menja, tako da ne utiče na druge komponente u sistemu.
- Ovo se postiže korišćenjem apstraktnih tipova podataka, čija je interna struktura i reprezentacija promenljivih tog tipa skrivena.
- Sav pristup takvim podacima se obavlja preko obezbeđenih operacija.

2. Provera validnosti svih ulaznih podataka

- Svi programi uzimaju ulazne podatke iz svog okruženja i obrađuju ih. Specifikacije prave pretpostavke o ovim podacima koje se odražavaju na stvarnu upotrebu.
- Međutim, specifikacija sistema često ne definiše koje akcije treba preduzeti ako su ulazni podaci neispravni.
- Greške mogu nastati kad korisnici unesu neispravne podatke, ali i kada neispravni podaci dolaze od senzora ili drugih sistema.
- Takođe, neki maliciozni napadi na sisteme se oslanjaju na namerno unošenje nekorektnih podataka.

2. Provere validnosti (2)

- **Provera opsega**
  - Proverava se da li je ulazni podatak u dozvoljenom opsegu (npr. verovatnoća koja se unosi bi trebalo da bude u opsegu od 0.0 do 1.0).
- **Provera veličine podatka**
  - Proverava se da li podaci ne prelaze neki zadati broj karaktera (npr. ime osobe verovatno neće imati više od 40 karaktera).
- **Provera reprezentacije podataka**
  - Proverava se da li ulazni podaci sadrže neke nedozvoljene karaktere (npr. imena ne sadrže cifre).
- **Opravdane provere**
  - Kada je ulazni podatak deo neke serije podataka i kada znamo nešto o vezama između članova serije (npr. potrošnja električne energije za jedno domaćinstvo u istom periodu godine).

3. Obezbeđivanje obrade svih izuzetaka

- Tokom izvršavanja programa, greške ili neočekivani događaji su neizbežni.
- Greške ili neočekivani događaji koji se javljaju tokom izvršavanja programa nazivaju se izuzeci.
- Kada se izuzetak pojavi, mora biti obrađen od strane sistema. Obrada se može vršiti u okviru samog programa ili se kontrola može prebaciti mehanizmu za rukovanje izuzecima.
- Programski jezici, kao što su Java, C++ i Ada, poseduju konstrukcije za rukovanje izuzecima, tako da nisu potrebne dodatne provere uslova, koje značajno komplikuju kod.

3. Obezbeđivanje obrade svih izuzetaka (2)

- Mehanizmi za rukovanje izuzecima obično rade jednu ili više od sledeće tri stvari:
  - Signaliziraju komponenti višeg nivoa da se izuzetak pojavio i obezbeđuju informacije o tipu izuzetka.
  - Izvršavaju neki alternativni proces. Dakle, rukovalac izuzetkom preduzima neke akcije kako bi se oporavio od problema.
  - Predaju kontrolu aktivnom sistemu za podršku koji upravlja izuzecima. Ovo je obično podrazumevana akcija kada se greška javi (npr. prilikom prekoračenja numeričke vrednosti). Uobičajena akcija ovog sistema je da zaustavi dalju obradu.

4. Minimizovanje upotrebe konstrukcija sklonih greškama

- Greške u programima, a samim tim i mnogi padovi programa su najčešće posledica ljudske greške.
- Ljudi će uvek praviti greške, ali je postalo jasno da su neki pristupi u programiranju skloniji greškama od drugih.
- Zbog toga, pojedine konstrukcije programskih jezika ili programerske tehnike treba izbegavati ili ih bar koristiti što je manje moguće.

4. Konstrukcije sklone greškama (2)

- **Bezuslovno grananje (go-to naredba)** – izbačena iz velikog broja modernih jezika, njena upotreba vodi tzv. špageti kodu, teškom za razumevanje i debagovanje.
- **Brojevi u pokretnom zarezu** – značajno neprecizna reprezentacija, što može predstavljati veliki problem prilikom poređenja ovakvih brojeva.
- **Pokazivači** – ukoliko referišu na pogrešnu lokaciju u memoriji, mogu oštetiti podatke. Takođe otežavaju proveru granica i razumevanje drugih struktura.
- **Dinamička alokacija memorije** – opasnost alokacije memorije prilikom izvršavanja programa je u tome što se može desiti da se ne dealocira korektno, što dovodi do curenja memorije. Ovakve pojave se teško detektuju.

4. Konstrukcije sklone greškama (3)

- **Paralelizacija** – kod procesa koji se izvršavaju konkurentno mogu postojati suptilne vremenske zavisnosti između njih. Vremenski problemi se obično ne mogu detektovati ispitivanjem programa.
- **Rekurzija** – greške kod rekurzije mogu dovesti do prekoračenja kapaciteta steka prilikom rekurzivnih poziva.
- **Prekidi** – prisiljavaju prelazak na određenu sekciju koda, bez obzira na trenutno stanje. Ovo može dovesti do prekida kritičnih operacija.
- **Nasleđivanje** – kod povezan sa objektom nije lokalizovan. Ovo otežava razumevanje ponašanja objekta i dovodi do toga da se greške teže otkrivaju.

4. Konstrukcije sklone greškama (4)

- **Aliasi** – upotreba više od jednog imena za referisanje istog objekta. Teško je pratiti promene stanja objekta kada postoji više imena koja treba razmotriti.
- **Neograničeni nizovi** – prekoračenje bafera se može javiti ako se piše van granica bafera koji je implementiran kao niz.
- **Podrazumevana obrada unosa** – akcija koja se izvršava nezavisno od ulaznih podataka. Ovo je sigurnosni propust koji napadač može iskoristiti da ubaci neočekivane ulazne podatke koji neće biti odbačeni od strane sistema.

5. Obezbeđivanje sposobnosti restartovanja

- Mnogi organizacioni informacioni sistemi su bazirani na kratkim transakcijama. Oni su dizajnirani tako da se promene u bazi podataka finiširaju tek nakon što se svi ostali procesi uspešno završe. Ukoliko dođe do greške, baza neće biti ažurirana.
- Međutim, postoje i sistemi koji obuhvataju duge transakcije, od nekoliko minuta ili nekoliko sati, kao što su npr. neka zahtevna matematička izračunavanja.
- Ukoliko sistem padne tokom trajanja ovakve transakcije, sav dotadašnji posao može biti izgubljen.
- Zbog toga je potrebno obezbediti sposobnost restartovanja koja je zasnovana na čuvanju kopija podataka koje se prikupljaju ili generišu tokom obrade. Ova sposobnost dozvoljava da se sistem ponovo pokrene korišćenjem ovih kopija, umesto da započinje obradu ispočetka.

6. Provera granica niza

- U nekim programskim jezicima, kao što je C, moguće je adresirati lokaciju u memoriji van granica deklarisanog niza.
- Jezici kao što je Java uvek proveravaju da li je indeks niza na koji se unosi podatak u okviru niza.
- Razlog zašto neki jezici ne uključuju automatsku proveru granica niza je dodatni trošak koji se zahteva svaki put kada se pristupa nizu, što značajno usporava izvršavanje programa.
- Ipak, nedostatak provere granica dovodi do sigurnosnih propusta, kao što je prekoračenje bafera i u opštem slučaju, do pada sistema.

7. Uključivanje čekanja prilikom pozivanja spoljašnjih komponenti

- Kod distribuiranih sistema, komponente sistema se izvršavaju na različitim kompjuterima i komunikacija se obavalja preko mreže. Ukoliko komponenta A čeka odgovor od komponente B, koji ne stiže usled pada komponente B, komponenta A ne može da nastavi sa radom i čekaće zauvek.
- Kako bi se ovo izbeglo, potrebno je uvesti period čekanja koji nam dozvoljava da definišemo za koliko vremena očekujemo odgovor. Ukoliko ne dobijemo odgovor u predviđenom periodu, pretpostavlja se da je došlo do pada sistema i preuzima se kontrola od pozvane komponente. Sistem dalje preduzima određene akcije kako bi se oporavio.

8. Imenovanje svih konstanti koje predstavljaju vrednosti u stvarnom svetu

- Konstantama koje predstavljaju vrednosti u stvarnom svetu (kao što je npr. stopa poreza) bi trebalo dati imena, umesto da se koriste njihove numeričke vrednosti.
- Manje je verovatno pogrešiti i uneti pogrešnu vrednost kada se koristi ime, nego kada se koristi numerička vrednost.
- Takođe, kada se vrednost promeni, nema potrebe pregledati ceo kod kako bismo našli gde je sve vrednost korišćena, već je dovoljno samo promeniti vrednost na mestu gde je konstanta deklarisana.

KRAJ

Hvala na pažnji!