Kako rade debageri

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Luka Kalinić, 1058/2018
luka.kalinic95@gmail.com

24. novembar 2018

Sažetak

Debagovanje je postupak pronalaženja i rešavanja defekata ili problema unutar programa koji sprečava optimalan rad softvera ili sistema. Taktike debagovanja uključuju: interaktivno debagovanje (engl. *interactive debugging*), analizu kontrole toka (engl. *control flow analysis*), testiranje na nivou jedinice (engl. *unit testing*), integraciono testiranje (engl. *integration testing*), analize log datoteke (engl. *log file analysis*), kontrolisanje na nivou aplikacije ili sistema, memory dumps i profajliranje.

Sadržaj

1 Uvod

Izrazi bag (engl. *bug*) i debagovanje (engl. *debugging*) popularno se prepisuju admiralki Grejs Hoper (*Grace Hopper*) u periodu četrdesetih godina dvadesetog veka. Dok je na univerzitetu Harvard radila na računaru *Mark II*, njeni saradnici su otkrili moljca zaglavljenog u releju čije prisustvo je ometalo rad sistema. Nakon toga, Grejs je primetila da saradnici vrše debagovanje sistema. Ipak, izraz bag, u smislu tehničke greške, datira još od 1878. godine kada je već bio deo inženjerskog žargona. Tada je taj izraz korišćen u hardverskom inženjerstvu da bi opisao mehaničke kvarove. U pismu saradniku 1878. godine, Tomas Edison (*Thomas Edison*) je napisao sledeće:

> *Upravo tako je bilo u svim mojim izumima. Prvi korak je intucija i dolazi u prasku. Nakon toga, pojavljuju se poteškoće. Tada se bagovi, ono kako nazivamo male greške i poteškoće, pokažu i potrebni su meseci intenzivnog praćenja, izučavanja i rada pre nego što je moguće postići komercijalni uspeh. U suprotnom, neuspeh je zagarantovan.*

Izraz debagovanje je takođe korišćen u aeronautici pre nego što je ušao u sferu računara. U jednom intervjuu Grejs Hoper je napomenula da ona nije skovala termin bag, već se moljac uklopio u već postojeću terminologiju. Direktor projekta atomske bombe iz drugog svetskog rata Menhetn (engl. *Manhattan*), Džulius Robert Openhajmer (*Julius Robert Oppenheimer*), koristio je pomenuti izraz u pismu doktoru Ernestu Lorensu (*Ernest Lawrence*) na Univerzitetu Berkli, povodom regrutovanja dodatnog tehničkog osoblja. U engleskom rečniku *Oxford*, u značenje reči debag citiran je termin debagovanje korišćen kao referenca za testiranje motora aviona u članku Žurnala kraljevskog aeronautičkog društva iz 1945. godine.

Hoperova buba je pronađena 1947. godine, ali taj termin nije bio usvojen od strane programera do ranih pedesetih godina. Članak napisan od strane S. Gil [1] iz 1951. godine je prva temeljna diskusija o programerskim greškama, ali ne koristi termine bag i debagovanje. Prvi pisani trag korišćenja tih termina pronađen je u naučim radovima iz 1952. godine, od kojih su u dva rada bili korišćeni pod navodnicima. Do 1963. godine debagovanje je bio izraz koji je bilo prihvatljivo spomenuti bez posebnog objašnjenja.

2 Složenost

Kako su softverski i elektronski sistemi postali generalno kompleksniji, različite uobičajene tehnike debagovanja su proširene sa sve više metoda za detektovanje anomalija, procenu uticaja i zakazivanje softverskih ažuriranja ili kompletnih ažuriranja sistema. Reči kao što su anomalija i nesaglasnost takođe mogu biti korišćene kao neutralniji termini da bi se izbeglo korišćenje reči greška ili bag tamo gde može biti implicirano da se sve greške ili bagovi moraju popraviti po svaku cenu. Umesto toga, može se izvršiti takozvana procena uticaja kako bi se utvrdlo da li će izmene neophodne za uklanjanje anomalije biti ekonomične za sistem ili će možda planirano novo izdanje učiniti tu izmenu nepotrebnom. Nisu svi problemi od ključnog značaja za funkcionisanje sistema. Takođe je važno izbeći situacije gde bi neka promena mogla izazvati veću uznemirenost korisnika na duge staze nego postojanje konkretne anomalije. Zasnivanjem odluka

o prihvatljivosti nekih anomalija može se izbeći kultura nula defekata (sistema koji je potpuno bez defekata), gde može doći do pojave da su ljudi spremni da poreknu postojanje problema samo da bi rezultat naizgled delovao kao da je potpuno oslobođen istih. S obzirom na kolateralne probleme, kao što je procena uticaja troškova u odnosu na benefit, doći će do ekspanzija tehnika debagovanja tako da se utvrdi učestanost anomalija da bi se procenio njihov uticaj na celokupan sistem.

3 Alati

Debagovanje ima širok spektar kompleksnosti - od popravke sitnih grešaka do izvođenja dugotrajnih i iscrpljujućih zadataka poput prikupljanja podataka, analize i zakazivanja ažuriranja. Veština programera predstavlja važan faktor njegove mogućnosti da ukloni problem, ali nivo težine debagovanja sofvera varira u zavisnosti od kompleksnosti sistema. Takođe zavisi do neke mere i od toga koji programski jezik se koristi, kao i od dostupnih alata kao što su debageri.

Debageri su softverski alati koji omogućavaju programeru da prati izvršavanje programa, da ga zaustavi, resetuje, postavi tačke prekida (engl. *brakepoint*) i promeni vrednosti u memoriji. Izraz debager može da se odnosi i na osobu koja vrši proces debagovanja.

3.1 Savremeni programski jezici i debagovanje

Generalno, savremeni programski jezici kao što je Java čine proces debagovanja lakšim jer u sebi sadrže alate kao što su rukovanje izuzecima (engl. *exception handling*) i *type checking* koji čine da pravi uzroci neočekivanog ponašanja programa budu lakše uočljivi. U programskim jezicima ranijih generacija kao što su *C* ili *Asembler*, bagovi mogu da izazovu tihe probleme kao što su oštećenje podataka u memoriji (engl. *memory corruption*) i često je teško doći do izvora inicijalnog problema. U takvim slučajevima, neophodno je primeniti alate koji služe za debagovanje memorije.

3.2 Alati za proveru semantike koda

U nekim situacijama, softverski alati za opštu namenu, koji su po prirodi jezično specifični, mogu biti vrlo korisni. Oni se nalaze u obliku statičkih alata za analizu kodova. Ovi alati rade tako što traže vrlo specifičan set poznatih problema, od kojih su neki uobičajeni a neki retki, unutar izvornog koda. Problemi detektovani na ovakav način retko kad bi mogli biti uhvaćeni od strane kompajlera ili interpretatora jer oni vrše proveru semantike koda, ali ne i sintakse. Neki alati mogu da detektuju i preko 300 jedinstvenih problema. Moguće je naći razne alate, komercijalne kao i besplatne, u različitim jezicima. Ovi alati mogu biti izuzetno korisni kada na projektu radi više programera. Tipičan primer detektovanog problema bi bio derefenciranje promenljive koje bi se pojavilo pre nego što se promenljivoj dodeli vrednost. Takođe, kao drugi primer možemo navesti alate koji izvode proveru jakog tipa (engl. *strong type*) u slučaju kada sam jezik to ne zahteva. Stoga su takvi alati bolji u pronalaženju mogućih grešaka u odnosu na stvarne greške. Zbog toga takvi alati imaju reputaciju lažnih pozitiva. Rani primer takvog ponašanja bi bio stari *Unix lint* program.

3.3 Debagovanje hardvera

Za debagovanje elektronskog hardvera kao i softvera niskog nivoa, često se koriste instrumenti kao što su osciloskopi, logički analizatori (engl. *logic analyzers*) i sklopni emulatori (engl. *in-circuit emulators*), pojedinačno ili u kombinaciji.

4 Proces debagovanja

Prilikom kompilacije projekta postoje razna podešavanja i opcije koje se mogu koristiti a koje služe za preciznije navođenje načina prevođenja projekta. Jedna od osnovnih opcija je mogućnost prevođenja u *release* ili *debug* modu.[4] U daljem tekstu će biti objašnjeno prevođenje u debug modu jer je ono od velikog značaja za sam proces debagovanja programa.

4.1 Prevođenje programa u debug modu

Debug mod je prevođenje programa u izvršnu verziju namenjenu programeru (za razliku od release moda koji prevodi program u izvršnu verziju namenjenu krajnjem korisniku). Ovo prevođenje obično isključuje optimizovanje radi lakšeg uparivanja izvornog i izvorišnog koda. Izvršavanje ovakvog koda može da bude manje efikasno (baš iz razloga što smo isključili optimizovanje). U izvršnu verziju se umeću podaci koji su potrebni za povezivanje izvornog i izvršnog koda, tj. omogućavaju debageru da precizno utvrdi koji deo koda se izvršava u datom trenutku. Zbog manjka optimizacije i dodatnih informacija, veličina izvršnog fajla je veća od veličine izvršnog fajla koji se dobije prevođenjem u release modu.

4.2 Osobine debagera

Debager može da započne proces i da ga prati i debaguje ili može da se nakači na proces koji se već izvršava. Debager omogućava izvršavanje programa instrukciju po instrukciju. omogućava postavljanje prekidnih tačaka (*engl. breakpoint*) i izvršavanje programa do tih tačaka prekida kao i praćenje promenljivih i stanja na steku prilikom izvršavanja (Poglavlje 5.1). Savremeni debageri omogućavaju i izmenu koda koji se izvršava i posmatranje efekta takvih izmena, debagovanje unazad, uslovne prekidne tačke i tačke posmatranja (*engl. watchpoints*). Primer ovakvog debagera je *Microsoft Visual Studio Debugger* [2]

4.3 Kako rade debageri

Kada se postavi prekidna tačka u programu sa željom da se na tom mestu zaustavi program, debager umetne na to mesto u softveru instrukciju prekida ili neku nevalidnu instrukciju. Kada se prilikom izvršavanja programa naiđe na ovu instrukciju desi se hardverski izuzetak koji uzrokuje prekid. Najpre se proveri da li je prekid u listi očekivanih prekida debagera (tj da li je u pitanju namerno zaustavljanje ili greška u originalnom kodu). Ukoliko je greška u originalnom kodu, onda se dopusti da se ta greška i izvrši i da program pukne. Ukoliko je u pitanju tačka prekida, prekid se prosledi na obradu debageru koji ga onda obradi tako što na tom mestu omogući uvid u sve vrednosti fizičkih registara procesa kao i u stanje memorije. Debager prikazuje pročitane informacije o procesu povezane sa informacijama o izvornom kodu koje se nalaze u programu umetnute

od strane kompajlera/linkera prilikom prevođenja programa. Ukoliko je u pitanju uslovna prekidna tačka, debager proverava uslov i u slučaju da uslov nije ispunjen, preskače se obrada prekida i samo se nastavlja dalje sa izvršavanjem procesa.

Kada programer poželi da nastavi sa izvršavanjem, debager zameni instrukciju prekida sa originalnom instrukcijom, izvrši je, zameni ponovo originalnu instrukciju instrukcijom prekida i ponovo prepusti kontrolu programu.

5 Tehnike debagovanja

5.1 Interaktivno debagovanje

Sistem za interaktivno debagovanje pruža programeru alate za testiranje i debagovanje programa. Danas postoji veliki broj sistema za interaktivno debagovanje koji se uveliko koriste. Bilo koji sistem za interaktivno debagovanje zahteva mogućnost kontrole toka izvršavanja programa. Postavljanjem prelomnih tačaka (*engl. brakepoints*) izvršavanje programa se pauzira, korišćenjem komandi debagera analizira se progres programa a zatim se ponovo nastavlja sa izvršavanjem programa. Moguće je postaviti i uslovne izraze koji se proveravaju tokom izvršavanja programa i ukoliko se uslovi ispune, izvršavanje programa se zaustavlja i vrše se analize. Nakon što se završi analiza programa, izvršavanje programa se nastavlja.

5.2 Print debagovanje

Print debagovanje (praćenje (engl. *tracing*)) je postupak praćenja *trace* iskaza ili print iskaza koji ukazuju na tok izvršavanja procesa. Ovo se ponekad zove i printf debagovanje zbog korišćenja printf funkcije u jeziku C. Ova vrsta debagovanja se pokreće komandom TRON u originalnim verzijama BASIC programskog jezika. TRON je akronim od reči *Trace On*. TRON uzrokuje da se brojevi redova svake BASIC komandne linije štampaju dok se program izvršava.

5.3 Daljinsko debagovanje

Daljinsko debagovanje (*engl. remote debugging*) je postupak debagovanja programa koji se izvršava na sistemu udaljenom od debagera. Da bi proces započeo, debager mora da se poveže sa udaljenim sistemom preko mreže. Tada debager može da kontroliše izvršavanje programa na udaljenom sistemu i da sakuplja informacije o njegovom stanju.

5.4 Post-mortem debagovanje

Post-mortem debagovanje je postupak debagovanja programa nakon njegovog prekida. Povezane tehnike često uključuju razne tehnike praćenja i/ili analizu snimljenog stanja radne memorije programa u momentu kada je došlo do prekida. [5] Tačan momenat prekida procesa može se ustanoviti automatski od strane sistema (npr. kada je proces završen zbog nekog odstupanja), preko instrukcija napisanih od strane programera ili eksplicitno od strane korisnika.

5.5 *Wolf fence* alogritam

Edvard Gaus (*Edward Gauss*) je opisao jednostavan ali vrlo koristan i sada slavan algoritam u članku iz 1982. godine:

> *Na Aljasci je jedan vuk. Kako da ga pronađete? Prvo izgradite ogradu kojom ćete prepoloviti državu na dva dela. Sačekajte da vuk počne da zavija, a onda utvrdite sa koje strane ograde se vuk nalazi. Ponavljajte proces samo na toj strani dok ne dođete do momenta kada možete videti vuka.*

Ovaj postupak je implementiran u *Git* kontroli verzioniranja tako što komanda *git bisect* koristi gore pomenuti algoritam da odredi koji *commit* je uveo određeni bag.

6 Debagovanje ugrađenih sistema

U suprotnosti sa dizajnom okruženja računarskog softvera opšte namene, primarna karakteristika ugrađenih okruženja (*engl. embedded environments*) je veliki broj različitih platformi koje su dostupne programerima (CPU arhitekture, vendora, operativnih sistema). Ugrađeni sistemi po definiciji nisu dizajnirani za opštu upotrebu. Tipično se razvijaju za pojedinačan zadatak ili manji broj zadataka i platforma se bira specifično da optimizuje tu primenu. Ne samo da ova činjenica otežava posao programerima ugrađenih sistema, već čini njihovo debagovanje i testiranje mnogo težim jer su različiti alati za debagovanje neophodni za različite platforme.

Uprkos izazovima koji postoje zbog pomenutih različitosti, neki alati su razvijeni komercijalno, a neki i kao istraživački prototipi. Oni svi koriste funkcionalnost dostupnu u jeftinim ugrađenim (*engl. embedded*) procesorima, *OCDM* (*On-Chip Debug Module*), čiji signali se šalju kroz standardni *JTAG* interfejs.

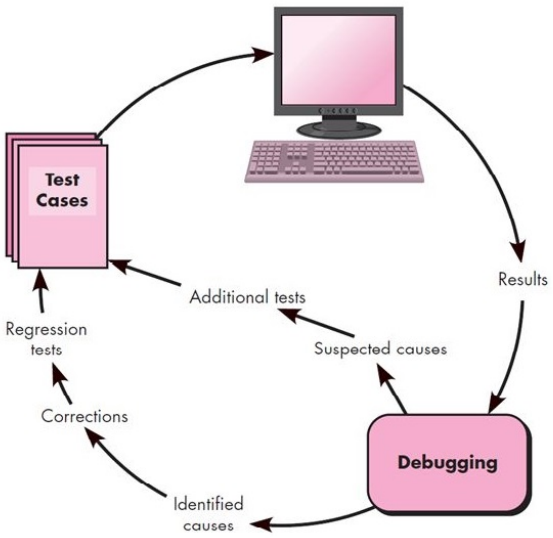
Pored uobičajenog zadatka identifikovanja bagova u sistemu, debagovanje ugrađenih sistema takođe prikuplja informacije o operativnom stanju sistema koje mogu biti korišćene u daljoj analizi - da se pronađe način kojim će se poboljšati performanse ili da se optimizuju druge važne karakteristike (potrošnja energije, pouzdanost, real-time odgovor i slično).

7 Anti-debagovanje

Anti-debagovanje [3] je implementacija jedne ili više tehnika unutar koda koje ometaju pokušaje obrnutog inženjeringa (*engl. reverse engineering*) ili debagovanja ciljanog procesa. Aktivno je korišćen od strane poznatih izdavača u sferi zaštite od kopiranja, ali je takođe korišćen i u malicioznim programima kako bi teže bili detektovani i eliminisani od strane antivirus programa. Te tehnike uključuju: tehnike bazirane na API-ju, tehnike bazirane na izuzecima, tehnike bazirane na blokovima procesa i niti itd.

8 Zaključak

Debagovanje je proces identifikovanja **pravog** problema i njegovog ispravljanja. Zaključujemo da je duplo teže debagovati program nego ga kodirati. (Slika 1) U širem kontekstu reči poput bag-a, greške, defekta ili propusta se mogu odnositi na bilo koju vrstu problema u bilo kojoj fazi procesa razvoja softvera. U užem smislu termin bag ili defekt se odnosi na grešku u kodu. Defekti nisu uvek loši za pojedinca. Uz pozitivan stav možemo da uvidimo prednosti poput poboljšanog razumevanja samog rada programa, učimo o sebi tj. kakve greške najčešće pravimo, stičemo uvid o čitljivosti i kvalitetu sopstvenog koda...



Slika 1: Složen proces debagovanja

Literatura

[1] S. Gill. The diagnosis of mistakes in programmes on the edsac. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 206(1087):538–554, 1951.

[2] Microsoft. anti-debugging. on-line at: https://code.visualstudio.com/docs/editor/debugging.

[3] Tyler Shields. anti-debugging, 2008. on-line at: https://www.veracode.com/blog/2008/12/anti-debugging-series-part-i.

[4] Pesch R. Shebs S. et al. Stallman, R. *Software Verification and Analysis*. Free Software Foundation, Boston, 2002.

[5] Stefan Wörthmüller. Post-mortem debugging, 2006. on-line at: http://www.drdobbs.com/tools/postmortem-debugging/185300443.