PT&SCHE

Introduction of part-time and short cycle studies in Serbia

Funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UNIVERZITET
METROPOLITAN
BEOGRAD

KI301 - KONSTRUISANJE SOFTVERA

Debagovanje

Lekcija 09

PRIRUČNIK ZA STUDENTE

KI301 - KONSTRUISANJE SOFTVERA

Lekcija 09

DEBAGOVANJE

- Debagovanje
- Poglavlje 1: Debugging-Uvod
- Poglavlje 2: Debugging tehnike
- Poglavlje 3: Alati za debugging
- Poglavlje 4: Vežba : Upotreba Java alata za debagovanje
- Zaključak

Uvod

L9:DEBAGOVANJE

Sadržaj:

- Debugging-Uvod
- Debugging tehnike
- Alati za debugging
- Debuggers
- Java alati za debugging - Vežba
- Debugging-pitanja - Vežba

Poglavlje 1

Debugging-Uvod

ANATOMIJA GREŠKE

Ima tri kategorije greške u programu.

Kada programeri pokušavaju da pronadju šta je pogrešno u njihovom programu, da pronadju grešku u programu, to se zove debugging, 'isterivanje buba'. Kaže se da program sadrži grešku, 'bubu'. Nemoguće je eliminisati sve moguće greške. Medjutim, jezički kompajleri omogućuju da se pronadju one očevidne greške tako da program može da počne da funkcioniše.

Anatomija greške**,** *Anatomy of a computer bug:*

Ima tri kategorije greške u programu,

-sintaksne greške, syntax errors,koje nastaju pogrešnim pisanjem komandi, npr PPRINT umesto PRINT, ili zaboravljanjem tačke-zareza, ;,

-izvršne greške, run-time errors, gde greška nastaje ako program naidje na nešto neočekivano, npr. ako korisnik ukuca kao ulazni podatak neki negativan broj umesto pozitivnog broja koji označava godište neke osobe,

-logičke greške, logic-errors, gde greške nastaju ako program radi ali ne radi ono što se očekuje od njega, pa proizvodi nepredvidljive rezultate

SINTAKSNE GREŠKE (SYNTAX ERRORS):

Najgore su sintaksne greške koje daju formalno dobru instrukciju ali ipak pogrešnu instrukciju koja naime nije željena instrukcija već na neki način promenjena.

Kompajleri mogu da detektuju pogrešno napisane komande, medjutim oni obično ne detektuju ako se neka varijabla pogrešno napiše, npr. sledeći program u Liberty Basic,

```
PROMPT ''HELLO''; RESULTS$

PRINT ''RESULTS''; RESULT$

END
```

Varijable RESULTS$I RESULT$su različite, pa program neće prikazati ono što mi očekujemo. Pored pogrešnog pisanja imena varijabli, može doći do mešanja velikih i malih slova u imenima varijabli, npr. varijable answer**$** i Answer**$** su potpuno različite, u jeziku C++.

Sintaksne greške obično se brzo i lako detektuju, jer daju pogrešnu instrukciju koja automatski sprečava program da radi uopšte. Najgore su sintaksne greške koje daju formalno dobru instrukciju ali ipak pogrešnu instrukciju koja naime nije željena instrukcija već na neki način promenjena. Onda program funkcioniše, ali e onako kako mi hoćemo. Neki programeri koriste kratka šifrovana imena varijabli, da bi smanjili mogućnost greške u kucanju imena varijabli, ali to nije dobro, jer onda program nije lako razumeti proveravati i menjati. Treba uvek proveriti imena varijabli, da li su pogrešno napisane ili sa pogrešnim velikim slovima.

RUN-TIME ERRORS

Izvršne greške se otkrivaju tako što se program uporno testira na najraznovrsnije ulazne podatke.

Izvršne greške, run-time errors, su prikrivene greške. Program može da funkcioniše korektno neko vreme korektno, I ove greške se pojave samo onda kada se pojave podaci koje programer nije nikada očekivao, npr. -1999 umesto 1999 za godinu rodjenja. Ove greške se nažalost otkriju tek kad zbog njih program zaustavi se da radi, tj slom, *crash*, programa.

Zato softverske kompanije testiraju njhove programme sa opštom publikom pomoću tzv. ‘beta’ verzije programa. Beta verzija programa je verzija koja je vrlo blizu prodaje programa, ali se želi ispitati u dodiru sa beta testers da koriste program da se vidi ima li greški koje su softveristi propustli da vide. Izvršne greške se otkrivaju tako što se program uporno testira na najraznovrsnije ulazne podatke. Npr. Za godinu rodjena treba probati ne samo 1999, već i 60.000, i 0, i -9.989.

KORAČANJE KROZ PROGRAM I INTERAKTIVNI DEBAGER

Stepping predstavlja izvršavanje programa linija po linija, i posmatranjem kako program radi.

Logičke geške su najsakrivenije, i najteže za otkrivanje tj dijagnozu. Program funkcioniše ali daje nepredvidljive tj. neočekivane rezultate a mi neshvatamo zašto. Da bi se našla logička greška, mora se program proveravati linija po linija da bi se utvrdilo šta nedostaje, ili šta je pogrešno u logici postojećih instrukcija. Pošto logičke greške se teško pronalaze, debuggers tj. "debageri" omogućuju specijalnedebugging, dijagnostičke, mogućnosti, i dva glavna načina da se pronadju logičke greške su:

stepping, koračanje kroz program,

i watching, razgledanje prrograma.

Stepping predstavlja izvršavanje programa linija po linija, i posmatranjem kako program radi. U momentu kada se otkrije da program radi nešto pogrešno, vi onda znate tačno koja linija u programu pravi grešku. Npr. za koračanje kroz neki program u Liberty BASIC, sledeće instrukcije se koriste:

Debug, Step Into, Step Out, Itd.

Umesto koračanja linija po linija kroz ceo program, od početka programa, što može biti nepotrebno, posebno ako se ima ideja šta može bitim pogrešno, onda se može program koračati kroz program ne od početka već neke izabrane tačke, npr. sredina ili pri kraju programa. Ovo se zove tracing trough your program.Kada se krača kroz program, onda se može posmatrati sadržaj pojedinih varijabli, i posmatrati kako se ove vrednosti varijabli menjaju, i koje linije izazivaju promene kojih varijabli. Ovo je posmatranje varijabli, tj.watching your variables.

TEHNIKA ŠTAMPANJA (“THE WRITE TECHNIQUE”)

For small programs we can use printing commands to test our code; for large programs and pointers this technique is ineffective (“the WRITE technique”).

Za male programe može se koristiti "tehnika štampanja" tj. komande za štampanje **(printing commands) za testiranje programa; ali za velike programe ova tehnika nije efektivna (“the WRITE technique”).**
Za veće programe se koristi alat : "debugger". To je alat koji omogućuje da se program izvršava usporeno (to run in “slow motion”), tako da se može ispitivati kako program radi instrukcija po instrukcija, u cilju otkrivanja zašto program ima defekte tj. u cilju detekcije/ eliminacije izvršnih greški (run-time errors):

Pomoću debagera može se:

Izvršavati program na kontrolisan tj. usporen način , npr. program može da se zaustavi u nekim tačkama i da se izvršava instrukcija po instrukciju, i da se posmatraju promenbljive u programu i menjaju promenljive u programu kod zaustavljanja programa.

VIDEO VEŽBA

Video vežba - Debagovanje Java programa pomoću Eclipse alata (Eclipse Java Tutorial 9 - Debug Java Program)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

ZADACI ZA VEŽBU

Zadaci

- .Demonstrirati koračanje kroz program pomoću debagera?
- Demonstrirati zastojne tačke u programu pomoću debagera?
- Demonstrirati posmatranje varijabli pomoću debagera?

Poglavlje 2

Debugging tehnike

VAŽNOST DEBUGGING-A

Za mnoge programere, debugging je najteži deo programiranja. Kod nekih projekata dedefektovanje nosi 50% od ukupnog vremena razvoja softvera.

Dijagnostika i korigovanje greške („dedefektovanje“), debugging, je proces identifikacije uzroka tj korena neke greške, i korigovanje te greške. Za razliku od testiranja softvera, što je proces inicijalnog detektovanja greški. Kod nekih projekata dedefektovanje nosi 50% od ukupnog vremena razvoja softvera. Za mnoge programere, *debugging* je najteži deo programiranja. Medjutim , dedefektovanje se može načiniti mnogo lakšim, ako se koriste odredjene tehnike, koje će biti opisani u ovom poglavlju. Softverski defekti, *bugs*, su greške u softveru, greške softverista (*errors, defects, faults*) .

Dedefektovanje omogućuje dijagnozu defekata. Medjutim nisu svi programeri vešti u programiranju. Najbolji dijagnostički programeri mogu biti tri puta brži od onih najlošijih. Takodje kod korigovanja pronadjenih defekata, može često doći do pravljenja novih defekata. Dole je dato jedno poredjenje izmedju jednog najboljeg i jednog najlošijeg programera, za jedan slučaj gde je bilo 12 defekata u softveru. Ovo nisu statistički podaci, već samo jedan primer, ali ilustrativan primer.

_____Najbolji programer_____Najlošiji programer

Vreme dedefektovanja u minutima _____10_____25

Broj nepronadjenih defkata_____ 1_____5 od 10

Dodati novi defekti_____1_____5

Ovo govori da dijagnoza greške je disciplina kojoj treba posvetiti veliku pažnju, odnosno programeri treba da se obučavaju u toj oblasti. Nije dovoljno reći da se dijagnoza greški, može obaviti pomoću ubacivanja serije instrukcija štampanja u program, u cilju nalaženja greške.

NEEFIKASNI PRILAZI DIJAGNOZI GREŠKE

Neefikasni prilazi dijagnozi greške su npr. pronalaženje greške pogadjanjem, pomoću slučajno tj haotično izabranih instrukcija štampanja, i pomoću slučajnih tj haotičnih izmena u programu.

Medjutim i ako je dedefektovanje teško za programere, ono je veoma korisno i predstavlja i pogodnost tj priliku za nešto korisno, naime

- ako je napravljena greška u programu to je prilika da se nauči nešto o tom programu
- takodje to je prilika da se nauči o tipovima i uzrocima grešaka koje se prave
- prilika da se nauče tehnike dijagnoze greški
- i prilika da se nauče tehnike korigovanja greški, gde se traži tačna dijagnoza i lečenje uzroka a ne simptoma problema

Neefikasni prilazi dijagnozi greške su

- pronalaženje greške pogadjanjem, pomoću slučajno tj haotično izabranih instrukcija štampanja u programu, i pomoću slučajnih tj haotičnih izmena u programu
- izbegavanje da se udubi duboko u problem
- već se površno pokušava ispraviti problem

Dijagnoza i korigovanje geške, sastoji se od nalaženja greške i popravke greške. Medjutim , nalaženje greške i razumevanje greške je obično 90% posla.

NAUČNI PRILAZ ISTRAŽIVANJU

Ovde se objašnjava tzv. naučni prilaz istraživanju nekog fenomena koji treba ispitati.

Naučni prilaz istraživanju nekog fenomena se sastoji od:

prikupi podatke kroz višestruke eksperimente

formirati hipotezu na osnovu relevantnih podataka

dizajnirati eksperiment u cilju dokazivanja ili obaranja hipoteze

dokazati ili oboriti hipotezu

ponoviti po potrebi

NAUČNI PRILAZ DEBUGGING-U

Ovde se izlaže sistematski prilaz dijagnozi greške.

Ovaj naučni prilaz ima puno sličnosti sa sledećim prilazom dijagnozi i krigovanju greški (naučni prilaz) koji je efikasan i koji se sastoji od

stabilizacije greške, tj napraviti grešku predvidljivim

locirati izvor greške tj kvara

prikupiti podatke koji izazivaju defekt

analizirati podatke koji su prikupljeni

formirati hipotezu o defektu

napraviti plan kako dokazati ili oboriti hipotezu, ili preko testiranja ili preko provere izvornog koda

dokazati ili oboriti hipotezu pomoću plana iz prethodne tačke

popraviti defekt

testirati popravku

potražiti slične greške

DEBUGGING- PRIMER

Ovde posmatramo primer debugging-a.

Ove korake kod dijagnoze i popravke greški ćemo ilustrovati na jednom primeru.

Primer:

Pretpostavimo da imamo program za bazu podataka o zaposlenima, i da program ima povremenu grešku. Naime , program treba da printuje listu zaposlenih i njihove poreske obaveze u alfabetskom redosledu. Dole je dat deo izlaznih podataka iz programa, ali printovanje nije po alfabetskom redosledu:

.....

*Mihic*_____***$****10*

*Maric-Simic*_____***$****9*

.....

Greška je što je *Maric-Simic* na listi posle *Mihic* umesto ispred.

STABILIZACIJA GREŠKE:

Stabilizacija greške zahteva testiranje, i to traženje test-slučaja koji ne samo da proizvodi tu grešku, već sužavanje opsega test-slučajeva, tj utvrdjivanje opsega koji izaziva grešku.

Ako se defekt pojavljuje nepredvidljivo, onda je vrlo teško izvršiti dijagnozu. Ako se medjutim, neki povremeni defekt napravi da se pojavljuje predvidljivo, to je jedan d najvažnijih i izazovnijih zadataka kod dedefektovanja. Neka greška koja se javlja nepredvidljivo je obično ustvari greška kod inicijalizacije. Npr ako izračunavanje zbira u redu ponekad a ponekad loše, onda neka varijabla u vezi izračunavanja tog zbira nije inicijalizovana, već često startuje kao 0, ali ne uvek.

Stabilizacija greške zahteva testiranje, i to traženje test-slučaja koji ne samo da proizvodi tu grešku, već sužavanje opsega test-slučajeva, tj utvrdjivanje opsega koji izaziva grešku. Potrebno je pojednostaviti test-slučaj tako, da bilo koja promena u test-slučaju izaziva promenu ponašanja greške. Onda promenama u test-slučaju i posmatranjem promena u grešci, može se izvršiti dijagnoza problema. Da bi se pojednostavio test-slučaj, npr pretpostavi se da 10 faktora u kombinaciji utiče na grešku. Onda se pretpostavi hipoteza koji su faktori nebitni. Onda se promene ti nebitni faktori, i onda se ponovo vrši test. Ako se greška opet javlja, onda se ovi faktori mogu eliminisati, i dobija se pojednostavljen test-slučaj. Onda se pokušava dalje pojednostavljenje test-slučaja.

Primer:U prethodnom prmeru , kod prvog izvršavanja programa tabela nije bila po alfabetskom redosledu, ali kod drugog izvršavanja programa je bila po alfabetskom redosledu:

.....

*Maric-Simic_____**$9***

*Mihic_____**$10***

.....

Dalje, posle unošenja Jovic-Panic u tabelu, ponovo je pozicija pogrešna za ovaj uneti podatak. Medjutim posle ponovnog izvršavanja programa lista je u redu. Hipoteza je, da posle unošenja jednog novog podatka dolazi do greške, jer kod unošenja više novih podataka nije bilo ovih problema.

LOCIRANJE IZVORA GEŠKE

Ovde se objašnjava kako se proverava tačnost hipoteze o grešci.

Kod lociranja greške može se sumnjati da je defekt rezultat jednog pogrešnog parametra, koji je izvan granice. Onda se može varirati sumnljivi parametar, ispod granice, na granici, iznad granice, i proveriti da li je hipoteza tačna.

Primer

Proverom izvornog koda, po hipotezi da kod dodavanja jednog novog podatka u gornoj tabeli se pravi greška, ali kod dodavanja više podataka ne, nije se uspelo naći uzrok greške. Ponovo se vrši test, dodaje se novi podatak, ali sa drugim početnim slovom, npr.

.....

*Kadic_____**$20***

*Maric-Simic_____**$9***

*Mihic_____**$10***

.....

Medjutim, ovaj put je redosled u redu. Ovo znači da je gornja hipoteza o unošenju samo jednog novog podatka, pogrešna. Ali analizom test slučajeva, utvrdjuje se da prva dva dodavanja podataka su nazivi sa crtom u imenu, Jovic-Panic. Onda se pravi nova hipoteza da crta u imenu pravi probleme, tj da podprogram za sortiranje ne tretira korektno imena sa crtom.

POPRAVKA GREŠKE:

Kod korigovanja greške često se prave nove greške.

Teže je pronaći grešku nego korigovati grešku. Medjutim , kod korigovanja greške često se prave nove greške. Sledeće su sugestije za izbegavanje pravljenja novih greški kod korigovanja postojećih greški:

pre korigovanja potrebno je razumeti problem, toliko razumeti npr. da se može predvideti svbako pojavljivanje te greške

takodje, razumeti ceo program, tj kontekst problema, jer onda se može rešiti problem kompletno a ne delimično

potvrditi da je dijagnoza korektna, pre nego se pristupi korigovanju greške, tj proveriti ispravnost hipoteze greške, odnosno obara ostale hipoteze koje nisu tačne

memorisati originalni izvorni kod, u cilju komparacije sa novim izvornim kodom

korigovati problem, a ne samo simptom

menjati program samo ako ste sigurni da je ispravna promena, jer ako se program menja po sistemu probe i greške, nije uopšte efikasan metod

vršiti jednu promenu programa po jednu, a ne više njih istovremeno, jer je to vrlo komplikovano i izaziva nove greške
proveriti izvršene korekcije

POPRAVKA GREŠKE - PRIMER

Primer

Primer:

result[client] = result[client] + Amount[Number];

Ako npr se ustanovi da gornji zbir greši za 5.5 za client jednak 10, pogrešno bi bilo korigovati simptom, a ne uzrok, na sledeći način:

If(client ==10){

result[10] =result[10]+5

REZIME TEHNIKA ZA NALAŽENJE I KORIGOVANJE GREŠKI:

Rezimiramo tehnike za nalaženje i korigovanje greški.

Tehnike za korigovanje greški:

Razumeti prvo problem, pa ga onda korigovati

Razumeti ceo program, a ne samo problem

Potvrditi dijagnozu greške, Memorisati početni izvorni kod

Popraviti problem, a ne simptom

Praviti jednu po jednu promenu u programu

Proveriti popravku

TEHNIKE ZA NALAŽENJE GREŠKI

Tehnike za nalaženje greški su sledeće.

Tehnike za nalaženje greški su sledeće:

Koristiti sve raspoložive podatrke da se napravi hipoteza o grešci

Profiniti test-slučajeve koji proizvode grešku

Ispitivati pojedine jedinice izolovano od ostatka programa

Integraciju obaviti postepeno, u malim koracima

Proveriti tipične greške

Napraviti pauzu u traženju greške, da bi se slegli utisci

Definisati vreme raspoloživo pre primene krajnjih mera
Napraviti listu krajnjih mera

KRAJNJE MERE ZA DIJAGNOZU GREŠKE:

Ponekad je potrebno rimeniti krajnje mere, koje će svajkako rešiti problem dijagnoze greške.

Ponekad je potrebno rimeniti krajnje mere, koje će svajkako rešiti problem dijagnoze greške, ako se nije uspelo pomoću testova i hipoteza o grešci. Ove krajnje mere su

uraditi kompletnu proveru dizajna i kodiranja dela programa koji ne radi dobro

odbaciti deo programa koji ne radi dobro, i ponovo ga iz početka dizajnirati i kodirati

odbaciti ceo program i ponovo ga dizajnirati i kodirati iz početka

uraditi kompilaciju sa kompletnom informacijom o debugging

pooštriti testiranje softverskih jedinica, i testirati softverske jedinice u izolaciji

koristiti debugger da se provere široki opsezi test parametara

promeniti kompajler

opremiti program sa štampanjima, i prikazima

kompilaciju i izvršavanje programa obaviti negde drugde, u drugom okruženju

integrisati program u malim komadima, i kompletno testirati svaki taj komad

INDIVIDUALNA VEŽBA

Zadaci za individualnu vežbu.

1) Uraditi zadatke iz debugging, date na :

http://www.cs.usyd.edu.au/~jchan3/soft1001/jme/debugging/debugging_task.html

2) Pogledati pokazni primer na linku:
https://www6.software.ibm.com/developerworks/education/j-debug/j-debug-ltr.pdf

Poglavlje 3

Alati za debugging

KOMPAJLERI, LINKERI I BUILD TOOLS

Build Tools: Cilj ovog alata je da minimizira vreme potrebno da se izgradi program koristeći trenutne verzije izvornih programskih fajlova.

Kompajleri (compilers) konvertuju izvorni kod u izvršni kod.

S druge strane, "linker" povezuje "objektne fajlove" (object files) koje je generisao kompajler sa standardnim kodom potrebnim da se napravi izvršni program.

Build Tools: Cilj ovog alata je da minimizira vreme potrebno da se izgradi program koristeći trenutne verzije izvornih programskih fajlova. Dakle, kod modifikacije nekih fajlova u okviru velikih programa sa puno fajlova, ovi alati omogućuju brzo i pouzdano izvođenje compile-link-run ciklusa. Naime ako menjate neke fajlove u nekom velikom programu koji se sastoji od puno fajlova, npr. 200 izvornih fajlova, onda build-alati puno pomažu kod izgradnje izmenjenog programa.Ovi alati se koriste kod testiranja i integracije i debugging-a softvera.

OSOBINE KOMPAJLERA

Ne treba totalno verovati kompajleru o lokaciji sintaksne greške.

Detektori sintaksnih greški: Kompajleri

Kompajleri su od velike koristi kod otkrivanja sintaksnih greški,

Medjutim ne treba totalno verovati kompajleru o lokaciji sintaksne greške, jer kompajleri znaju da pogreše i da netačno lociraju liniju programa gde je problematična sintaksna greška

Takodje ne treba totalno verovati porukama proizvedenim od strane kompajlera, ponekad su netačne, jer kompajleri nisu tako savršeni programi, već samo pomoćno sredstvo kojem ne treba totalno verovati

Ne treba verovati tzv. kaskadnim porukama kompajlera, jer, naime, posle pojave prve greške, javlja se još niz kaskadnih grešaka, i interpretacija kompajlera je često pogrešna kada su u pitanju kaskadne greške

Podela programa kod kompilacije, naime kompilacija pojedinih delova programa umesto kompilacije celog programa, je vrlo korisna za sintaksne greške

DIJAGNOSTIČKI ALATI

Komparatori izvornog koda su korisni kod modifikacija programa u cilju korigovanja greške.

Dijagnostički alat:

- **'Skele za program'** tj. Scaffolding Kao što je prethodno napomenuto, kod nalaženja defekta, nekad je neophodno , Izvući problematični deo izvornog koda, napisati dodatni kod za testiranje ovog problematičnog koda, tzv. 'skele za program', i onda egzekucija tog test-programa.
- **Okviri za testiranje** tj. Test frameworks- pomažu kod testiranja pomoću skela, naime test-frameworks alati omogućuju laku izradu skela i upotrebu tih skela za testiranje. Npr. alat JUnit,CppUnit, NUnit, itd.
- **Komparatori izvornog koda** (Source-code comparators):Komparatori izvornog koda su korisni kod modifikacija programa u cilju korigovanja greške. Naime nekada je potrebno uporediti modifikovan program i početni program, da bi se neka promena preispitala ili izbacila, jer npr kod korigovanja programa, često se prave nove greške u novoj verziji programa, i ovi komparatori omogućuju pomažu kod detekcije novih greški, jer pomažu osvežavanju memorije programera

ALATI ZA DEBUGGING-LISTA

Sledeći alati se koriste kod debugging-a.

Sledeći alati se koriste kod debugging-a:

- Test scaffolding (skele za testiranje)
- Alati za poredjenje različitih verzija programa („Diff tools")
- Trace-monitors
- Debugger (Interactive Debugger) za dijagnozu greški

Debugger je alat koji omogućuje da

- koračate kroz program korak-po-korak, tj. tzv. "slow motion" programa,
- vršite inspekciju programa tako što posmatrate npr. vrednost nekih varijablu
- koračanje kroz program pomoću debugger-a što ima sličnosti sa onim kad drugi programer korača kroz nečiji program, dakle to je neka vrsta zamene za pregled programa od strane drugog programera

DIJAGNOSTIČKI ALAT: PROGRAMI ZA DIJAGNOZU GREŠKE (DEBUGGERS)

Primenom programa za dijagnozu greške, programa debugger, i tehnike koračanja, pauztiranja, i posmatranja, mogu otkriti uzrki neke greške, i zatim ova greška ispraviti.

Debugger: Debugger, to je specijalni program koji omogućuje dijagnozu i eliminaciju greški u nekom programu.

Koristi se nekoliko načina da se identifikuje greška:

- Stepping, tj. 'koračanje' kroz program, gde debugger izvršava vaš program liniju po liniju, tako da se može locirati gde je greška
- Breakpoints, pauziranje programa u pojedinim tačkama, znači umesto da se ceo program ispituje linija po linija, ovde se definišu zaustavne tačke, pa se onda može ispitivati linija po linija, a ostatak programa se preskače kod ispitivanja
- Watching, posmatranje, koje omogfućuje da se ispita koje podatke memoriše vaš program u memoriji, tj. da li memoriše očekivane podatke

Korišćenjem prethodne tri tehnike, mogu se otkriti uzroci greški. Prema tome, ako kod izvršavanja ili testiranja programa se utvrdi da program ne radi kako treba, onda se primenom programa za dijagnozu greške, programa *debugger*, i tehnike koračanja, pauztiranja, i posmatranja, mogu otkriti uzrki neke greške, i zatim ova greška ispraviti. Medjutim, kod ispravljanja greški može se napraviti nova greška, ako se ne razume u potpunosti šta se u programu dešava.

Programi za dijagnostiku greške se mogu kupiti na tržištu. Programi za dijagnostiku omogućuju: da se postave zaustavne tačke u programu, da se program jednostavno zaustavi kod odredjene linije u programu ,ili da se zaustavi kada se nekoj varijabli zadaje vrednost, omogućuju izvršavanje programa liniju po liniju, tzv. koračanje kroz program ili podprogram, ili preskakanje podprograma , omogućuju izvršavanje programa unazad do tačke gde je defekt započeo , takodje omogućuju printovanje raznih iskaza u toku izvršavanja programa, npr 'pogledaj ovde' ili slično, pomoću njih se mogu ispitivati podaci u programu , npr. dinamički alociranu matricu podataka

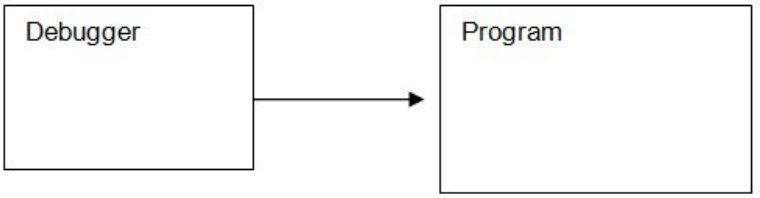
VAŽNOST DEBUGGER-A

Dijagnostički program je samo jedno pomoćno sredstvo, i on nije zamena za razmišljanje kao glavni alat dijagnostike greške.

Postoje kontroverzna mišljenja o dijagnostičkim programima. Neki programeri su protiv njihove upotrebe, argumentujući da se defekti brže i tačnije otkrivaju pomoću razmišljanja i tehnika opisanih u prethodnim glavama teksta, i da se mentalnom egzekucijom programa, a ne dijagnostičkim programom, trebaju otkrivati defekti. Medjutim ovo mišljenje da treba

odbaciti dijagnostičke programe, nije u redu. Medjutim treba reći, da svako orudje se može koristiti ispod ili iznad svojih mogućnosti, i to važi i za dijagnostičke programe.

Naime , dijagnostički program je samo jedno pomoćno sredstvo, i on nije zamena za razmišljanje kao glavni alat dijagnostike greške. Ali ni razmišljanje nije totalna zamena za dijagnostički program. Kod interaktivne dijagnostike se koristi debugger da se menjaju razni parametri u programu u toku izvršavanja programa, medjutim interaktivni debugger ima nedostatke da ohrabruje prilaz pokušaj-i-pogreši, *trial-and-error*, tj haotično otkrivanje greški, umesto sistematskog prilaza tj sistematskog dizajna.



VIDEO VEŽBA

Video vežba - Upotreba Eclipse debagera (Using the Eclipse Debugger)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO VEŽBA 2

Video vežba - Vežba iz debagovanja (Tutorial : How to Debug a java program in eclipse and other tips or shortcuts)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

ZADACI ZA VEŽBU

Zadaci

- .Demonstrirati koračanje kroz program pomoću debagera?
- Demonstrirati zastojne tačke u programu pomoću debagera?
- Demonstrirati posmatranje varijabli pomoću debagera?

Poglavlje 4

Vežba : Upotreba Java alata za debagovanje

JAVA DEBUGGER-ECLIPSE DEBUGGING

Java debugger- pokazni primer.

Na sledećem linku proučiti upotrebu Java debugger-a:

http://www.vogella.com/tutorials/EclipseDebugging/article.html

Java Debugging with Eclipse - Tutorial

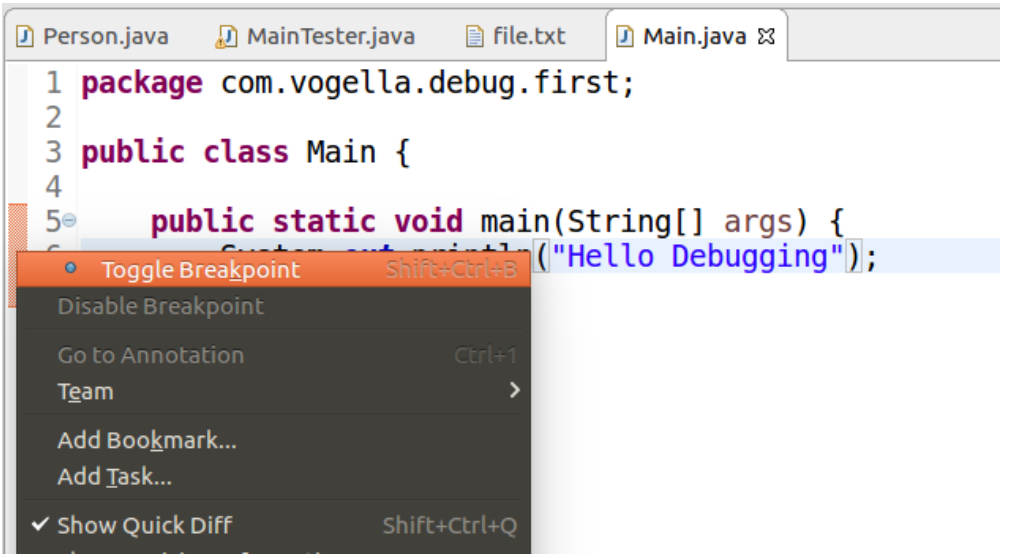
Šta je debugging?: Debugging vam omogućuje da izvršavate program interaktivno, dok posmatrate (watching) source-code i varijable u toku same egzekucije. Zastojna tačka (breakpoint) u source-code označava gde program da stane u toku debugging-a. Kada program stane na zastojnoj tački, možete da ispitujete vrednosti varijabli i menjate ih, itd. Takođe može se specificirati watchpoint, da zaustavite egzekuciju ako se neko polje učitava ili modifikuje. Breakpoint i watchpoint se nazivaju stop-points.

Debugging podrška u Eclipse-u alatu**:** Eclipse omogućuje da startujete Java program u Debug-mode. Eclipse omogućuje da kontrolišete egzekuciju pomoću debug-komandi.

SETTING BREAKPOINTS:

Slika ilustruje zadavanje zastojnih tačaka.

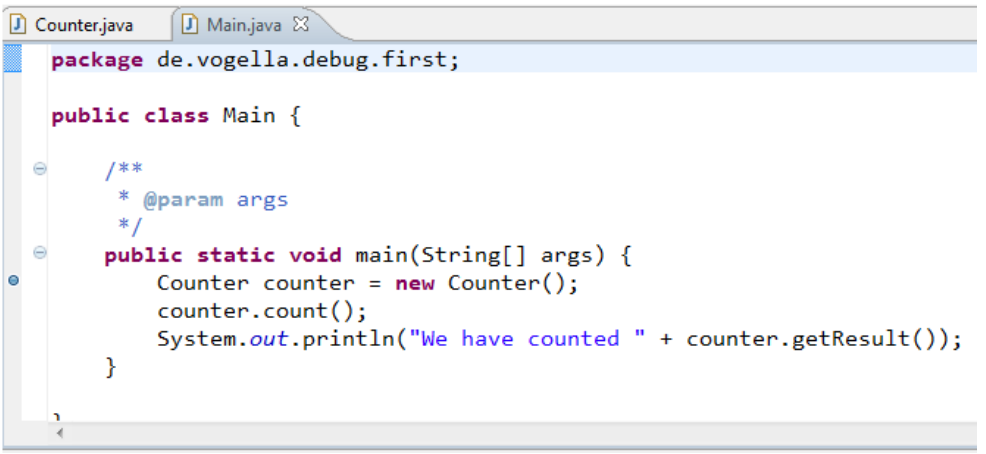
Setting Breakpoints: Da bi specificirali zastojnu tačku u programu, pšomoću right-click u levoj margini u Java-editor-u i izabrati Toggle Breakpoint. Alternativno, možete pomoću double-click na istoj poziciji.



PRIMER ZASTOJNE TAČKE

Na slici vidimo primer zastojne tačke.

Na primer, na sledećem screenshot-u je postavljena zastojna tačka kod instrukcije: Counter counter = new Counter();.

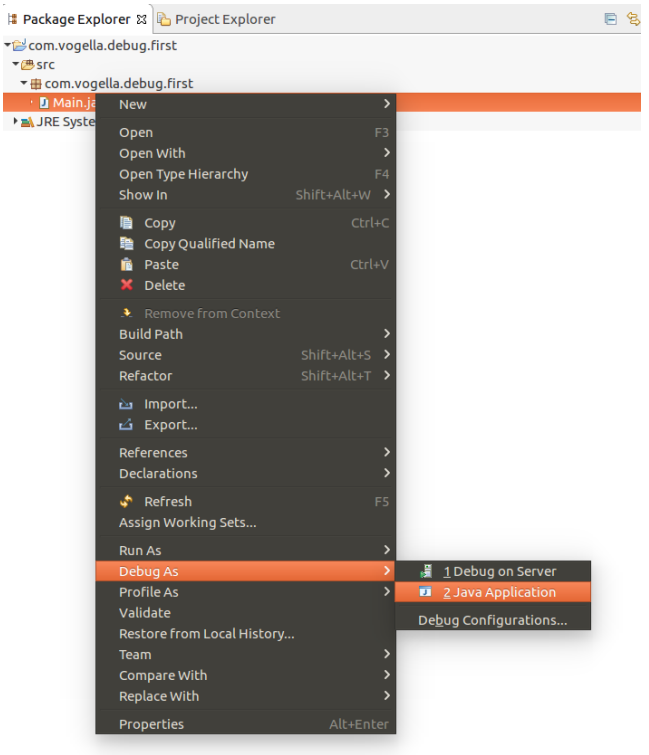


Slika 4.1 -Primer zastojne tačke

STARTOVANJE DEBUGGER-A

Da bi debug-ovali vašu aplikaciju, izabrati neki Java-fajl koji ima main()-metodu.

Startovanje Debugger-a. Da bi debug-ovali vašu aplikaciju, izabrati neki Java-fajl koji ima main()-metodu. Onda, desni-klik na toj metodi, i izabrati: Debug As ▸ Java Application.

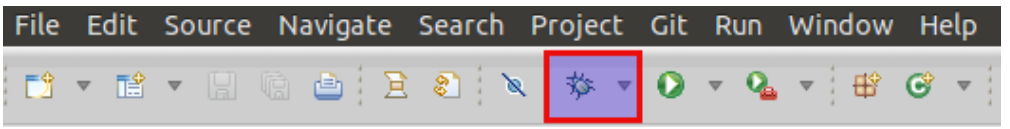


Slika 4.2 -Startovanje debugger-a

ECLIPSE TOOLBAR

Na slici je prikazan Eclipse-toolbar.

IAko ste startovali aplikaciju pomoću context-menu, možete da koristite Debug-button u Eclipse-toolbar.

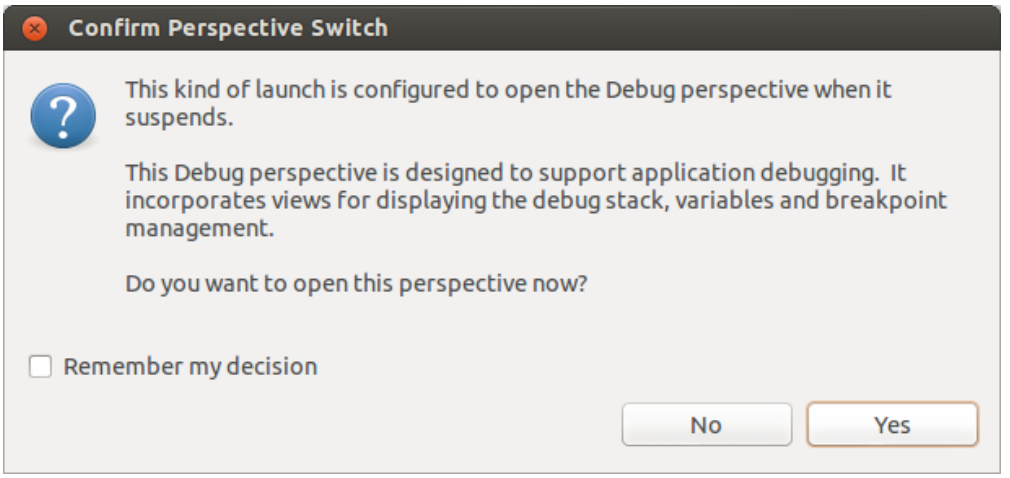


Slika 4.3 -Eklips toolbar

DEFINE BREAKPOINTS

Da biste debagovali program, potrebno je definisati "breakpoints".

Da biste debagovali program, potrebno je definisati "breakpoints". Kada dostignete zastojnu tačku, Eclipse vas pita da li želite da se prebacite u "Debug perspective", Odgovorite sa YES, i Eclipse otvara tu perspektivu, kao što se dole vidi.



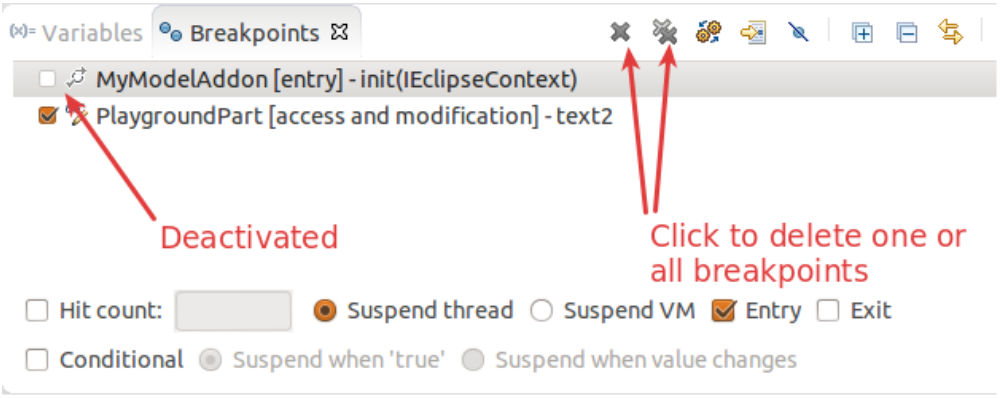
Slika 4.4 -Definisanje zastojne tačke

KONTROLISANJE EGZEKUCIJE PROGRAMA:

Pomoću Breakpoints-view možete da izbrišete ili deaktivirate delete breakpoints i watchpoints.

Controlling the program execution: Eclipse ima dugmad toolbar-u, za kontrolu egzekucije programa koji debug-ujete. Postoji dugme koje vam omogućujr koračanje kroz program.

Breakpoints view and deactivating breakpoints: Pomoću Breakpoints-view možete da izbrišete ili deaktivirate delete breakpoints i watchpoints. Dole su demonstrirane dole na slici.



Slika 4.5 -Kontrola egzekucije programa

EVALUACIJA VARIABLI U DEBUGGER-U

Pomoću Variables-view prikatzuju se polja (fields) i lokalne variable iz trenutno izvršavajućeg stack-a.

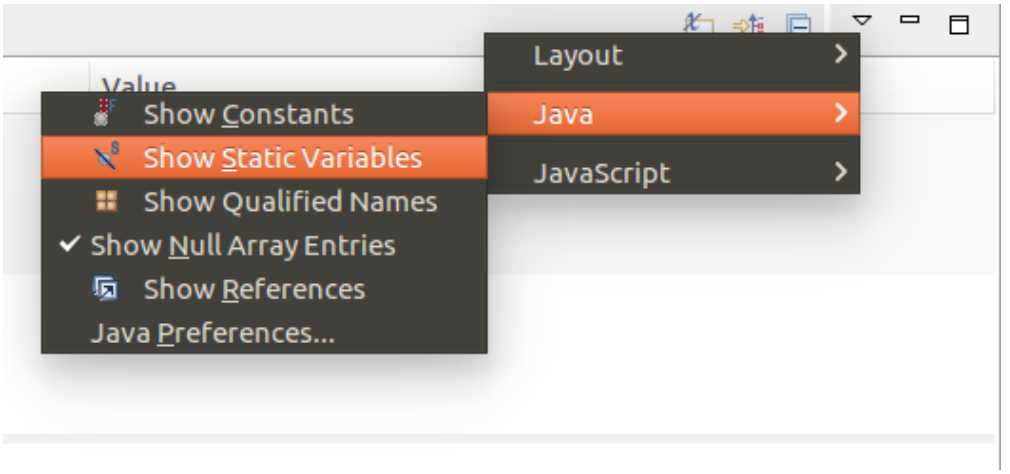
Slika 4.6 -Evaluacija varijabli

Evaluacija variabli u debugger-u: Pomoću Variables-view prikatzuju se polja (fields) i lokalne variable iz trenutno izvršavajućeg stack-a.

DROP-DOWN MENU

Pomoću drop-down menija se mogu prikazati statičke varijable.

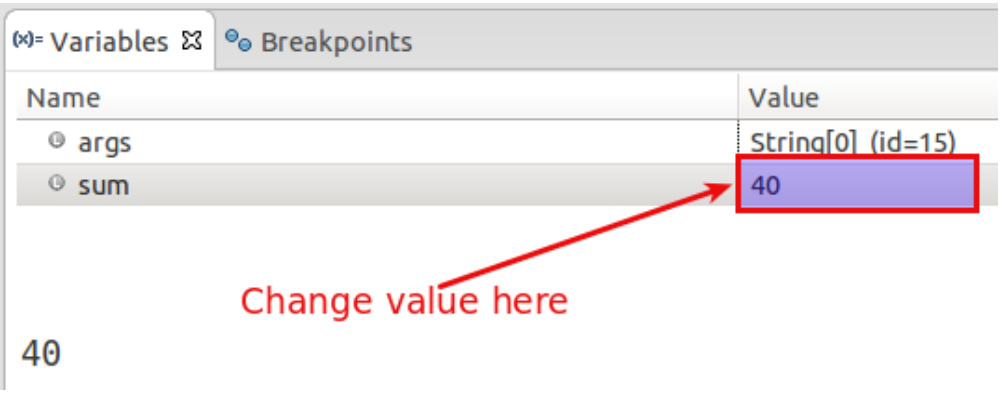
Pomoću drop-down menija se mogu prikazati statičke varijable. Na donjoj slici je prikazan drop-down meni.



Slika 4.7 -Drop-down meni

PROMENA VREDNOSTI VARIJABLE

Pomoću Variables-view može se promeniti vrednost koja je zadana nekoj varijabli, i to u toku runtime.



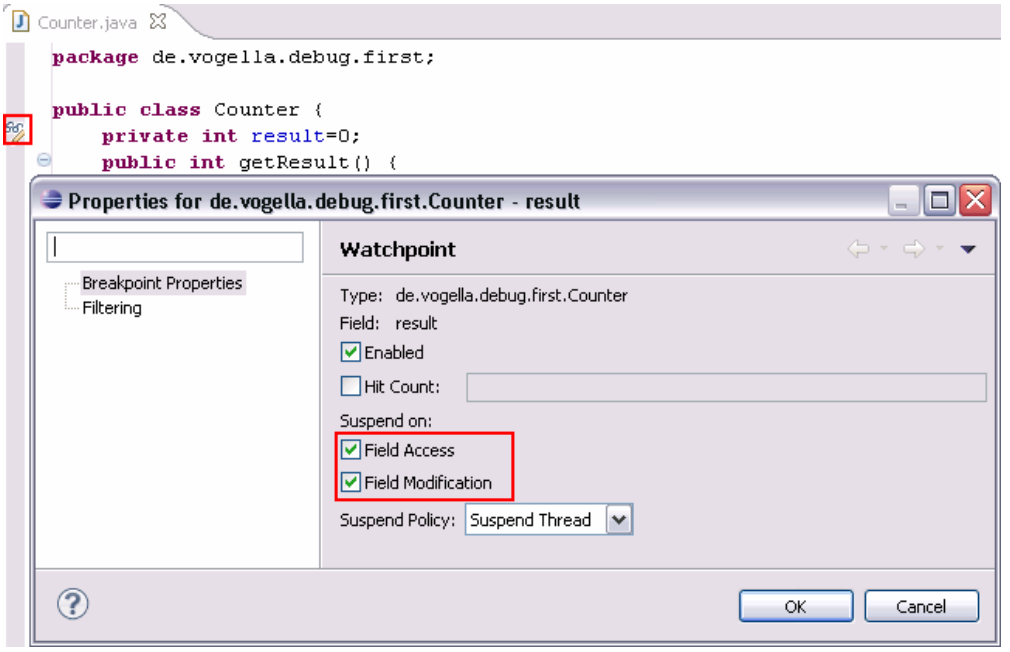
Slika 4.8 -Promena instrukcije

Promena zadate vrednosti varijable: Pomoću Variables-view može se promeniti vrednost koja je zadana nekoj varijabli, i to u toku runtime.To je ilustrovano na slici.

WATCHPOINT

Tačka posmatranja tj. watchpoint, je u stvari breakpoint zadata za polje.

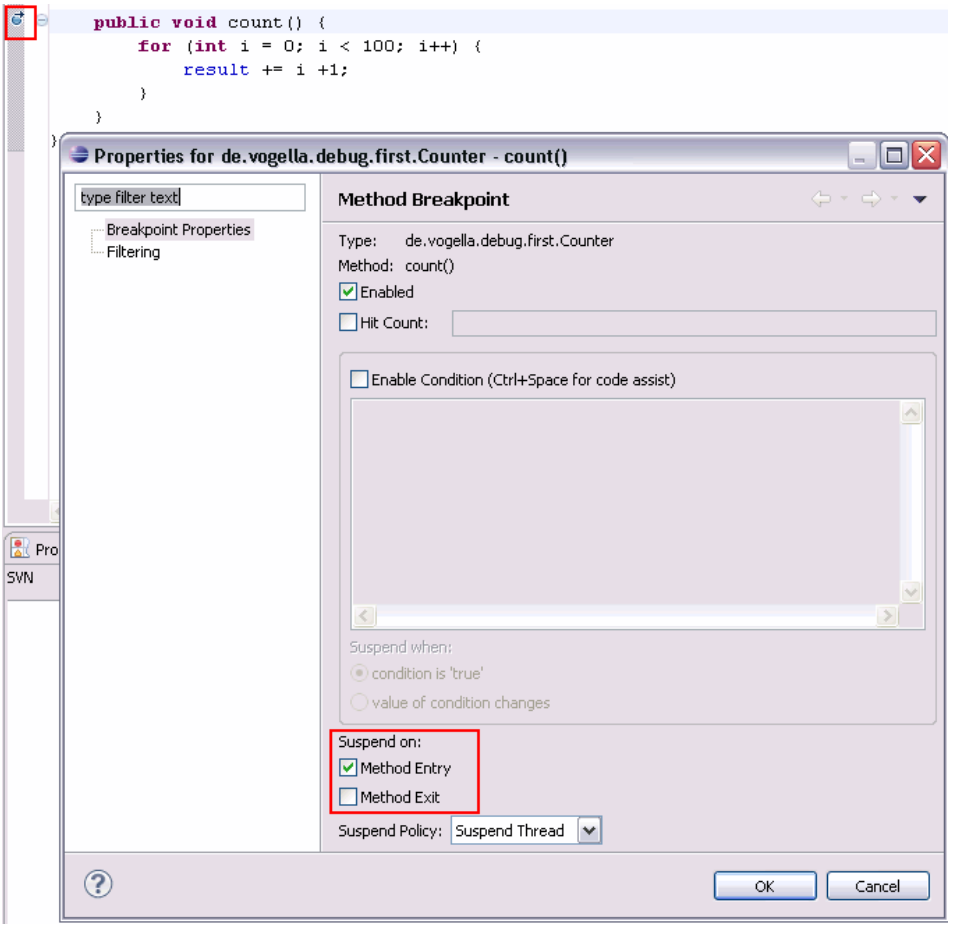
Watchpoint: Tačka posmatranja tj. watchpoint, je u stvari breakpoint zadata za polje. Debugger će da stane kadgod naiđe na to polje. Možete da zadate tačku posmatranja, watcpoint, pomoću double-clicking na levoj margini, pored deklaracije polja.



Slika 4.9 0 0 0 0 0 0 0 0 0 0 0 0-Tačka posmatranja (watchpoint)

METHOD-BREAKPOINT

Zastojna tačka metode,method-breakpoint, je zadata pomoću double-clicking na levoj margini u of editoru, odmah pored method-header.



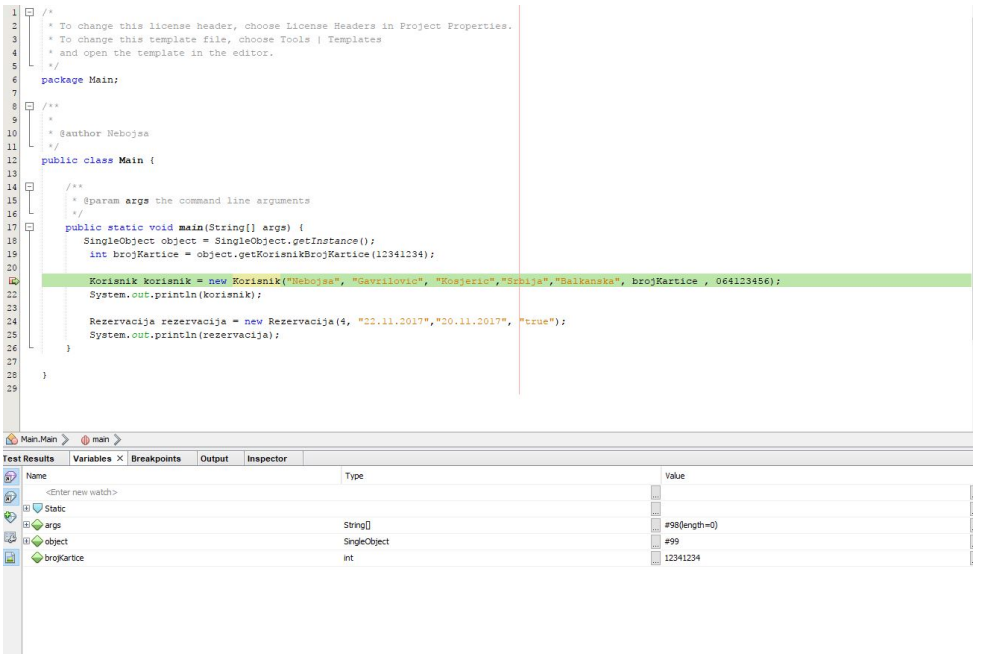
Slika 4.10 -Metodska zastojna tačka

Method breakpoint : Zastojna tačka metode,method-breakpoint, je zadata pomoću double-clicking na levoj margini u of editoru, odmah pored method-header. Na slici je ilustrovano zadavanje zastojne tačke metode.

POKAZNA VEŽBA - UPOTREBA DEBBUGING ALATA U NETBEANS RAZVOJNOM OKRUŽENJU

Upotreba debbuging alata u NetBeans razvojnom okruženju omogućava proveru programa napisanih u Java programskom jeziku na jednostavan i intuitivan način.

Na početku potrebno je postaviti breakpoint u okviru određene klase. Za primer uzeta je klasa Korisnik u okviru programa za rezervaciju karata. Breakpoint će biti postavljen u sledećem delu programskog koda:

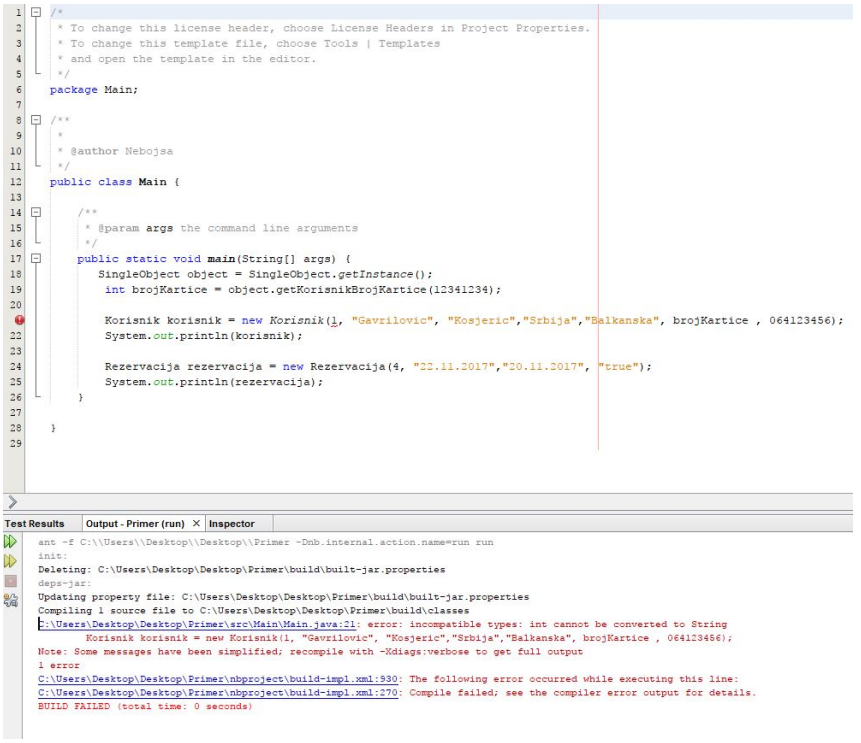


Slika 4.11 Primer postavljanja breakpointa u okviru klase Korisnik

POKAZNA VEŽBA - DOBIJENI REZULTATI UPOTREBE DEBBUGING ALATA U NETBEANS RAZVOJNOM OKRUŽENJU

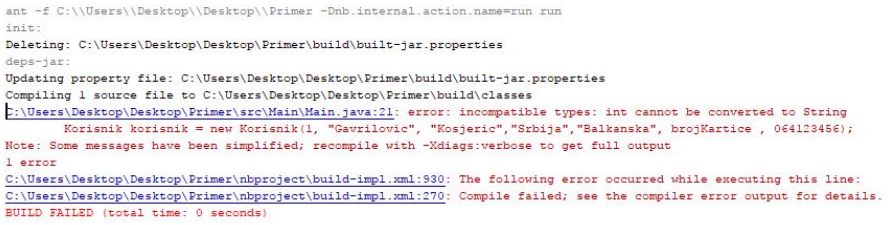
Dobijeni rezultati debbuging alata omogućavaju programeru da na brz i efikasan način izvrši izmene u programu i nastavi sa daljim radom.

Debbuger proverava kreiranje novog korisnika sa parametrima koji su mu dodeljeni (primer sa slike 12). Sve je označeno zelenom bojom jer tipovi podataka prilikom kreiranja novog korisnika odgovaraju prvobitno definisanim tipovima u okviru programa. Za potrebe testiranja umesto imena korisnika postavićemo broj 1. Programski kod sada izgleda ovako:



Slika 4.12 Primer koda gde je umesto tipa String unet broj

Umesto očekivanog tipa string unet je broj. Ponovo je pokrenut debbuger u okviru NetBeans razvojnog okruženja i nakon provere programskog koda prikazano je upozorenje da programski kod nije validan i da se podaci koji se očekuju ne poklapaju sa podacima koji su uneti kroz program. Na taj način prikazana je greška i programer mora izvršiti izmene u kodu kako bi nastavio testiranje i proveru programskog koda.



Slika 4.13 Ispisivanje greške za različit tip podataka od očekivanog

POKAZNA VEŽBA - POREĐENJE UPOTREBE DEBBUGING ALATA U RAZLIČITIM RAZVOJNIM OKRUŽENJIMA

Upotreba debbuging alata je slična u različitim razvojnim okruženjima za Java programski jezik.

U nastavku je dat primer upotrebe debbuging-a u NetBeans razvojnom okruženju:

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Primer upotrebe debbuging-a u Eclipse razvojnom okruženju:

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

DEBUGGING-PITANJA?

Evo liste pitanja iz debugging-a.

- Koje tehnike koriste debugger-i?
- Koji su nedostaci debugger alata?
- Koji problemi se mogu pojaviti kod upotrebe debugger-a?
- U čemu je razlika izmedju pojedinih kompajlera?
- Šta su to run-time greške?
- Navesti tipične komande raspoložive u debbuger alatima?
- Objasniti koračanje (stepping) kroz program ?

TEST PITANJA IZ DIJAGNOSTIKE

Navodimo test pitanja iz dijagnostike.

Kako se dijagnoza greške može koristiti kao prilika da se nauči o programu, o greškama programiranja,

Kako se dijagnoza greške može koristiti kao prilika da se nauči o prilazu rešavanja problema uopšte?

Zašto je prilaz probaj-pogreši, tj prilaz preko probe i greške, neefikasan kod dijagnoze greški u softveru?

Da li treba pretpostaviti da je greška usled previda ili nerazumevanja programera ?

Da li treba koristiti naučni prilaz istraživanja, kod stabilizacije greške?

Da li treba koristiti naučni prilaz istraživanja, kod lociranja greške?

Koje su tehnike za lociranje greške?

Da li je potrebno verifikovati i kako korekciju greške?

Koliko treba verovati kompajleru?

Šta su to skele programa i kad se koriste (scaffolding)?

Šta je to interaktivna dijagnostika, i koji su nedostaci?

Dati jedan primer dijagnostičke greške?

INDIVIDUALNA VEŽBA

Java Debugging pomoću Eclipse -individualna vežba

Zadaci:

- Na primeru jedne izabrane aplikacije demonstrirati primenu Java debugger-a kod :Setting Breakpoints ?
- Takođe demonstrirati primenu Java debugger-a kod : Evaluating variables in the debugger ?
- Takođe demonstrirati primenu Java debugger-a kod :: Breakpoints view and deactivating breakpoints ?
- Takođe demonstrirati primenu Java debugger-a kod :Setting Watchpoint ?

DOMAĆI ZADATAK - DZ9

Šesti domaći zadatak je individualan za svakog studenta.

Domaći zadatak 9 je individualan za svakog studenta, i dobija se po definisanim instrukcijama.

Domaći zadatak se imenuje:
KI301-DZ9-ImePrezime-brIndeksa gde su vrednosti Ime, Prezime i br.Indeksa vaši podaci.

Domaći zadatak je potrebno poslati na adresu asistenta: nebojsa.gavrilovic@metropolitan.ac.rs sa naslovom (subject mail-a) **KI301-DZ9.**

Posebno je potrebno voditi računa o pravilnom imenovanju mail-a prilikom slanja domaćih zadataka.

Napomena:
Domaći zadaci treba da budu realizovani u zadatku navedenom razvojnom okruženju i da predstavljaju jedinstveno rešenje svakog studenta. Prepisivanje i preuzimanje programskog koda sa interneta strogo je zabranjeno.

Poglavlje 5

Zaključak

ZAKLJUČAK

Logičke geške su najsakrivenije, i najteže za otkrivanje tj dijagnozu. Program funkcioniše ali daje nepredvidljive tj. neočekivane rezultate a mi neshvatamo zašto. Da bi se našla logička greška, mora se program proveravati linija po linija da bi se utvrdilo šta nedostaje, ili šta je pogrešno u logici postojećih instrukcija. Pošto logičke greške se teško pronalaze, kompajleri , language compilers, omogućuju specijalne debugging, dijagnostičke, mogućnosti, i dva glavna načina da se pronadju logičke greške su:
stepping, koračanje kroz program, i watching, razgledanje prrograma.
Stepping predstavlja izvršavanje programa linija po linija, i posmatranjem kako program radi. U momentu kada se otkrije da program radi nešto pogrešno, vi onda znate tačno koja linija u programu pravi grešku. Npr. za koračanje kroz neki program u Liberty BASIC, sledeće instrukcije se koriste:
Debug, Step Into, Step Out, Itd. Umesto koračanja linija po linija kroz ceo program, od početka programa, što može biti nepotrebno, posebno ako se ima ideja šta može bitim pogrešno, onda se može program koračati kroz program ne od početka već neke izabrane tačke, npr. sredina ili pri kraju programa. Ovo se zove tracing trough your program.Kada se krača kroz program, onda se može posmatrati sadržaj pojedinih varijabli, i posmatrati kako se ove vrednosti varijabli menjaju, i koje linije izazivaju promene kojih varijabli. Ovo je posmatranje varijabli, tj.watching your variables.
Dijagnostički alat:
‘Skele za program’ tj. Scaffolding Kao što je prethodno napomenuto, kod nalaženja defekta, nekad je neophodno , Izvući problematični deo izvornog koda, napisati dodatni kod za testiranje ovog problematičnog koda, tzv. ‘skele za program’, i onda egzekucija tog test-programa.
Okviri za testiranje tj. Test frameworks- pomažu kod testiranja pomoću skela, naime test-frameworks alati omogućuju laku izradu skela i upotrebu tih skela za testiranje. Npr. alat JUnit, CppUnit, NUnit, itd.
Komparatori izvornog koda (Source-code comparators):Komparatori izvornog koda su korisni kod modifikacija programa u cilju korigovanja greške. Naime nekada je potrebno uporediti modifikovan program i početni program, da bi se neka promena preispitala ili izbacila, jer npr kod korigovanja programa, često se prave nove greške u novoj verziji programa, i ovi komparatori omogućuju pomažu kod detekcije novih greški, jer pomažu osvežavanju memorije programera

LITERATURA

1. Code Complete: A practical handbook of software construction, by S. McConnell, Microsoft Press, ISBN 0-7356-1967-0, https://www.amazon.com/Code-Complete-Practical-Handbook-

Construction/dp/0735619670
2. SWEBOK-V3, https://www.computer.org/web/swebok/v3
3. Theory and Problems of Software Engineering - Schaum's Outline Series, by David Gustafson, ISBN 0-07-137794-8, http://www.mhebooklibrary.com/doi/abs/10.1036/0071377948

http://www.vogella.com/tutorials/EclipseDebugging/article.html, Eclipse Debugging

http://www.cs.usyd.edu.au/~jchan3/soft1001/jme/debugging/debugging_task.html, Debugging Tutorial