Lekcija 11 – Konstrukcija, Detekcija/Lokali-zacija Grešaka

Primer loše implementacije

```
void HandleStuff(CORP_DATA inputRec, int crntQtr, EMP_DATA empRec,
   Double estimRevenue, double ytdRevenue, int screenx, int screeny, Color newColor,
   Color prevColor, StatusType status, int expenseType) {
int i;
for ( i = 0; i < 100; i++ )
        {
        inputRec.revenue[i] = 0;
        inputRec.expense[i] = corpExpense[crntQtr][i];
        }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
        for ( i = 0; i < 12; i++ )
                profit[i] = revenue[i] - expense.type1[i];
        }
else if ( expenseType == 2 ) {
                profit[i] = revenue[i] - expense.type2[i];
        }
else if ( expenseType == 3 )
                profit[i] = revenue[i] - expense.type3[i];
                }
```


O značaju lepote i sređenosti okruženja (pa stoga i izvornog kôda) vredi pročitati članak o razbijenim prozorima (Broken Windows): https://en.wikipedia.org/wiki/Broken_windows_theory. Ova ideja se primenila i u softverskom inženjerstvu, i ogledava se u tome da lep program sam po sebi motiviše sklad, pravilnu strukturu i čitljivost zapisa istog.

Laboratorijski zadatak: pokušajte identifikovati koji su sve principi OOAD prekršeni u primeru HandleStuff. Šta je sa dokumentacijom? Odakle potiču "magični" brojevi (konstante u programu)? Da li je primenjen princip defanzivnog programiranja? Da li ima nekorišćenih promenljivih? Da li ima bočnih efekata (*side effects*)? Da li se uopšte ovde obaveštava klijent o statusu izvršenja metode?

Da li kod ovog primera možemo prepoznati da je ono pisano u Java jeziku? Šta govori Javin stil zapisa programa? Zašto su tz. *Style Guide*-ovi korisni (pogotovo u velikim organizacijama)?

Defanzivno programiranje

- Program treba sam sebe da zaštiti od pogrešnih ulaznih podataka (svake vrste).
- Primer: videti listing iz knjige `HousePriceExampleCh12`
- Iskaz `assert` treba koristiti za proveru pretpostavki (uslove za koje se pretpostavlja da će uvek važiti u normalnim situacijama) i invarijanti.
- Nikad se ne smeju greške ignorisati bez ikakvog signala ka klijentu (klijent može biti i neki drugi modul, koji koristi naš program).
- Primer: videti listinge `fileTest.c` i `errTest.c` iz knjige.
- Primer: videti listinge `FileTest.java`, `FileTestWrong.java`, `TestExceptions.java` i `TestNullCatch.java` iz knjige za izuzetke (*exceptions*) i rukovanje greškama.

Lokalizacija i ispravljanje grešaka

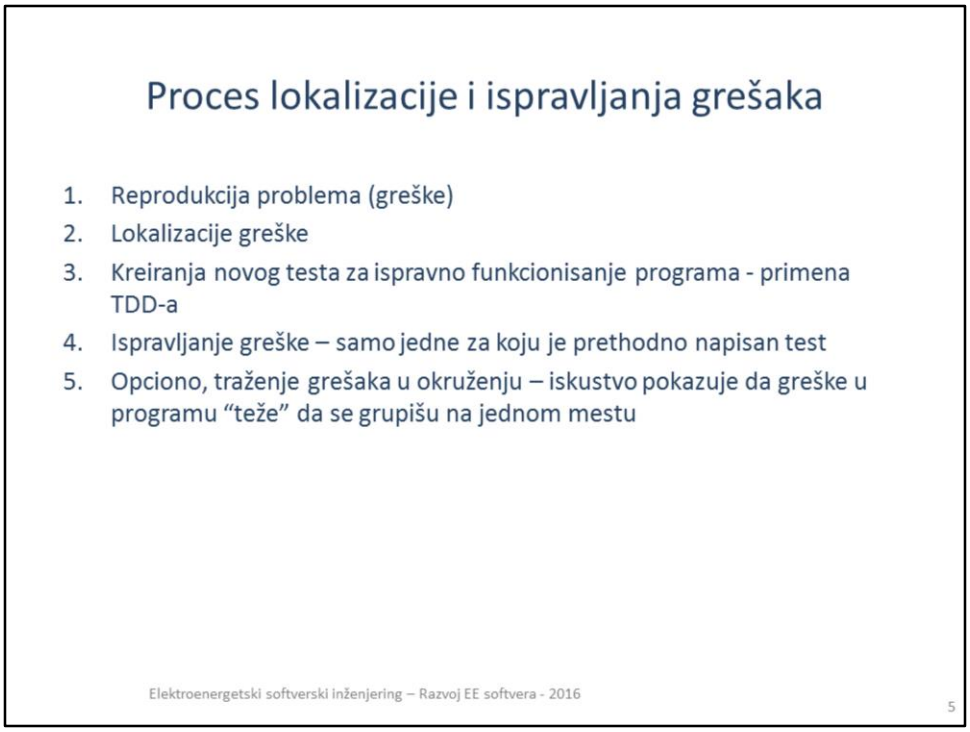
- Zajednički naziv za ove aktivnosti je *debugging*. To je proces nalaženja uzroka greške i ispravljanje iste.
- Greške se mogu podeliti u sledeće 3 kategorije:
  - sintaksne
  - semantičke

```
// Gde je ovde greška?
while (j < MAX_LEN); {
    // do stuff here
    j++;
}
```

  - logičke
- Nikad ne radite sledeće: nagađati gde je greška, zataškati simptom umesto da rešite problem i skidati odgovornost sa sebe (nalaziti izgovore).

Postoje programi, koji statički analiziraju izvorni kôd i prijavljuju potencijalne sementičke greške. Na primer, FindBugs (http://findbugs.sourceforge.net) za Java programe. Sintaksne greške prijavljuje sam prevodilac.

ZAPAMTITE: nema ničeg goreg od lošeg izgovora!



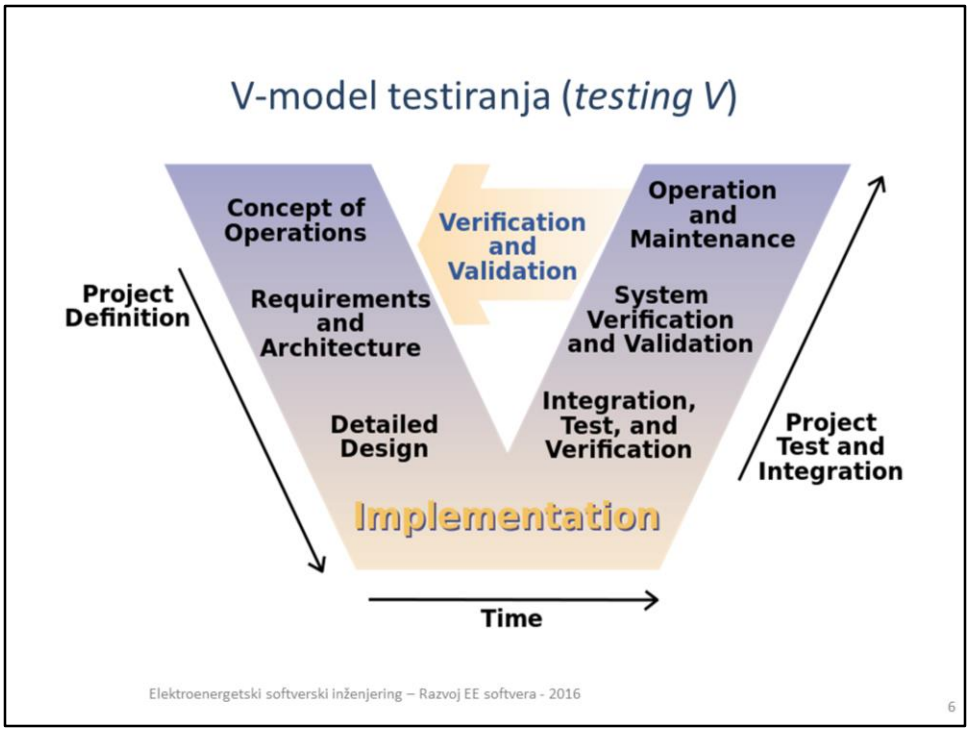
Suštinska razlika između razvoja novih funkcija i lokalizacije grešaka, jeste, strela vremena. Naime, ovde se polazi od činjeničnog stanja (simptom(i) greške) i kreće vremenski unazad do momenta pojave uzroka greške.

Kod reprodukcije problema, u veoma retkom broju slučajeva je to nemoguće uraditi, na osnovu raspoloživih informacija. Međutim, nemogućnost reprodukcije je onda sama po sebi svojevrsna greška u sistemu, koja se mora ispraviti (nedostatak log-ova šta se dešava u produkciji, nekontrolisane izmene produkcione verzije sistema, itd.). Takođe, ovo može biti uzrokovano pojavom koja se zove Heisenbug (https://en.wikipedia.org/wiki/Heisenbug). Uzroci mogu biti razni: greška u inicijalizaciji objekta, neadekvatna kontrola pristupa deljenim resursima u programima sa više niti, itd.

Naravno, kad govorimo reprodukciji problema, onda mislimo na nalaženje minimalnog skupa okolnosti (najčešće količine i vrste podataka), koji dovode do greške. Nema prevelike koristi ako grešku možemo reprodukovati pod uslovom da repliciramo ceo produkcioni sistem sa svim mogućim podacima.

NAPOMENA:

U knjizi postoji ozbiljan propust u vezi saveta kako napisati test za grešku. Test se uvek piše sa ciljem provere ispravnosti programa. Prema tome, u početku test za grešku mora da se zacrveni, sve dotle dok se greška ne ispravi. U knjizi je ovo opisano potpuno suprotno, što eliminiše mogućnost upotrebe pomenutog testa u bateriji regresionih testova sistema.



Sliku V-modela i njegov opis možete videti ovde: https://en.wikipedia.org/wiki/V-Model_(software_development)

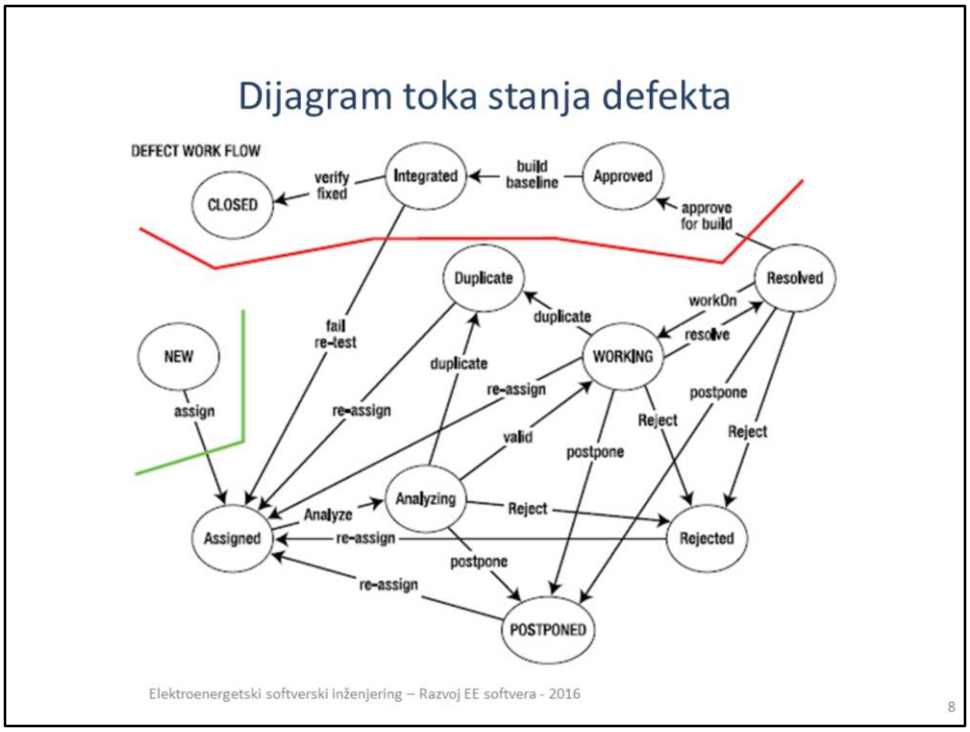
Kao što vidimo, V-model je primenljiv za sve kategorije testova: *unit (white-box)*, *integration* i *sistem (black-box)*.

Primer: videti listinge iz knjige u folderu JUnitTestExample za poglavlje 14.

Treba uvek pamtiti sledeći citat (dat u originalu) Edsger-a Dijkstra-e: "…program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

Kvalitet (*quality assurance and control*)

- Testiranje samo po sebi može otkriti do 50% grešaka. Zato se koriste još sledeće 3 statičke tehnike (pojmovi su dati u originalu):
  - *walkthroughs*
  - *code reviews*
  - *code inspections* (varijanta Faganovog procesa) sa sledećim kriterijumima:
    - lista čestih tipova grešaka na koje se treba fokusirati
    - fokus je naći greške, a ne kako ih ispraviti
    - svi učesnici se moraju pre sesije pripremiti
    - svako ima posebnu ulogu
    - svi učesnici su prošli formalnu obuku za ovu aktivnost
    - moderator nije autor (kao kod *code review*-a) i morao je proći još neku dodatni obuku
    - autor treba sa moderatorom da zabeleži sve greške i napravi plan ispravke
    - metrike se uvek sakupe (vreme potrebno za pripremu sesije, broj pronađenih grešaka, njihova težina, broj proverenih linija kôda, itd.)



Najpopularniji open-source sistem za praćenje defekata je Bugzilla (https://www.bugzilla.org).