SPLOŠEN KONCEPT NAČRTOVANJA TESTNIH PRIMEROV

Tomaž Dogša
cV&Vs Center za verifikacijo in validacijo sistemov
Fakulteta za elektrotehniko, računalništvo in informatiko
Univerza v Mariboru, Smetanova 17, 2000 Maribor
E-pošta: tdogsa@uni-mb.si

Izvleček

Učinkovito testiranje zmanjša skupne stroške načrtovanja in vzdrževanja ter omogoča objektiven pogled na kakovost programske opreme. V prispevku bomo prikazali enega izmed možnih splošnih konceptov načrtovanja testnih primerov, ki temelji na testirnem modelu. Za zgled smo izbrali dve preprosti in najpogosteje uporabljeni strategiji.

Abstract

An effective testing reduces the costs of development and maintenance. At the same time it enables an objective view on software quality. In the paper we shall describe the general concepts of test case design based on testing models. We have illustrated this with two simple and most widely used strategies.

■■■

1. Uvod

Testiranje programske opreme je še vedno ena izmed najdražjih aktivnosti, ki jih izvajamo v življenjskem ciklusu programskega produkta. Učinkovito testiranje zmanjša skupne stroške in omogoča objektiven pogled na kakovost programske opreme. S pojmom testiranje označujemo skupek naslednjih aktivnosti: načrtovanje testnih primerov, izvajanje programa (ali pa samo njegove komponente), opazovanje in beleženje rezultatov tega poganjanja ter njihovo vrednotenje glede na določen vidik [IEEE, 1990b]. V skupini teh aktivnosti je načrtovanje testnih primerov zagotovo ena izmed najzahtevnejših nalog.

Za sistematičen opis postopka načrtovanja testnih primerov obstaja več sinonimov. Myers govori o metodologiji načrtovanja testnih primerov ([MYERS,1979], str. 36), Beizer o testirni strategiji [BEIZER, 1990], Binder uporablja testirne šablone ([BINDER, 2000], str. 338), mnogi drugi pa se zavzemajo za pojma: testirna metoda in testirna tehnika. V tem prispevku smo se odločili za testirno strategijo.

Testni primer (test case) je sestavljen iz opisa vhodnih podatkov in pričakovanega obnašanja. Kakovosten testni primer mora biti učinkovit, razumljiv in zagotoviti mora ponovljivost. Učinkovitost je povezana z verjetnostjo odkritja nepravilnosti. Razumljiv je takrat, ko ga lahko izvede oseba, ki ni avtor tega testnega primera. Če so testni primeri zadostno razumljivi, je izvedba testa v večini primerov najenostavnejša aktivnost v celotnem testirnem procesu (več o opisovanju testnih primerov glej v [DOGŠA,1999,a]).

Načrtovanje testnih primerov poteka po podobnih korakih (fazah) kot načrtovanje programske opreme. Mnogi testni primeri so v bistvu programi (testni skripti), s katerimi delno ali pa popolnoma avtomatiziramo nekatere testirne aktivnosti [DUSTIN,1999]. Ker se program med razvojem spreminja, je potrebno ustrezno prilagajati tudi testne primere. Podobno kot pri programski opremi lahko tudi tukaj govorimo o načrtovanju, implementaciji, preverjanju, uporabi in vzdrževanju testnih primerov.

S testiranjem nikakor ni mogoče dokazati odsotnosti napak ali pravilnost delovanja, saj bi zato potrebovali izredno veliko število testnih primerov. Izjema so skrajno preprosti programi, katere lahko preskusimo z vsemi možnimi kombinacijami vhodnih podatkov. Kratke programe lahko tudi formalno verificiramo (matematično dokazovanje pravilnosti) in tako dokažemo, da ne vsebujejo napak. S testiranjem lahko dokažemo samo: 1. prisotnost določene lastnosti in 2. prisotnost ene ali več napak.

Vsak testni primer je načrtovan z nekim namenom. Glede na vrsto namena jih lahko razdelimo v dve veliki skupni: skupina pozitivnih (clean test case, pozitive test case) in skupina negativnih testnih primerov (negative test case, dirty test case). Namen pozitivnega testnega primera je dokazovanje prisotnosti določene lastnosti. Testni primer, s katerim npr. ugotavljamo, ali je sploh implementirana funkcija za izračun obresti, spada v skupino *pozitivnih testnih primerov*. Pozitivne testne primere je relativno enostavno načrtovati. Žal nas ne morejo prepričati o pravilnosti delovanja programa. Ker s testiranjem ni možno dokazati odsotnosti napak, je osnovni cilj testiranja dokazovanje njihove prisotnosti. Takim testnim primerom

pravimo *negativni testni primeri*. Za učinkovito testiranje naj bo število negativnih testnih primerov večje od pozitivnih (Beitzer priporoča razmerje 5:1 [BEITZER,1995]).

V prispevku bomo prikazali splošen koncept načrtovanja testnih primerov, ki temelji na testirnem modelu. Pobudnika za ta novejši pristop sta predvsem Binder [BINDER,2000] in Beizer [BEIZER,1990]. V drugem poglavju bomo opisali pomen predpostavke o napaki ali nepravilnosti, ki je začetno izhodišče za načrtovanje testnih primerov. Nato bomo opisali koncept testirnega modela in definirali osnovne zahteve za testirno strategijo.

2. Predpostavka o napaki ali nepravilnosti

Ideja testiranja je zelo preprosta: preverjevalec vedno predpostavlja, da program vsebuje eno ali več napak oziroma, da ima določene nepravilnosti. Ta predpostavka (bug assumption, fault assumption) je zelo pomembno začetno izhodišče za načrtovanje testnih primerov. Obstoj te predpostavke je seveda samoposebi umeven, saj če bi predpostavljali, da v programu ni nobenih napak, bi bilo testiranje nepotrebno[1]. Za ilustracijo je na sliki 1 prikazanih nekaj predpostavk.

1. Program bo pomotoma sprejel negativno vrednost.
2. V predikatu, ki se nahaja v odločitvenem stavku, je napaka.
3. Izpis tabele so pozabili implementirati.
4. Program ne bo deloval pravilno po letu 2000.
5. Če bo indeks M=0, se while zanka ne bo nikoli končala.
6. Pri določeni kombinaciji vhodnih podatkov bo program odpovedal.

Slika 1: Niz predpostavk o napakah ali nepravilnostih

Predpostavka o napaki ali nepravilnosti je v bistvu namen testnega primera. Pove, kaj želimo s testnim primerom dokazati. Zelo pogosto pretvorimo predpostavko v vprašanje, npr.: Ali je izpis tabele implementiran? S testnim primerom, ki izhaja iz te predpostavke, lahko enostavno dokažemo, ali tabela je ali ni implementirana. Ne glede na izid tega testa, dobimo vedno nedvoumen odgovor. Ker je pravilna postavitev predpostavke in namena testnega primera zelo pomemben korak pri načrtovanju, bomo navedli še nekaj napačnih oziroma neprimernih predpostavk. Npr.: *1. Namen testnega primera je preveriti pravilnost izračuna inflacije. 2. Namen testnega primera je ugotavljanje nepravilnosti.* Kaj lahko sklepamo, če program testiramo in v prvem primeru ne odpove? Je brez napak? Kot smo že v uvodu opozorili, s testiranjem ni možno dokazati pravilnost, ampak samo prisotnost napak ali določenih lastnosti. Namen pri drugem zgledu je preveč splošen, saj velja za vse testne primere. Če je namen testnega primera preveč splošen, ga bomo s težavo vzdrževali. Če uporabimo analogijo z razvojnim modelom programske opreme, ima namen testnega primera enako vlogo kot jo imajo zahteve (requirements) za program.

Po izboru namena testnega primera sledi njegova implementacija. Najprej si napravimo kratek osnutek, katerega nato postopoma izdelamo do zadostnih podrobnosti, ki zagotavljajo ponovljivost in razumljivost. Najbolj pogosta metoda, ki se v tem koraku uporablja, je postopno izboljševanje.

Zgled[2]:

Namen testnega primera: Ali je izpis tabele implementiran?

Osnutek: Izpišemo tabelo.
Izboljšan 1. osnutek: Odpremo datoteko s podatki, zahtevamo izračun stroškov. Nastalo tabelo izpišemo.
Izboljšan 2. osnutek: . . . .
:
:

Opis testnega primera: Odpremo datoteko test1.dat. V meniju Analiza zahtevamo izračun stroškov (opcija 7). Nastalo tabelo izpišemo (meni Tisk, opcija 2.

Pričakovani rezultat: Izpisana mora biti celotna tabela.

3. Testirni model

Analiza zahtev in postavitev specifikacij sta ena izmed najzahtevnejših faz v razvojnem ciklu programske opreme. Če so specifikacije pravilno zastavljene, postane implementacija v mnogih primerih rutinska zadeva. Analogno velja za načrtovanje testnih primerov. Če je namen pravilno izbran in opisan, postane realizacija testnega primera rutinska zadeva, ki jo lahko izvede preverjevalčev pomočnik. Takoj vidimo, da je izbor namena testnega primera ključni problem, ki

1 *Nekateri enačijo pojma analiza ter meritve in testiranje. Za tiste ta trditev seveda ne velja. Če želimo samo ovrednotiti lastnosti nekega produkta, je to v bistvu samo analiza, ne pa testiranje. Če vrednotenju dodamo še primerjavo z referenčnim podatkom, potem gre za testiranje.*

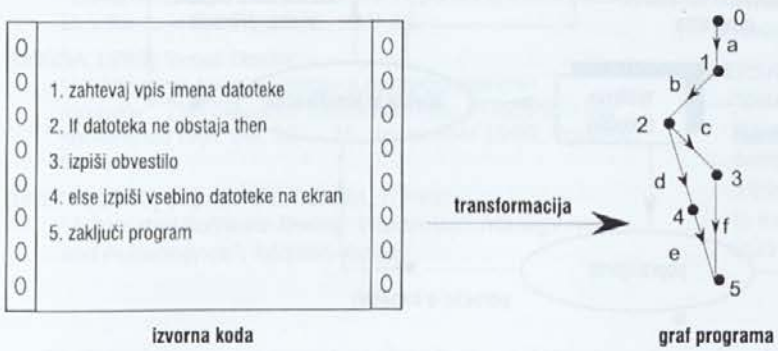
2 *Pri opisu testnega primera v prejšnjem zgledu smo zaradi preglednost prikazali le najnujnejše atribute.*

nastopa pri načrtovanju testnih primerov. Za rešitev tega problema si preverjevalci pomagajo z različnimi testirnimi modeli. Testirni model je pripomoček, ki preverjevalcu pomaga določiti namen testnega primera. V večini primerov je objekt, ki ga testiramo, zelo kompleksen. Testirni model je **poenostavljen** opis (obnašanja ali lastnosti) objekta (npr. programa). Izpuščene so vse podrobnosti, katerih preverjevalec ne potrebuje. Pogosto so testirni modeli podobni modelom, ki jih uporabljajo načrtovalci (graf prehajanja stanj, odločitvene tabele, diagram toka podatkov itd.). Večino modelov lahko predstavimo z grafom, z matriko ali s tabelo, pa tudi s seznamom.

Prvi testirni model, ki so ga uporabili za tvorjenje testnih vzorcev, je bil krmilni diagram (flowgraph, control flowgraph) oziroma graf programa (glej sliko 2). Ta model lahko zelo hitro tvorimo iz diagrama poteka (flowchart). Načrtovanju testnih primerov na podlagi krmilnega modela pravimo tudi testiranje poti (path testing). Ker je teorija testiranja programskih poti zelo dobro obdelana, se v literaturi z njo zelo pogosto srečamo [BEIZER,1990], [DOGŠA,1994], [MYERS,1979], [VLIET,1993], [JORGENSEN,1995].

Mnoge ideje, nastale v zvezi s tem modelom, je možno posplošiti in ustvariti splošen koncept testirnega modela. Najbolj preprost testirni model je seznam predpostavk o napakah ali nepravilnostih (glej sliko 1). Preproste testirne modele prikazujemo z usmerjenim grafom, kompleksnejše pa z matriko ali s tabelo. Vsak usmerjeni graf je sestavljen iz vozlišč in usmerjenih povezav. Razvoj testirnega modela poteka po naslednjih korakih:

1. določimo, kaj bodo predstavljala vozlišča
2. določimo, kaj bomo predstavili z relacijo med vozlišči
3. ustrezno povežemo vozlišča med seboj in
4. če je potrebno, opremimo vozlišča in povezave z dodatnim atributom, povezanim z določeno lastnostjo (npr. pogostost)



Slika 2: Graf programa je zelo pogost testirni model

4. Testirna strategija

Za izborom testirnega modela sledi postavitev ustrezne testirne strategije. Njen osrednji del je seveda testirni model. Testirna strategija ima podobno vlogo kot sistemske specifikacije. Če je pravilno opisana, lahko tretja oseba (npr. preverjevalčev pomočnik) relativno enostavno tvori testne primere. Vsaka pravilno zastavljena testirna strategija mora odgovoriti na naslednja vprašanja (v oklepajih so predlogi za imena posameznih alinej):

1. Kdaj in kje je ta strategija uporabna? (Območje uporabnosti)
2. Katere vrste napak ali nepravilnosti odkriva in katerih ne? (Opis predpostavke o napakah)
3. Na kakšnem testirnem modelu temelji? (Opis testirnega modela)
4. Kako na podlagi testirnega modela tvorimo testne primere? (Navodilo za načrtovanje)
5. Kateri pogoji morajo biti izpolnjeni, da bomo lahko začeli z načrtovanjem? (Pogoji za načrtovanje)
6. Kdaj je strategija izčrpana? (Terminalna kriterijska funkcija za določeno strategijo)

Omejili smo se samo na šest najpomembnejših elementov, ki tvorijo opis strategije. Nekateri dodajajo še alineje o avtomatizaciji, zgledu, morebitnih problemih in omejitvah (glej npr. [BEITZER,1995], [BINDER,2000]). Za zgled smo izbrali preprosto strategijo, ki uporablja pozitivne testne primere. Poimenovali jo bomo *Preverjanje prisotnosti zahtev*. Ker je ena izmed najpreprostejših in najpogosteje uporabljenih, navajamo njen opis:

1. Strategija je uporabna v vseh primerih, kjer so znane specifikacije in zahteve, med katerimi ni nobenih relacij. Uporablja se lahko za testiranje kompletnih programov ali pa samo komponent.
2. Predpostavka o napaki: določena zahteva ni implementirana. S to strategijo odkrivamo zahteve, ki niso implementirane. Razen zelo redkih izjem ne bomo odkrili napačno implementiranih zahtev in zahtev, ki so po nepotrebnem implementirane.
3. Testirni model je seznam zahtev.
4. Za vsako zahtevo tvorimo en testni primer. Vhodne podatke si poljubno izberemo.
5. Z načrtovanjem testnih primerov lahko začnemo, ko so zahteve postavljene.
6. Testirna strategija je izčrpana, ko preverimo prisotnost vsake zahteve v seznamu.

S to testirno strategijo bomo samo ugotovili, ali so vse lastnosti in zahteve implementirane. Le v redkih primerih bomo odkrili tudi druge nepravilnosti. Za ugotavljanje prisotnosti napak moramo

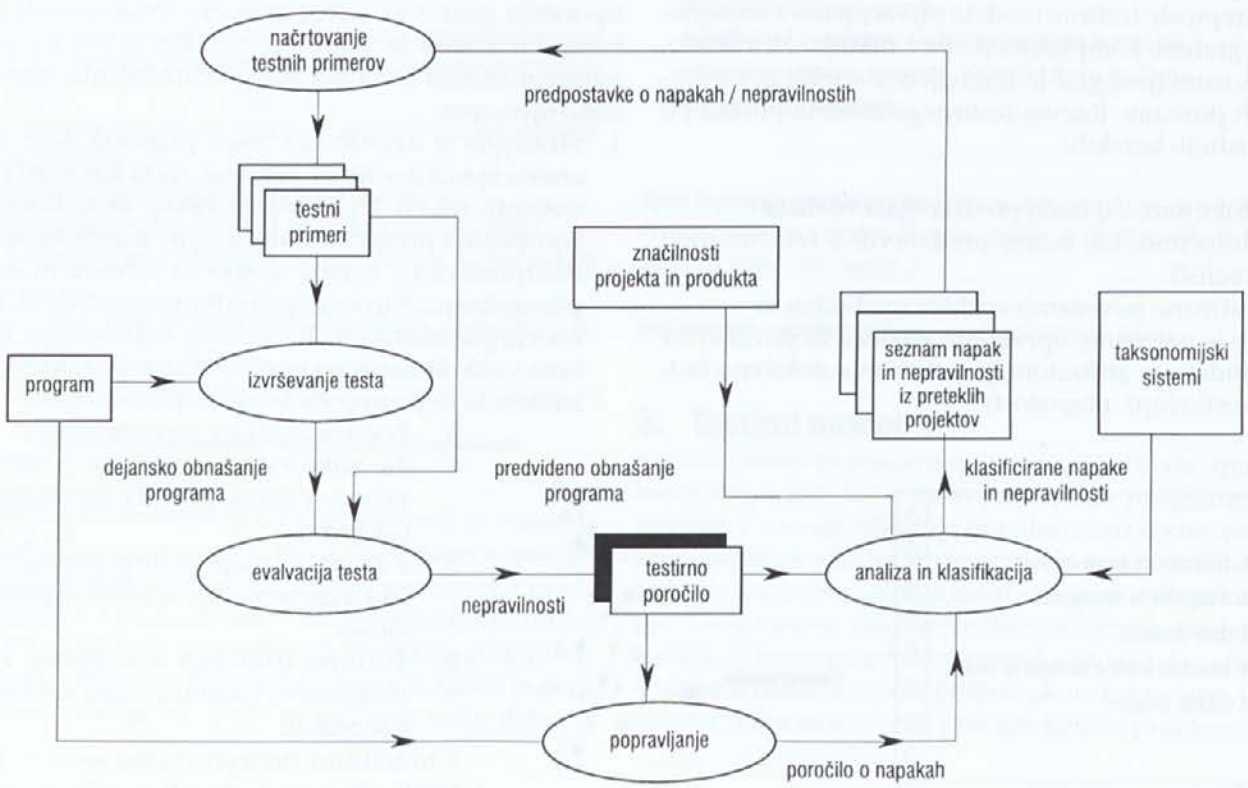
postaviti še najmanj eno ali več strategij, ki temeljijo na negativnih testnih primerih.

Druga najpogosteje uporabljena strategija uporablja seznam nepravilnosti ali napak iz preteklih projektov (glej primer na sliki 1). Za vsako nepravilnost, za katero predpostavljamo, da je prisotna, tvorimo enega ali pa več testnih primerov. Če je ta seznam tudi formalno zapisan, lahko govorimo o *sistematični strategiji*, sicer pa o *intuitivni*. Zagotovo sta intuitivno ugibanje napak in nepravilnosti ter preverjanje prisotnosti zahtev najbolj uporabljeni strategiji, saj ju uporablja večina neprofesionalnih preverjevalcev. Slabost intuitivnega ugibanja napak in nepravilnosti je predvsem v tem, da je ta strategija tesno vezana na izkušnje preverjevalca. Z njegovim odhodom na drugo delovno mesto lahko zelo pade učinkovitost preverjanja. Rešitev je v sistematičnem zbiranju najdenih napak in nepravilnosti (glej sliko 3). Preverjevalec analizira vsako poročilo o najdenih nepravilnostih in poročilo o najdenih napakah. Glede na izbran taksonomijski sistem tvori seznam, ki ga kasneje uporablja pri načrtovanju testnih primerov (več o tem glej v [BEIZER,1990], [ZAVERSKI,1999], [KANER,1993]).

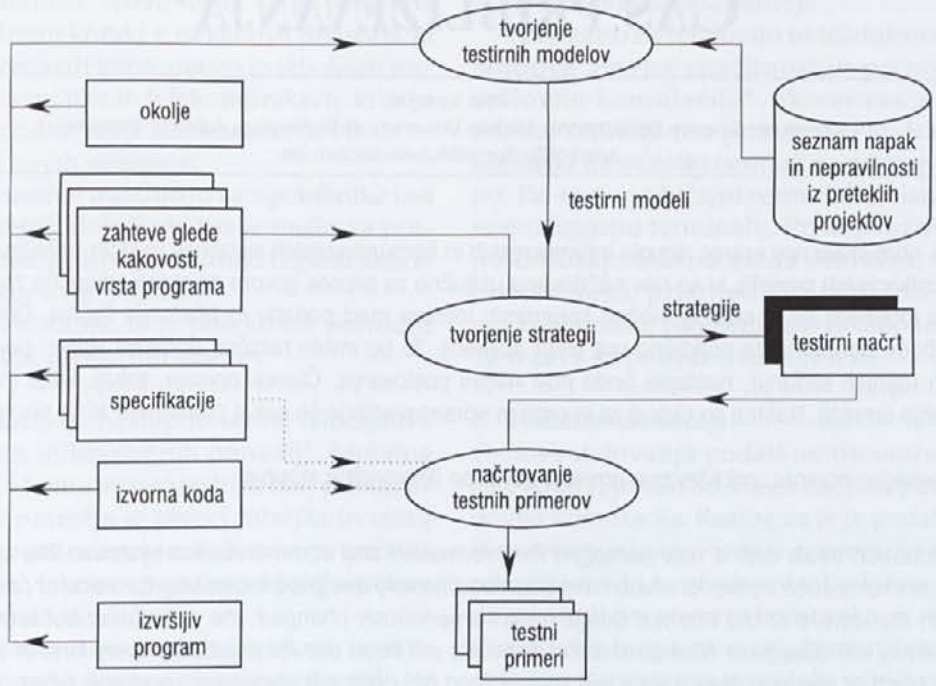
Izbor in število strategij ter testirnih modelov sta odvisna predvsem od zahtevane kakovosti produkta, razpoložljivih resursov in vrste produkta ali projekta. Splošen postopek načrtovanja testnih primerov prikazuje slika 4. Uporabljene strategije so običajno opisane v testirnem načrtu ali pa v priročniku za kakovost.

5. Sklep

Pri razvoju in kasnejšem vzdrževanju programske opreme se neprestano spreminjata izvorna koda in zahteve. Temu se mora prilagajati tudi preverjanje. Stroške načrtovanja in kasnejšega vzdrževanja testnih primerov lahko znižamo le s sistematičnim pristopom. Prikazali smo enega izmed možnih splošnih konceptov načrtovanja testnih primerov, ki temelji na testirnem modelu. Izbor in število strategij ter testirnih modelov sta odvisna predvsem od zahtevane kakovosti produkta, razpoložljivih resursov in vrste produkta ali projekta. Največ napora zahteva izbor pravega testirnega modela in njegovo vzdrževanje. Izhodišče za načrtovanje testnih primerov predstavlja predpostavka o napaki ali nepravilnosti, ki jo določimo s pomočjo modela. Za zgled smo izbrali dve preprosti in najpogosteje uporabljeni strategiji. Ideje za druge zahtevnejše modele lahko dobi preverjevalec v navedeni literaturi (npr. [BEIZER,1990], [BINDER,2000].



Slika 3: Sistematično zbiranje napak in nepravilnosti



Slika 4: Model načrtovanja testnih primerov

6. Literatura

[BEIZER,1990] B. Beizer:
"Software Testing Techniques", Van Nostrand Reinhold, New York,1990, 2. izdaja.

[BEIZER,1995] Boris Beizer:
"Black-Box Testing, Techniques for Functional Testing of Software and Systems", John Wiley and Sons, Inc., 1995.

[BINDER,2000] Robert Binder:
"Testing Object-oriented systems: models, patterns and tools", Addion Wesley Longman, Inc. 2000.

[DOGŠA,1994] T. Dogša:
"Verifikacija in validacija programske opreme", Tehniška fakulteta, Maribor, 1993.

[DOGŠA,1999,a] Tomaž Dogša:
"Dokumentiranje testnih vzorcev", Uporabna informatika, številka 1, letnik VII, 1999, str.9-16.

[DOGŠA,1999] Tomaž Dogša:
"Načrtovanje testnih vzorcev s pomočjo testirnih modelov", Zbornik osme Elektrotehniške in računalniške konference ERK '99, 23. - 25. september 1999, Portorož, Slovenija.

[DUSTIN,1999] E. Dustin, J. Rashka, J. Paul:
"Automated Software Testing: Intoduction, Management, and Performance", Addison-Wesley

[IEEE,1990b]
"IEEE Standard Glossary of Software Engineering Terminology", IEEE Std. 610.12-1990, Revision and redesignation of IEEE Std. 792-1983, The Institute of Electrical and Electronics Engineers, USA, 1990.

[JORGENSEN,1995] Paul C.Jorgensen:
"Software testing - A Craftsman's Approach", CRC Press LLC 1995.

[KANER,1993] Cem Kaner, Jack Falk, Hung Quoc Nguyen:
"Testing Computer Software", Van Nostrand Reinhold, 1993.

[MYERS ,1979] J. G. Myers:
"The Art of Software Testing", John Wiley and Sons, Inc., New York, 1979.

[ZAVERSKI,1999] Igor Zaverski, Tomaž Dogša:
"Ortogonalna klasifikacija napak programov v C++", Objektna tehnologija v Sloveniji ŠtudiĆ OTS'99 : zbornik četrtega strokovnega srečanja, Maribor, 16. in 17. junij 1999. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, Center za objektno tehnologijo, 1999, str. 209-217.

◆

Dr. Tomaž Dogša je docent na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru, kjer predava na dodiplomski in podiplomski stopnji in vodi Center za verifikacijo in validacijo sistemov. Na raziskovalnem področju se ukvarja predvsem z V&V tehnologijo, tako tudi s testirnimi orodji.