Poglavje 1

Zanesljivost programske opreme

V pričujočem poglavju si bomo ogledali metode za oceno in doseganje zanesljivosti programske opreme. Omenjene metode so se začele razvijati v sedemdesetih letih prejšnjega stoletja.

Napake v programski opremi so z razliko od strojnih komponent le *logične narave* in so v produkt vnešene v fazi razvoja bodisi kot *logične napake* izhajajoče iz napačnega razumevanja specifikacij, bodisi pa kot klasične *programerske napake*. Skozi uporabo programske opreme v eksploatacijski dobi do porojevanja novih napak praviloma ne prihaja, razen če programske opreme ne nadgrajujemo. Z razliko od strojne opreme se programska oprema ne "obrabi", lahko pa zastari glede na potrebe in zahteve uporabnika. Glede na krivuljo s tremi življenskimi fazami iz poglavja o teoriji zanesljivosti, se v starostni dobi $\lambda(t)$ ne veča, temveč ostaja konstantna. Ob upoštevanju možnosti sprotnega nadgrajevanja programske opreme je v dobi eksploatacije možno pričakovati konstantno (brez nadgrajevanja) ali stopničasto $\lambda(t)$ (z nadgrajevanjem).

Osnovna mera za kompleksnost programske opreme je število programskih vrstic izvorne kode (angl. *lines of code*, LOC). Z razmahom programske industrije v sedemdesetih in osemdesetih letih prejšnjega stoletja je veljala groba ocena, da lahko pričakujemo na 1000 vrstic izvorne kode 8 vnešenih napak [1].

Za vsak programski produkt lahko predpostavimo, da ob zaključku faze izdelave (programiranja) vsebuje N napak. Omenjeno število proizvajalcu ni znano. Ob predpostavki, da gre produkt v fazo testiranja, lahko pričakujemo odkritje in odpravo n napak ($n \leq N$). Ker proizvajalec ne pozna vrednosti N, je v fazi testiranja razpet med dilemo, ali je odkril že vse napake (ali vsaj večino njih) in lahko produkt pošlje v eksploatacijsko fazo, ali pa mora z dragim testiranjem nadaljevati.

V naslednjih razdelkih pričujočega poglavja si bomo ogledali vse dejavnike, ki tako ali drugače vplivajo na končno zanesljivost programskih produktov.

1.1 Razvojno življenski cikel programske opreme

Za programsko opremo je značilen njen razvojno - življenski cikel (angl. *software development life cycle*), ki se po [2] deli v naslednje faze:

1. *Začetek projekta*: porodi se osnovna ideja o potrebi po novem programskem produktu;

2. *Definicija osnovnih funkcij in zahtev*: pripravi se spisek osnovnih funkcij, ki naj bi jih programska oprema vršila in dodatne zahteve v smislu maksimalne velikosti eksekucijske kode, potrebe po dinamičnem in trajnem pomnenju, hitrosti odzivanja, načina hrambe podatkov, osnovne definicije bremena vključno z njegovo velikostjo, frekvenco in sestavo ter njegovimi spremembami v prihodnosti, obliko in posebnostmi interakcijskih vmesnikov itd.;

3. *Izdelava specifikacij*: funkcije in zahteve iz predhodnje alineje prenesemo v natančen specifikacijski zapis; v njem morajo biti vse funkcije in zahteve enolično definirane; končna verzija specifikacij ne sme dopuščati nikakršnih dvoumnosti v zvezi z izvedbo programske opreme, v njej pa mora biti natančno opisana tudi predvidena strojna oprema, na kateri bo tekla aplikacija; glavnina specikacij nedvoumno definira logiko realizacije ključnih funkcij programske opreme; logiko realizacije opišemo z diagrami poteka, pseudo algoritmi, podatkovnimi tokovi itd.; pri specifikacijah moramo biti pozorni na njihovo kompletnost (angl. *completness*); tipične lastnosti slabih specifikacij so njihova nepopolnost, konfliktnost posameznih delov, ali pa celo napačnost posameznih delov;

4. *Izdelava prototipa in sistemski test funkcionalnosti*: v določenih primerih za preverjanje ustreznosti specifikacij izdelamo prototip z osnovnimi funkcionalnostmi, ki ga v nadaljevanju testiramo ob prisotnosti naročnika ali potencialnega uporabnika;

5. *Revizija specifikacij*: do revizije specifikacij pride, če izdelava prototipa in njegovi sistemski testi funkcionalnosti privedejo do ugotovitev o neustreznosti delovanja predvidene programske opreme;

6. *Končno načrtovanje kode* (angl. *final design*): v tej fazi izdelamo dokončni tehnični načrt kode z vsemi potrebnimi moduli, funkcijami in procedurami v primeru strukturno proceduralnega pristopa k programiranju (angl. *structured procedural programming*, SPP), ali z vsemi potrebnimi moduli in objekti v primeru objektnega pristopa k programiranju (angl. *object oriented programming*, OOP); v to fazo sodijo tudi dokončna izbira jezika izvedbe, izbire knjižnic, razvojnih okolij, prevajalnika, operacijskega sistema itd.;

7. *Končna implementacija kode* (angl. *code final design*): v tej fazi realiziramo (sprogramiramo) izvorno kodo na osnovi predhodno navedenih alinej;

8. *Enotsko testiranje* (angl. *unit test*): v tej fazi testiramo pravilnosti delovanja posameznih modulov ali objektov programske opreme (npr. vmesnikov, decizijskih jeder itd.); hipotetično odpravimo vse najdene napake;

9. *Integracijsko testiranje* (angl. *integration test*): v tej fazi testiramo pravilnost interakcije med posameznimi moduli, funkcijami, objekti itd.; hipotetično odpravimo vse najdene napake;

10. *Sistemsko testiranje* (angl. *system test*): v tej fazi testiramo pravilnost delovanja aplikacije kot celote; hipotetično odpravimo vse najdene napake;

11. *Testiranje sprejetja* (angl. *acceptance test*): izvede se ob prisotnosti naročnika ali preizkuševalcev programskega produkta in vodi do odločitve, ali je programski produkt zrel za prehod v eksploatacijsko dobo; v primeru misijsko kritičnih aplikacij lahko novo programsko opremo postavimo za potrebe tega testa v delujoče stanje ob bok že delujoči starejši sistemski rešitvi, ki opravlja enako funkcijo; na ta način lahko primerjamo delovanje obeh sistemskih rešitev na daljši rok in na ta način pridemo do potrditve ustreznosti delovanja nove rešitve;

12. *Eksploatacijska doba* (angl. *field deployment*): programska oprema nastopi in vrši svojo funkcijo pri naročniku - kupcu;

13. *Upravljanje s programsko opremo v dobi eksploatacije* (angl. *field maintenance*): pod to fazo imamo v mislih možnost odpravljanja napak in nadgradenj; tudi v primeru, da v tej fazi nismo predvideli nadgrajevanja in odpravljanja napak, je pomembna začetna faza eksploatacije (angl. *early field deployment*), v kateri gre pričakovati nepričakovano porajanje napak in vsaj za to začetno fazo moramo imeti pripravljen scenarij načina odpravljanja le teh; omenjene napake nenazadnje izvirajo iz nepredvidenih načinov uporabe programske rešitve in eventuelnega odstopanja vrednosti vhodnih podatkov; po [2] je odprava napak v tej fazi 25 krat do 100 krat dražja od odprave napak v fazi testiranja;

14. *Redizajniranje programske opreme* (angl. *redesign*),

15. *Umik iz eksploatacije* (angl. *discard*).

V smislu finančnega vložka za izvedbo posameznih življenskih faz kažejo izkušnje velikih proizvajalcev programske opreme naslednja cenovna razmerja med fazami [2]:

- faze 1-6: 40% razvojne cene,
- faza 7: 20% razvojne cene,
- faze 8-15: 40% razvojne cene.

Delež vnešenih napak je po izkušnjah proizvajalcev skozi življenske faze sledeč:

- faze 1-6: vnos 55% napak,
- faze 7-11: vnos 40% napak,
- faze 12-15: vnos 5% napak.

1.2 Klasifikacija programskih produktov

Programske produkte in s tem posredno tudi njihove proizvajalce klasificiramo v različne razrede glede na doslednost izvedbe predhodno navedenih faz razvojno življenskega cikla programskega produkta. Ena od najbolj znanih klasifikacij je tako imenovani zmožnostno zrelostni model (angl. *capability maturity model* - CMM model) [3], ki razdeli proizvajalce in njihove produkte v pet razredov in sicer po vrsti v

1. osnovni razred (angl. *initial class*): gre za osnovni razred, v katerega sodi stihijski "ad hoc" razvoj programske opreme;
2. razred z zmožnostjo prenosa izkušenj (angl. *repeteable*): gre za razred v katerem so proizvajalci zmožni prenašati izkušnje iz razvoja predhodnih produktov v nove produkte;
3. razred z definiranim pristopom k razvoju (angl. *defined*): proizvajalci vodijo projektno organizacijo razvoja z dobro dokumentacijo;
4. razred z dobrim upravljanjem razvojnega procesa (angl. *managed*): proces razvoja je pod popolno statistično kontrolo;
5. razred s prisotnim konceptom vseskozne nadgradnje in optimizacije programske opreme (angl. *optimizing*): gre za razred, v katerega se običajno uvrščajo le proizvajalci, ki gradijo aplikacije v določeni tržni niši; nove verzije so nadgradnje starejših, produkti pa so predmet stalne optimizacije, testiranja, izboljšav itd.

Zanesljivost in posredno tudi cena produkta sta predoločeni z uvrstitivijo v razred. V višjem razredu je produkt, višja je njegova cena in praviloma tudi višja zanesljivost njegovega delovanja.

1.3 Programska redundanca

Ena od metod za zviševanje programske zanesljivosti je *programska redundanca*. Slednja v obliki paralelnega delovanja n enakih aplikacij omenjenega zvišanja ne doprinese, ker gre pričakovati, da bodo vse enake verzije programske opreme odpovedale na istem mestu (na mestu, kjer imamo v vseh verzijah isto napako). Resnici na ljubo v nekaterih primerih ni tako. V primeru, da aplikacija odpove zaradi posebnega stanja registrov gostiteljskega sistema, njegove sistemske ure itd., lahko pride do odpovedi le ene od n identičnih aplikacij.

Navkljub temu obstaja z vidika redundance rešitev, ki jo imenujemo za *n-verzijsko programiranje* [2]. Končno decizijo, ki naj bi jo vršila posamezna aplikacija, v tem primeru zaupamo n paralelno delujočim aplikacijam (n gostiteljskim sistemom), ki so napisane na osnovi enakih specifikacij, a izhajajo kot produkt iz različnih programerskih hiš. Napako v tovrstnih sistemih identificiramo po principu glasovalnih tehnik in jo na ta način tudi eliminiramo. Metoda n verzijskega programiranja ima naslednje značilnosti:

- izdelava n verzij programske opreme zahteva plačilo n-kratne cene razvoja programske opreme,
- n verzij naj bi bilo po funkcionalni plati ekvivalentnih,
- pri vzporednem razvoju n aplikacij ne sme priti do komunikacije med različnimi izvajalci, da ne bi prihajalo do skupinskega napačnega interpretiranja specifikacij,
- aplikacije naj bi bile po možnosti razvite z različnimi razvojnimi orodji,
- posamezni problemi iz specifikacij naj bi bili rešeni na osnovi različnih algoritmov (npr. različni pristopi k iskanju ničel kvadratnih enačb),
- vseh n verzij programske opreme mora iti skozi enake testne postopke.

Tovrstne metode so zaradi svoje visoke cene uporabljane samo v visoko zanesljivih sistemih, ki vršijo kritične misije.

1.4 "Roll back and recovery" metode

Ena od pogosteje uporabljanih tehnik za dvig zanesljivosti programske opreme so "*roll back and recovery*" metode [2]. Temeljijo na sprotni detekciji napak in ponovitvi določenega števila procesnih korakov. Slednji pristop je uspešen, če je porajajoča napaka minljiva, netrajna ali prehodna (angl. *transient error*). Napaka se tako hipotetično pojavi kot posledica stanja sistema ali vpliva iz okolja, pri čemer upamo, da se ob ponovitvi procesiranja zaradi novega globalnega stanja sistema ali konca neželjenega vpliva iz okolja ne bo ponovila. Metode delimo na dve podskupini in sicer na

- "*forward error recovery*": napake se v tem primeru zavedamo, a procesiramo naprej (zgled bi bil sledenje letalu v kontroli letenja), pri čemer predpostavljamo, da je v prihodnjih procesnih korakih ne bo več,
- "*backward error recovery*": napaka v procesiranju nastopi in procesiranje "prevrtimo" za nekaj korakov nazaj.

V praksi so v uporabi najpogosteje drugo navedene metode, ki jih glede na število vzvratnih korakov ločujemo na naslednje skupine:

- "*reboot*" ali "*restart*" tehnike: omenjene tehnike vračajo sistem v začetno stanje, kar nam vzame relativno veliko časa za ponoven zagon sistema in aplikacije ter njeno izvajanje do točke odpovedi; drugi problem tehnike je v tem, da izgubimo kopico delovnih podatkov;
- "*recovery*" tehnike: v slednjem primeru je izguba časa manjša, saj se ponovno naloži le aplikacija in opcijsko del operacijskega sistema;
- *žurnalske* tehnike: omenjene tehnike beležijo procesne korake vključno z vhodnimi podatki; metoda je zaradi veliko pomnilne obremenitve najkompleksnejša, ključno vprašanje, ki pa se nam postavlja v tem primeru je, kolikšni naj bosta globina pomnjenja zgodovine dogodkov in gostota beleženja procesnih korakov; če ne shranjujemo vseh procesnih korakov govorimo o metodi "*check painting*"-a;
- "retry" tehnike: temeljijo na ponavljanju posameznega procesnega koraka vse do njegove uspešne izvedbe;

1.5 Osnove testiranja programske opreme

Eden od temeljev za zagotavljanje zanesljivosti programske opreme je odprava čimvečjega števila napak pred nastopom eksploatacijske dobe, kar dosežemo s *testiranjem*. Popolnih postopkov testiranja v večini primerov ne moremo zagotoviti, ker je

- število različnih možnih vhodnih podatkov preveliko,
- število različnih izhodnih podatkov preveliko in
- število različnih možnih poti skozi program preveliko.

Testiranje v večini primerov ne more dokazati, da napak v programski opremi ni, lahko pa dokaže (in eliminira) pristotnost le teh. Izkušnje pri testiranju po [3] kažejo na naslednja dejstva:

- večje število napak najdemo, več jih v programu verjetno še obstaja,
- bolj ko testiramo programsko opremo, bolj postaja le ta imuna na naše teste (pesticidni paradoks) in
- produktne specifikacije večinoma niso nikdar končane.

Poleg enotskih, integracijskih, sistemskih testov in testov sprejetja, ki smo jih že opisali, obstaja še množica drugačnih klasifikacij testiranja. Posebni vrsti testiranj sta testiranji preživetja (angl. *test to pass*) in odpovedi (angl. *test to fail*) aplikacije. V prvem primeru je testiranje namenjeno preverjanju ali aplikacija deluje, v drugem primeru pa preverjanju ali aplikacija lahko odpove in pod kakšnimi pogoji to dosežemo.

Glede na izvajalca testiranja ločujemo med testi, ki jih izvaja razvijalec, med testi, ki jih izvaja končni naročnik in testi, ki jih izvaja tretja stranka (angl. *third party testing*). Slednji način je zanimiv predvsem zato, ker je testiranje izvedeno bolj neodvisno. Naročnik testa je lahko razvijalec, ali pa naročnik programske rešitve. Posebna primera testiranj komercialno široko razvejanih programskih produktov sta tako imenova "alfa" in "beta" testiranje. Uporabljata se v zadnji fazi testiranja širokopotrošnih programskih produktov, pri čemer prvega izvaja razvijalec sam, drugega pa množica izbranih uporabnikov, ki niso povezani z razvijalcem.

Vse naštete pristope k testiranju razdelimo po [3] na štiri skupine in sicer na

- *statično "black box" testiranje* (angl. *static black box testing*),
- *dinamično "black box" testiranje* (angl. *dynamic black box testing*),
- *statično "white box" testiranje* (angl. *static white box testing*) in
- *dinamično "white box" testiranje* (angl. *dynamic white box testing*).

Pri tem nam termin *statičnega* predstavlja kakršnokoli testiranje programske opreme, pri katerem se programska oprema ne izvaja, termin *dinamičnega* pa testiranje, kjer programsko opremo izvajamo. Termin "Black box" testiranja pomeni, da vpogleda v izvorno kodo nimamo na razpolago, termin "White box" testiranja pa, da imamo na razpolago vpogled v izvorno kodo programske opreme.

Z vidika testiranja ločujemo med pojmoma *verifikacije* in *validacije*. Prvi pojem predstavlja preverjanje, ali programska oprema zadošča specifikacijam, drugi pa preverjanje, ali programska oprema zadošča potrebam uporabnika.

1.5.1 Statično "Black box" testiranje

Iz imena metode glede na predhodnje razlage sledi, da v tem primeru programa ne izvajamo, niti nimamo vpogleda v njegovo izvorno kodo. V praksi gre za testiranje ali natančen pregled specifikacij.

1.5.2 Dinamično "Black box" testiranje

Tudi v tem primeru vpogleda v kodo nimamo, jo imamo pa možnost izvajati. V tej situaciji smo soočeni s testiranjem obnašanja aplikacije. Ključno vlogo v tem primeru igrajo *testni primeri* ali *testni profili* (angl. *test cases* ali *test profiles*). Gre za vnaprej pripravljene množice vhodnih podatkov, s katerimi preverjamo pravilnost delovanja. Določitev števila in vsebine posameznih množic je izredno pomembna, saj nas lahko vodi do preveč ali premalo testiranja, pa tudi do situacije, ko testiramo napačne segmente programske opreme. Z vidika sestave testnih profilov se na začetku nagibamo k "test to pass" strategiji, ki v profile uvršča vhodne podatke, s katerimi skušamo dokazati, da program deluje pravilno, proti koncu pa k "test to fail" strategiji, s katero skušamo dokazati, da program v nekih mejnih situacijah deluje nepravilno. Pri formiranju

testnih profilov se moramo držati pravil *ekvivalenčnega particioniranja*. Slednje pomeni, da posamezne testne množice vhodnih podatkov pripravljamo tako, da v iste množice uvrstimo sorodne vhodne podatke, ki skušajo odkriti določen tip eventuelne napake. Oglejmo si navedeno na zgledu okenske aplikacije, ki vrši funkcijo enostavnega kalkulatorja. Tako bi lahko testni profili (množice) po vrsti bili

$$T_1 = \{1+1, 1+2, 1+3, ...\}, \tag{1.1}$$

$$T_2 = \left\{1+9...9, \frac{1}{0}, ...\right\}, \tag{1.2}$$

$$T_3 = \{Ctrl + C, Ctrl + Shift + C, ...\}. \tag{1.3}$$

Pri tem prvi testni profil T_1 preverja "odziv" na enostavne računske operacije, T_2 "odziv" na mejne računske probleme (npr. pogoj prekoračitve), T_3 pa odziv na kontrolne okenske ukaze. Testiranje izvedemo le za izbrane reprezentativne elemente posamezne particijske množice. Tako s particioniranjem skušamo določiti po eni strani vse reprezentativne zglede vhodnih podatkov, po drugi strani pa količino testiranja obdržati v nekem zmernem obsegu.

Z vidika usmeritve testiranja ločujemo med *podatkovno usmerjenim testiranjem* (angl *data testing*) in *testiranjem programskega toka* (angl. *program flow testing*).

Izvajanje podatkovno usmerjenega testiranja temelji predvsem na ustrezni definiciji ekvivalenčnih particij (množic). Slednje se usmerjajo na podatkovne strukture (kazalci, konec - začetek polja, nizi, itd.), same množice pa lahko delimo tudi na cilje testiranja, ki vodijo od preživetvenih pa vse do tistih, ki bi radi delovanje programa za vsako ceno "zlomili" (angl. *test to pass*, *boundary test*, *test to fail*).

Izvajanje testiranja programskega toka temelji na množici stanj, ki jih lahko zavzame programska oprema in na povezavah, ki vodijo med njimi. Pri tem igra pomembno vlogo tudi verjetnost nahajanja v posameznem stanju. Večja je verjetnost posameznega stanja, več testiranja moramo nameniti potem, ki vodijo iz tega stanja ali v to stanje. Tudi stanja, povezave med njimi in njihove verjetnosti vgrajujemo v testne profile. Ker je število stanj običajno izredno veliko (nenazadnje tudi zaradi stanj strojne opreme gostiteljskega sistema) je vse poti enostavno nemogoče stestirati. Osnova za temeljito testiranje je diagram prehajanja med stanji (angl. *state transition diagram*). V njem so stanja, povezave med njimi, verjetnosti posameznih stanj in vhodni podatki, ki povzročijo posamezne prehode med stanji. Na osnovi omenjenega diagrama so tendence pravil testiranja po [3] sledeče:

- vsako stanje testiraj vsaj enkrat, pri čemer način prihoda v stanje ni važen; povedano drugače, naj bi testiranje izvedlo vsaj en "uporabniški" obhod po diagramu stanj;
- testiraj le najbolj verjetna prehajanja med stanji;
- testiraj "najmanj običajne" povezave med stanji;

- testiraj vse potencialne "napačne" interakcije in vračanja iz njih;
- testiraj naključno izbrana prehajanja med stanji.

Na tem mestu se lahko poslužujemo tudi *repeticijskega testiranja* za odkrivanje pomnilniških razpok (angl. *memory leaks*), *stresnega testiranja* ob prisotnosti minimalne pristonosti resursov kot sta npr. CPU ter disk in *bremenskega testiranja*, ker skušamo aplikaciji na vhod dovesti maksimalne obremenitve (npr. velike vhodne datoteke, velika računska bremena itd.).

1.5.3 Statično "White box" testiranje

V primeru statičnega "White box" testiranja imamo vpogled v izvorno kodo aplikacije, ki pa je ne izvajamo. Omenjena metoda se izvaja predvsem pri večjih in misijsko bolj kritičnih projektih. Metoda vsebuje množico verifikacijskih postopkov (npr. strukturalno analizo) za dokazovanje pravilnosti kode. Zapis kode se ocenjuje tudi skozi standarde in priporočila. Zgled slednjih so izogibanje "nevarnim" konstruktom (npr. že zastarelim programskim ukazom, kot so GO TO, PEEK, POKE, DO WHILE itd.) programskih jezikov. Standarde za omenjena področja gradijo neodvisne organizacije kot so American National Standards Institute (ANSI), Association for Computer Machinery (ACM), organizacija IEEE itd. Vrste napak, ki naj bi jih odkrili v tej fazi, so sledeče:

- podatkovno referenčne napake (angl. *data reference errors*),
- podatkovno deklaracijske napake (angl. *data declaration errors*),
- računske napake (npr. prirejanje različnih tipov itd.),
- primerjalne napake,
- napake toka korakov (postavitev begin-end sekvenc, do-while sekvenc itd.),
- napake v parametrih procedur in objektnih tipov,
- I/O napake (branje iz datotek, perifernih naprav, itd.).

Metoda statičnega "White box" testiranja je podprta tudi z ustreznimi programskimi orodji, ki vršijo analizo izvorne kode in opozarjajo na eventuelne napake. Del teh funkcionalnosti podpirajo tudi sami prevajalniki.

1.5.4 Dinamično "White box" testiranje

V tem primeru imamo opravka tako z možnostjo izvajanja kode po korakih, kot tudi vpogled vanjo. Dinamično "White box" testiranje imenujemo tudi za *strukturalno testiranje*, ker imamo vpogled v celotno strukturo programa. Z vidika povedanega vršimo štiri različne funkcije:

- testiranje funkcij, procedur, knjižnic, aplikacijskih programskih vmesnikov, objektov itd., kar deloma sovpada z enotskim testiranjem;

- testiranje na krovnem nivoju, kar deloma sovpada s sistemskim testiranjem;
- vpogled v spremenljivke in stanja programa;
- vpogled v to, katero kodo s testi zaobjamemo, katere pa ne; slednje lahko vpliva na korekcijo testnih profilov;

Pomembno je, da dinamičnega "White box" testiranja neposredno ne enačimo z "debuggingom". Cilj prvega je najdba napake, cilj drugega pa njena odprava.

1.5.5 Avtomatizacija postopkov testiranja

Na tržišču je prisotnih vse več programskih orodij, ki samodejno vršijo testiranje programske opreme. Delimo jih v dve skupini in sicer na *invazivna* in *neinvazivna* orodja. Prva omogočajo tudi popravke izvorne kode, druga pa ne. Glede na njihove zmožnosti orodja razdelimo na naslednje skupine, pri čemer kompleksnost orodij narašča z vrstnim redom navajanja:

- *monitorji*: omenjena orodja so sposobna prikaza katere vrstice izvorne kode so bile izvajane, katere poti po programski opremi so bile prehojene itd.; tovrstna orodja imenujemo tudi za kodno pokrivne analizatorje (angl. *code coverage analyzer*);
- *gonilniki* (angl. *drivers*): za gonilnike imenujemo orodja, ki so sposobna krmiliti testirane aplikacije (npr. dostavljati vhodne podatke testirani aplikaciji);
- *sprejemniki* (angl. *stubs*): za sprejemnike imenujemo orodja, ki so sposobna sprejemati odzive testirane aplikacije, jih ustrezno analizirati in formirati poročilo o testiranju;
- *stresna* in *bremenska orodja*: omenjena orodja nam omogočajo umetno zniževati razpoložljive resurse, ali povzročati umetna aplikacijska bremena;
- *generatorji šuma*: omenjena orodja znajo generirati pomensko spremenljiv šum v aplikacijskem okolju (npr. generiranje šuma na komunikacijskem kanalu);
- *snemalniki* in *predvajalniki*: omenjena orodja so sposobna beleženja interakcije s testirano aplikacijo in ponavljanja vhodnih obremenitev z različnimi frekvencami pojavitev (hitrost predvajanja posnetka);
- *programirani makroji*: omenjena orodja znajo pognati testirano aplikacijo, ji v časovnih sekvencah podtikati vhode in brati aplikacijske odzive; večinoma delujejo po principu programiranega števila ponovitev; možnosti vejanj običajno niso integrirane;
- poljubno *programabilna avtomatizirana testna orodja*;

Ena od možnih strategij izvajanja postopkov testiranja so tudi avtomatizirani "monkey" testi. Njihovo ključno vodilo je, da skušajo v smislu interakcije s testirano aplikacijo oponašati uporabnika. Osnovna ideja "monkey" testov leži v veliki intenzivnosti interakcije s testirano aplikacijo. Ideja izhaja iz hipotetične predpostavke, da če pred 10^6 osebnih računalnikov posedemo 10^6 opic, ki bodo tipkale 10^6 delovnih ur, obstaja že nezanemarljiva verjetnost, da bo vsaj en podsegment teksta enak ali podoben kakemu znanemu literarnemu delu. "Monkey" teste razdelimo na tri skupine in sicer na

- "Dumb monkey" teste: navedeni testi oponašajo najbolj neukega uporabnika; v večini primerov lahko pričakujemo hitro odpoved testirane aplikacije nenazadnje tudi zaradi izredno velike frekvence interakcij; tipične razloge za odpovedi v tem primeru lahko iščemo v pomnilniških razpokah (angl. *memory leaks*) in slabih specifikacijah testirane programske opreme;

- "Semi smart monkey" teste: omenjena orodja so kompleksnejša saj z razliko od predhodno navedenih vršijo beleženje procesnih korakov in omogočajo ponovne samodejne zagone testirane aplikacije;

- "Smart monkey" teste: najkompleksnejša avtomatizirana orodja, ki vedo o aplikaciji toliko, kot ve aplikacija sama o sebi; uporabnik ali „monkey“ ve, kje v programu se nahaja, od kod je prišel, kam lahko gre in ali so odzivi testirane aplikacije pravilni; običajno je osnova za takšno orodje diagram prehajanja stanj testirane programske opreme;

1.6 Metrike za ocenjevanje kompleksnosti programske kode

Z navedenimi metrikami skušamo oceniti kompleksnost opazovane programske izvorne kode. Za primer naštejmo metrike, ki jih za ocenjevanje izvorne kode priporoča Microsoft v okviru svojega razvojnega okolja Visual Studio 2013 [4]:

- *MI indeks* (angl. *maintainability index* - MI): metoda podpira izračun MI indeksa (vrednosti med 0 in 100), ki predstavlja stopnjo možnosti obvladovanja programske opreme; večji je indeks, večja je predvidena stopnja možnosti obvladovanja;

- *ciklomatična kompleksnost* (angl. *cyclomatic complexity*): metoda izmeri ciklomatično kompleksnost izvorne kode; premo sorazmerno je pogojena s številom različnih poti, ki vodijo skozi program in s tem posredno odvisna od števila vejanj; večja je ciklomatična kompleksnost programa, daljše bo moralo biti testiranje in manjša bo možnost obvladovanja programske opreme; začetnik na področju raziskav ciklomatičnih metrik je McCabe, ki l.1976 vpelje prvo tovrstno metriko [1];

- *globina dedovanja* (angl. *depth of inheritance*): metoda oceni globino dedovanja pri implementiranih objektih; večja je globina dedovanja, manjša je možnost obvladovanja programske opreme;

- *povezovanje razredov* (angl. *class coupling*): metoda meri povezanost razredov preko parametrov, lokalnih spremenljivk, klicev metod itd.; večja je povezanost, manjša je možnost obvladovanja programske opreme in njene ponovne uporabe (angl. *reuse*) zaradi medsebojne odvisnosti.

Izbira ocenjevalne metrike je odvisna od paradigme programiranja (SPP ali OOP), programskega jezika in razvojnega okolja. Ovrednotenje metrik je običajno integrirano v samo razvojno okolje in je avtomatizirano. Obstajajo pa tudi posebna orodja, ki ponujajo samo izračune metrik, pri čemer predstavlja vhod v takšno orodje izvorna programska koda. Primer takšnega orodja je TestWell proizvajalca VerifySoft Inc., ki ponuja izračune metrik za programsko kodo pisano v jezikih C, C++ in Java [5].

1.7 Modeli za ocenjevanje preostalega števila napak v programski opremi

Osnova ideja tovrstnih modelov izhaja iz dejstva, da nikdar ne moremo dokazati, da v novo nastali programski opremi ni napak. Predpostavimo, da smo programsko opremo po njenem dokončanju testirali od časovne točke t_0 do časovne točke t_1 ($t_0 \leq t_1$) in v času testiranja našli in hipno odpravili n napak. Če predpostavimo, da je bilo začetno, a sicer nepoznano število napak N, imamo v časovni točki t_1 torej neodkritih še N-n napak. V časovni točki t_1 smo v dilemi ali s testiranjem nadaljevati, ali pa v tej časovni točki s testiranjem zaključiti. Na našo odločitev bi seveda vplivalo poznavanje števila N-n, ki pa je zaradi neznane vrednosti spremenljivke N neznano.

V ta namen se v sedemdesetih letih prejšnjega stoletja začnejo razvijati modeli za oceno ali predikcijo začetnega števila napak N v programski opremi. Modeli za ocenjevanje ali napovedovanje preostalega števila napak se v praksi uporabljajo predvsem na področju misijsko kritičnih aplikacij. Razlogi za to so sledeči:

- njihova uporaba je časovno potratna in s tem posredno draga,

- izbira modelov je velika in težko je izbrati pravega,

- eden od pogledov razvojnikov programskih produktov je ta, da modelov ni potrebno uporabljati, ker so bile za razvoj uporabljane moderne razvojne tehnologije,

- z razvojem komunikacij med napravami, na katerih tečejo programski produkti, se razvije možnost cenenega odpravljanja napak aplikacij v dobi eksploatacije (angl. *early field deployment*).

V nadaljevanju pričujočega razdelka si bomo ogledali dva primera tovrstnih modelov. Oba sta nastala v sedemdesetih letih prejšnjega stoletja in danes nista več aktualna, sta pa zanimiva z vidika načina določitve števila napak N v programski opremi.

1.7.1 Halsteadova metrika

M. H. Halstead predstavi svojo metriko kompleksnosti programskega produkta leta 1977 [1]. Za njeno definicijo vpelje naslednje spremenljivke:

- k_1: število različnih operacij v izvorni kodi,
- k_2: število različnih operandov v izvorni kodi,
- K_1: število vseh operacij v izvorni kodi in
- K_2: število vseh operandov v izvorni kodi.

Odtod Halstead vpelje metrike programske kompleksnosti, ki jih po vrsti imenuje za *dolžino programa* K, *dolžino programskega besednjaka* k in *obseg programa* V. Po vrsti jih definira z izrazi

$$K = K_1 + K_2, \tag{1.4}$$

$$k = k_1 + k_2, \tag{1.5}$$

$$V = K * \log_2 k. \tag{1.6}$$

Halstead na osnovi empiričnih rezultatov meritev po [1] definira *pričakovano število napak v programu* N pred testiranjem na dva načina in sicer z izrazoma

$$N = \frac{V}{3.000}, \tag{1.7}$$

$$N = \frac{A}{3.000}, \tag{1.8}$$

pri čemer je A definiran kot

$$A = \left(\frac{V}{\frac{2k_2}{k_1 K_2}} \right)^{\frac{2}{3}}. \tag{1.9}$$

Empiričnost obeh ocen izhaja iz analize programskih produktov pisanih s takrat aktualnima programskima jezikoma Cobol in Fortran, ki sta bila v sedemdesetih letih prejšnjega stoletja najbolj razširjena. Prvi jezik se je uporabljal za pisanje poslovnih aplikacij, drugi pa za pisanje računsko usmerjenih inženirskih aplikacij. Na tem mestu posebej poudarimo, da je Halsteadova metrika popolnoma *deterministične narave* in pridobljena empirično.

1.7.2 Jelinski Moranda model

Jelinski Moranda model se pojavi leta 1972 in predstavlja enega od prvih modelov za ocenjevanje števila napak N [1], [6]. Predpostavke ob uporabi navedenega modela so sledeče:

- program na samem začetku testiranja vsebuje N napak, pri čemer je N neznano število; testiranje vršimo na časovnem intervalu $[t_0, t_n]$;
- vse napake v programu so med seboj neodvisne in nastopajo (se manifestirajo) z enako intenzivnostjo;
- časovne točke najdb napak na intervalu $[t_0, t_n]$ so med seboj neodvisne; napake se odpravijo hipno in idealno (napaka je zagotovo odpravljena);
- intenzivnost odpovedovanja je na časovnem podintervalu med dvema najdenima napakama konstantna in premo sorazmerna številu preostalih (še ne najdenih) napak.

Potrebno je omeniti, da predhodno naštete predpostavke niso najbolj realistične. Napake med seboj praviloma niso neodvisne in se ne manifestirajo z enako intenzivnostjo. Tudi njihova odprava običajno ni hipna in idealna - za popravke običajno ne moremo trditi z gotovostjo, da napako odpravljajo.

Ob sprejetju predpostavk v predhodnjih alineah zapišimo izraz

$$\lambda(t_i) = \phi * [N - (i-1)], i = 1, 2, ..., N, \tag{1.10}$$

kjer t_i predstavlja čas med najdbo $(i-1)$-te in i-te napake, $\lambda(t_i)$ intenzivnost odpovedovanja na omenjenem časovnem intervalu, N začetno število napak v programu in ϕ doprinos posamezne napake, pri čemer je navedeni doprinos za vse napake enak. Funkcija $\lambda(t_i)$ je tako stopničasto padajoče narave, pri čemer je višina stopnice (ϕ) konstantna, dolžina stopnice pa variabilna (odvisna od dolžine časovnega intervala med najdbama $(i-1)$-te in i-te napake. Slednje bi lahko zapisali z zaporedjem izrazov

$$\lambda(t_1) = \phi * [N], \lambda(t_2) = \phi * [N-1], ... \tag{1.11}$$

Zanesljivostne funkcije posameznih odsekov s konstantno intenzivnostjo odpovedovanja bi lahko zapisali z izrazom

$$R(t) = e^{-\phi * (N-(i-1)) * t}, t \in [t_{i-1,e}, t_{i,e}], \tag{1.12}$$

pri čemer $t_{i-1,e}$ predstavlja čas najdbe $(i-1)$-te napake, $t_{i,e}$ čas najdbe i-te napake in $(i-1)$ najdeno število napak do nastopa časovnega intervala t_i.

Na osnovi MLE metode (angl. *maximum likelihood*) pridemo do izrazov

$$n \ln \frac{(N-1) * \sum_{i=1}^{n} t_i - \sum_{i=1}^{n} (i-1) t_i}{N * \sum_{i=1}^{n} t_i - \sum_{i=1}^{n} (i-1) * t_i} + \ln \frac{N}{n-1} \geq 0, \tag{1.13}$$

$$n \ln \frac{(N+1) * \sum_{i=1}^{n} t_i - \sum_{i=1}^{n} (i-1)t_i}{N * \sum_{i=1}^{n} t_i - \sum_{i=1}^{n} (i-1) * t_i} + \ln \frac{N-n+1}{N+1} \geq 0, \tag{1.14}$$

s katerima lahko pridobimo oceno za N, pri čemer mora izbrani N izpolniti obe predhodno navedeni relaciji. Odtod lahko pridobimo tudi oceno faktorja ϕ po izrazu

$$\phi = \frac{n}{N * \sum_{i=1}^{n} t_i - \sum_{i=1}^{n} (i-1) * t_i}. \tag{1.15}$$

Več podatkov o izpeljavi predhodnih izrazov po MLE metodi bralec lahko najde v delih [1], [6].

Zgled 1 *Predpostavimo, da smo pri testiranju programske opreme zabeležili naslednje čase najdb 26 različnih programskih napak*

$$T = [9, 21, 32, 36, 43, 45, 50, 58, 63, 70, 71, 77, 78, 87, 91, 92, 95, 98, ...] \tag{1.16}$$

$$T = [..., 104, 105, 116, 149, 156, 247, 249, 250] \tag{1.17}$$

Ob upoštevanju izrazov

$$\sum_{i=1}^{n} t_i = 250, \sum_{i=1}^{n} (i-1) * t_i = 4020, \tag{1.18}$$

začnemo z iteracijo določanja vrednosti N s poizkušanjem ($N = 1, 2, ..$). Pri vrednosti $N = 32$ ni izpolnjena 1.relacija, pri $N = 33$ pa 2.relacija. Iz tega sledi, da je ocenjena vrednost nekje med 32 in 33 napakami. Odtod lahko preizkušanje relacij nadaljujemo z realnimi števili med navedenima vrednostima. Ocena, ki zadošča obema relacijama je $N = 32, 2$. Na osnovi ocene za N lahko ocenimo tudi ϕ in sicer dobimo vrednost

$$\phi = \frac{26}{32, 2 * 250 - 4020} = 0, 0065. \tag{1.19}$$

1.8 Programerska pravila za produkcijo varne kode

Za doseganje čim manjšega števila napak v programski opremi se za potrebe področja izdelave programske opreme misijsko kritičnih aplikacij formirajo pravila za produkcijo t.i. *varne kode* (angl. *code safety*). Pravila agencije NASA na tem področju so sledeča [7], [8]:

1. vsi segmenti usmerjanja toka programa naj bodo enostavni in brez skočnih instrukcij (`goto`, `setjmp`, `longjmp` itd.); uporaba rekurzije je prepovedana;

2. vse zanke morajo imeti določeno gornjo mejo števila izvajanj; s tem preprečimo neskončno izvajanje zanke in vse neželjene posledice njenega izvajanja; obvezno statično testiranje kode nas mora privesti do zaključka, da se število izvajanj zanke ne povzpne nad predvideno mejo števila izvajanj; če do te ugotovitve ne pridemo, s tem kršimo navedeno pravilo;

3. uporaba dinamične alokacije pomnilnika skozi čas delovanja aplikacije je prepovedana;

4. dolžina funkcij ob enovrstičnem navajanju ukazov in deklaracij je omejena na „eno stran potiskanega papirja“ (ne več kot 60 vrstic izvorne kode); s tem se izognemo kompleksnim in s tem posredno nepreglednim funkcijam;

5. gostota vstavkov (angl. *assertion density*) v obliki preverjanih trditev (angl. *assertion*) mora biti navzgor omejena (npr. maksimalno dva vstavka na funkcijo); preko vstavkov programer kontrolira resničnost na prvi pogled sicer veljavnih dejstev; s tem preverjamo delovanje programa v anomaličnih situacijah (npr. prekoračitvi količine dinamičnega pomnilnika); vstavki morajo biti logične True/False narave;

6. podatkovni objekti morajo biti deklarirani na najmanjšem možnem nivoju obsega;

7. za vse funkcije, ki naj bi vračale vrednosti (angl. *non-void functions*), je potrebno preveriti vrnjene vrednosti s točke klica funkcije; znotraj vsake funkcije je potrebno preveriti veljavnost parametrov, preko katerih je bila funkcija klicana;

8. pri prevajanju mora biti delovanje predprocesorja (angl. *preprocessor*) omejeno le na vstavljanje „header“ datotek in enostavnih makro definicij;

9. uporaba kazalcev ni zaželjena in je omejena; dereferenciranje kazalca (sklic na podatkovno vsebino, na katero kaže kazalec) je lahko le enonivojsko;

10. vsa izvorna koda mora biti prevedena po vsakem popravku; pri tem mora imeti prevajalnik vključeno tudi opozarjanje (angl. *warning messages*), pri čemer se mora prevajanje končati ne samo brez napak (angl. *errors*), temveč tudi brez opozoril; izvorna koda mora biti istočasno tudi redno preverjana s kodnim analizatorjem in tudi slednji naj ne bi poročal o napakah ali opozorilih;

NASA večino svoje izvorne kode za misijsko kritične sisteme v zadnjem desetletju razvija v zbirniku in programskem jeziku C, tako da gre vsa navedena pravila razumevati skozi prizmo ukazov tega programskega jezika. Njena izvorna koda se običajno prevaja in ne intepretira (tolmači) ter piše strukturalno (modularno) in ne objektno. Prevajana koda je z vidika izvajanja hitrejša, kar ji omogoča dovolj hitro odzivanje v sistemih, ki morajo teči v realnem času.

Literatura

[1] H. Pham, *Software reliability.* Springer - Verlag, Singapore, 2000.

[2] M. L. Shooman, *Reliability of computer systems and networks: fault tolerance, analysis, and design.* J.Wiley and Sons, 2002.

[3] R. Patton, *Software testing.* Sams, USA, 2000.

[4] "Code metrics values - Microsoft Visual studio 2013." `https://msdn.microsoft.com/en-us/library/bb385914.aspx/`, April 2015.

[5] "Code Complexity Measures." `http://www.verifysoft.com/en_cmtx.html/`, April 2015.

[6] J. Virant, *Zanesljivost računalniških sistemov.* Založba FE in FRI, Ljubljana, Slovenija, 1993.

[7] "NASA's rules for code safety." `https://jaxenter.com/power-ten-nasas-coding-commandments-114124.html`, Marec 2019.

[8] "G.J.Holzmann: The Power of Ten – Rules for Developing Safety Critical Code." `http://pixelscommander.com/wp-content/uploads/2014/12/P10.pdf`, Marec 2019.