UNIVERSITATEA
DIN BUCUREȘTI

Metode de dezvoltare software

Testare - partea 1

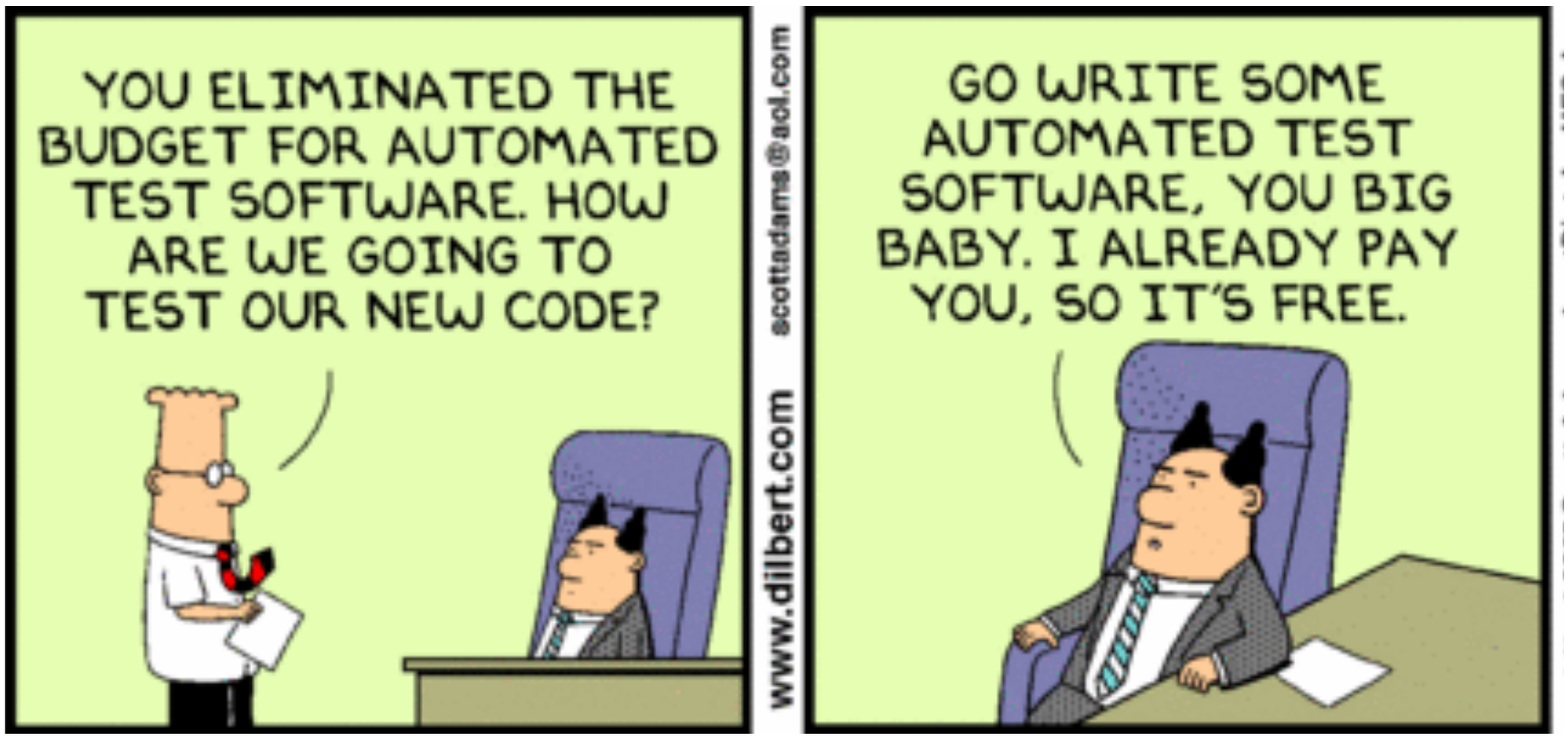
10.04.2017

Alin Ștefănescu

Testare software

Prezentare bazată pe materiale de Florentin Ipate (UniBuc) și Florin Leon (UT Iași)

Testare… în practică



de la dilbert.com

Generalități - validare și verificare (V&V)

Verificare

- construim corect produsul?
- se referă la dezvoltarea produsului

Validare

- construim produsul corect?
- se referă la respectarea specificaţiilor, la utilitatea produsului

- Verificarea şi validarea trebuie să stabilească încrederea că produsul este potrivit pentru scopul său
- Aceasta ***nu*** înseamnă că produsul este lipsit de defecte
- Doar că produsul trebuie să fie suficient de bun pentru utilizare

Evaluarea unui produs software

Depinde de scopul produsului, de aşteptările utilizatorilor şi factorii de piaţă:

- Funcţionalitatea programului
  - nivelul de încredere depinde de cât de critic este sistemul pentru utilizatori
- Aşteptările utilizatorilor
  - utilizatorii pot avea grade diferite de aşteptări pentru diferite tipuri de produse software
- Mediul de afaceri
  - lansarea rapidă pe piaţă a unui produs poate fi uneori mai importantă decât găsirea tuturor defectelor în program

Testarea unui program

- evidenţiază prezenţa erorilor şi *nu* absenţa lor
- este singura tehnică de validare pentru cerinţe non-funcţionale, deoarece programul trebuie executat pentru a i se analiza comportamentul
- este utilizată de obicei alături de verificarea statică (*static analysis*) pentru o siguranţă cât mai mare

Terminologie (IEEE)

- Eroare (engl. “*error*”)
  - o acţiune umană care are ca rezultat un defect în produsul software
- Defect (engl. “*fault*”)
  - consecinţa unei erori în produsul software
  - un defect poate fi latent: nu cauzează probleme cât timp nu apar condiţiile care determină execuţia anumitor linii de cod
- Defecţiune (engl. “*failure*”)
  - manifestarea unui defect: când execuţia programului întâlneşte un defect, acesta provoacă o defecţiune
  - abaterea programului de la comportamentul aşteptat

Bug

“Bug”: termen colocvial utilizat deseori ca sinonim pentru „defect”

de la dilbert.com

Testarea şi depanarea

- testarea de validare
  - intenţionează să arate că produsul nu îndeplineşte cerinţele
  - testele încearcă să arate că o cerinţă nu a fost implementată adecvat
- testarea defectelor
  - teste proiectate să descopere prezenţa defectelor în sistem
  - testele încearcă să descopere defecte
- depanarea (“*debugging*”)
  - are ca scop localizarea şi repararea erorilor corespunzătoare
  - implică formularea unor ipoteze asupra comportamentului programului, corectarea defectelor şi apoi re-testarea programului

Asigurarea calității

- testarea se referă la detectarea defectelor
- **asigurarea calităţii** se referă la prevenirea lor
  - se ocupă de procesele de dezvoltare care să conducă la producerea unui software de calitate
  - include procesul de testare a produsului

Câteva principii de testare

- o parte necesară a unui caz de test este definirea **ieşirii** sau **rezultatului** aşteptat

- programatorii nu ar trebui să-şi testeze propriile programe (excepție face testarea de nivel foarte jos - testarea unitară)

- ideal, organizaţiile ar trebui să folosească și companii (sau departamente) externe pentru testarea propriilor programe

- rezultatele testelor trebuie analizate amănunțit

- trebuie scrise cazuri de test atât pentru condiţii de intrare invalide şi neaşteptate, cât şi pentru condiţii de intrare valide şi aşteptate

Câteva principii de testare (continuare)

- programul trebuie examinat pentru a vedea dacă nu face ce trebuie; de asemenea, trebuie examinat pentru a vedea dacă nu cumva face ceva ce nu trebuie

- pe cât posibil, cazurile de test trebuie salvate și re-executate după efectuarea unor modificari

- probabilitatea ca mai multe erori să existe într-o secţiune a programului este proporţională cu numărul de erori deja descoperite în acea secţiune

- efortul de testare nu trebuie subapreciat

- creativitatea necesară procesului de testare nu trebuie subapreciată

Modelul V

Cerințe utilizator → Testare de acceptanță

validare

verificare

Cerințe sistem → Testare de sistem

Arhitectura sistem → Testare de integrare

Module → Testare unitară

IMPLEMENTARE

+ multe alte tipuri de testare

Testarea unitară (unit testing)

- o unitate (sau un modul) se referă de obicei la un element atomic (clasă sau funcţie), dar poate însemna şi un element de nivel mai înalt: bibliotecă, driver etc.
- testarea unei unităţi se face în izolare
  - pentru simularea apelurilor externe se pot utiliza funcţii externe fictive (engl. “stubs”)

Testarea de integrare (integration testing)

- Testează interacţiunea mai multor unităţi
- Testarea este determinată de arhitectură

Testarea sistemului (system testing)

- testarea sistemului testează aplicaţia ca întreg şi este determinată de scenariile de analiză

- aplicaţia trebuie să execute cu succes toate scenariile pentru a putea fi pusă la dispoziţia clientului

- spre deosebire de testarea internă şi a componentelor, care se face prin program, testarea aplicaţiei se face de obicei cu script-uri care rulează sistemul cu o serie de parametri şi colectează rezultatele

- testarea aplicaţiei trebuie să fie realizată de o echipă independentă de echipa de implementare

- testele se bazează pe specificaţiile sistemului

Testarea de acceptanță (acceptance testing)

- testele de acceptanță determină dacă sunt îndeplinite cerințele unei specificații sau ale contractului cu clientul.
- sunt de diferite tipuri:
  - teste rulate de dezvoltator înainte de a livra produsul software
  - teste rulate de utilizator (*user acceptance testing*)
  - teste de operaționalitate (*operational testing*)
  - testare alfa și beta: alfa la dezvoltator, beta la client cu un grup ales de utilizatori

Testele de regresie (regression testing)

- un test valid generează un set de rezultate verificate, numit “standardul de aur”
- testele de regresie sunt utilizate la **re-testare**, după realizarea unor modificări, pentru a asigura faptul că modificările nu au introdus noi defecte în codul care funcţiona bine anterior
- pe măsură ce dezvoltarea continuă, sunt adăugate alte teste noi, iar testele vechi pot rămâne valide sau nu
- dacă un test vechi nu mai este valid, rezultatele sale sunt modificate în standardul de aur
- acest mecanism previne regresia sistemului într-o stare de eroare anterioară

Testarea performanţei (performance testing)

- O parte din testare se concentrează pe evaluarea proprietăţilor non-funcționale ale sistemului, cum ar fi:
  - siguranța (“reliability”) - menţinerea unui nivel specificat de performanţă
  - securitatea - persoanele neautorizate să nu aibă acces, iar celor autorizate să nu le fie refuzat accesul
  - utilizabilitatea - capacitatea de a fi înţeles, învăţat şi utilizat
  - load & stress testing (v. următoarele două slide-uri)

Testarea la încărcare (load testing)

- asigură faptul că sistemul poate gestiona un volum aşteptat de date, similar cu acela din locaţia-destinaţie (de exemplu la client)
- verifică eficienţa sistemului şi modul în care scalează acesta pentru un **mediu real** de execuţie

Testarea la stres (stress testing)

- solicită sistemul **dincolo de încărcarea maximă** proiectată
- supraîncărcarea testează modul în care „cade” sistemul
  - sistemele nu trebuie să eşueze catastrofal
  - testarea la stres verifică pierderile inacceptabile de date sau funcţionalităţi
- deseori apar aici conflicte între teste. Fiecare test funcţionează corect atunci când este făcut separat. Când două teste sunt rulate în paralel, unul sau ambele teste pot eşua
  - cauza este de obicei managementul incorect al accesului la resurse critice (de exemplu, memoria)
- o altă variantă, “soak testing”, presupune rularea sistemului pentru o perioadă lungă de timp (zile, săptămâni, luni)
  - în acest caz, de exemplu scurgerile nesemnificative de memorie se pot acumula şi pot provoca căderea sistemului

Testarea interfeţei cu utilizatorul (GUI testing)

- majoritatea aplicaţiilor din zilele noastre au interfeţe grafice cu utilizatorul (GUI)
  - testarea interfeţei grafice poate pune anumite probleme
  - cele mai multe interfeţe, dacă nu chiar toate, au bucle de evenimente, care conţin cozi de mesaje de la mouse, tastatură, ferestre, touchscreen etc.
  - asociate cu fiecare eveniment sunt coordonatele ecranului
  - testarea interfeţei cu utilizatorul presupune memorarea tuturor acestor informaţii şi elaborarea unei modalităţi prin care mesajele să fie trimise din nou aplicaţiei, la un moment ulterior
- de obicei se folosesc scripturi pentru testare

Testarea utilizabilităţii (usability testing)

- testează cât de uşor de folosit este sistemul
- se poate face în laboratoare sau „pe teren” cu utilizatori din lumea reală
- exemple de metode folosite:
  - testare “pe hol” (hallway testing): cu câțiva utilizatori aleatori
  - testare de la distanță: analizarea logurilor utilizatorilor (dacă își dau acordul pentru aceasta)
  - recenzii ale unor experți (externi)
  - A/B testing: în special pentru web design, modificarea unui singur element din UI (d.ex. culoarea sau poziția unui buton) și verificarea comportamentului unui grup de utilizatori


testare
de tip
cutie neagră

Testarea de tip “cutie neagră”

- testarea exhaustivă nu este fezabilă
- generarea aleatorie a cazurilor de test nu este eficientă
- posibilă soluție: se iau în considerare numai intrările (într-un modul, componentă sau sistem) şi ieşirile dorite, conform specificaţiilor
  - structura internă este ignorată (de unde și numele de “**black box testing**”)
  - deoarece se bazează pe funcționalitatea descrisă în specificații, se mai numește și **testare funcţională**
  - poate fi folosită în principiu la orice nivel de testare (unitară, integrare, sistem)

Observații

- datele de test sunt generate pe baza specificației (cerințelor) programului, structura programului nejucând nici un rol
- tipul de specificație ideal pentru testarea “cutie neagră” este alcătuit din pre-condiții și post-condiții
- exemple de metode de testare de tip “cutie neagră”:
  - partiționare în clase de echivalență
  - analiza valorilor de frontieră
  - partiționarea în categorii
  - graful cauză-efect
  - testarea tuturor perechilor
  - testarea bazată pe modele
  - etc.

Partiționare în clase de echivalență

- ideea de bază este de a partiționa domeniul problemei (datele de intrare) în ***partiții de echivalență*** sau ***clase de echivalență*** astfel încât, din punctul de vedere al specificației, datele dintr-o clasă să fie tratate în mod identic

- cum toate valorile dintr-o clasă au același comportament, presupunem că toate valorile dintr-o clasă sunt procesate în același fel, fiind deci suficient să se aleagă câte o valoare din fiecare clasă

- domeniul de ieșire este tratat asemănător

Partiționare în clase de echivalență

- clasele de echivalență nu trebuie să se suprapună, deci orice clase care s-ar suprapune trebuie descompuse în clase separate

- după ce clasele au fost identificate, **se alege o valoare din fiecare clasă**. În plus, pot fi alese și date invalide (care sunt în afara claselor și nu sunt procesate de nici o clasă)

- alegerea valorilor din fiecare clasă este arbitrară deoarece se presupune că toate valorile sunt procesate într-un mod identic

Exemplu - căutare simplă într-un șir

Se testează un program care verifică **dacă un caracter se află într-un șir de cel mult 20 de caractere**.

Mai precis, se introduc ***n*** caractere, unde *n* este un întreg între 1 și 20, iar apoi un caracter ***c***, care este căutat printre cele *n* caractere introduse anterior.

Programul va produce o ieșire care indică prima poziție din șir unde a fost găsit caracterul *c* sau un mesaj indicând ca acesta nu a fost găsit.

La final, utilizatorul are opțiunea să caute un alt caracter tastând ***y*** (yes) sau să termine procesul tastând ***n*** (no).

Domeniul de intrări

Există 4 intrări:

- un întreg pozitiv - ***n***
- un șir de caractere - ***x***
- caracterul care se caută - ***c***
- opțiunea de a căuta sau nu un alt caracter - ***s***

Clase de echivalență pentru intrări

- *n* trebuie să fie între 1 și 20, deci se disting 3 clase de echivalență:
  - N_1 = 1..20
  - N_2 = { *n* | *n* < 1 }
  - N_3 = { *n* | *n* > 20 }
- întregul *n* determină lungimea șirului de caractere și nu se precizează nimic despre tratarea diferită a șirurilor de lungime diferită, deci a doua intrare nu determină clase de echivalență suplimentare
- *c* nu determină deocamdată clase de echivalență suplimentare
- opțiunea de a căuta un nou caracter este binară, deci se disting 2 clase de echivalență
  - S_1 = { *y* }
  - S_2 = { *n* }

Ieșiri

Pentru ieșiri, există următoarele 2 posibile variante:

- poziția la care caracterul se găsește în șir, sau
- un mesaj care arată că nu a fost găsit caracterul.

Acestea sunt folosite pentru a împărți **domeniul de intrare** în 2 clase: una pentru cazul în care caracterul se află în șirul de caractere și una pentru cazul în care acesta lipsește:

- C_1(*x*) = { *c* | *c* se află în *x* }
- C_2(*x*) = { *c* | *c* nu se află în *x* }

Clase de echivalență pentru program

Clasele de echivalență pentru întregul program se obțin ca o combinație a claselor individuale:

- E_111 = { (n, x, c, s) | n ∈ N_1, |x| = n, c ∈ C_1(x), s ∈ S_1 }
- E_112 = { (n, x, c, s) | n ∈ N_1, |x| = n, c ∈ C_1(x), s ∈ S_2 }
- E_121 = { (n, x, c, s) | n ∈ N_1, |x| = n, c ∈ C_2(x), s ∈ S_1 }
- E_122 = { (n, x, c, s) | n ∈ N_1, |x| = n, c ∈ C_2(x), s ∈ S_2 }
- E_2 = { (n, x, c, s) | n ∈ N_2 }
- E_3 = { (n, x, c, s) | n ∈ N_3 }

Date de test

Setul de date de test se alcătuiește alegându-se o valoare a intrărilor pentru fiecare clasă de echivalență. De exemplu:

- E_111 : (3, abc, a, y)
- E_112 : (3, abc, a, n)
- E_121 : (3, abc, d, y)
- E_122 : (3, abc, d, n)
- E_2 : (0, _, _, _)
- E_3 : (25, _, _, _)

Intrări și rezultate

Intrări				**Rezultatul care trebuie afișat**
n	***x***	***c***	***s***	
3	abc	a	y	afișează poziția 1; se cere introducerea unui nou caracter
3	abc	a	n	afișează poziția 1
3	abc	d	y	afișează “caracterul nu apare”; se cere introducerea unui nou caracter
3	abc	d	n	afișează “caracterul nu apare”
0				cere introducerea unui întreg între 1 și 20
25				cere introducerea unui întreg între 1 și 20

Avantaje și dezavantaje

Avantaje

- reduce drastic numărul de date de test doar pe baza specificației
- potrivită pentru aplicații de tipul procesării datelor, în care intrarile și ieșirile sunt ușor de identificat și iau valori distincte

Dezavantaje

- modul de definire a claselor nu este evident (nu există nicio modalitate riguroasă sau măcar niște indicații clare pentru identificarea acestora).
- în unele cazuri, deși specificația ar putea sugera că un grup de valori sunt procesate identic, acest lucru nu este adevărat. (Acest lucru întărește ideea ca metodele de tip “cutie neagră” trebuie combinate cu cele de tip “cutie albă”.)
- mai puțin aplicabile pentru situații când intrările și ieșirile sunt simple, dar procesarea este complexă.

Analiza valorilor de frontieră (*boundary value analysis*)

- analiza valorilor de frontieră este o altă metodă de tip “cutie neagră”
- este folosită de obicei împreună cu partiționarea în clase de echivalență
- ea se concentrează pe examinarea valorilor de frontieră ale claselor, care de regulă sunt o sursă importantă de erori
- această metodă adaugă informații suplimentare pentru generarea setului de date de test

Valori de frontieră

Pentru exemplul nostru, odată ce au fost identificate clasele, valorile de frontieră sunt ușor de identificat:

- valorile 0, 1, 20, 21 pentru *n*
- caracterul c poate să se găsească în șirul *x* pe prima sau pe ultima poziție.

Deci se vor testa următoarele valori:

- N_1 : 1, 20
- N_2 : 0
- N_3 : 21
- C_1 : c_11 se află pe prima poziție în *x*, c_12 se află pe ultima poziție în *x*
- pentru restul claselor se ia câte o valoare (arbitrară)

Date de test

Astfel, pentru E_111 si E_112 vom alege câte 3 date de test (*x* de 1 caracter și *x* de 20 de caractere în care *c* se găsește pe poziția 1 și pe poziția 20), iar pentru E_121 și E_122 câte 2 date de test (*x* de 1 caracter și *x* de 20 de caractere).

În total vom avea 12 date de test:

- E_111 : (1, a, a, y), (20, abcdefghijklmnoprstu, a, y), (20, abcdefghijklmnoprstu, u, y)
- E_112 : (1, a, a, n), (20, abcdefghijklmnoprstu, a, n), (20, abcdefghijklmnoprstu, u, n)
- E_121: (1, a, b, y), (20, abcdefghijklmnoprstu, z, y)
- E_122: (1, a, b, n), (20, abcdefghijklmnoprstu, z, n)
- E_2 : (0, _, _, _)
- E_3 : (21, _, _, _)

Intrări (tabelar)

Intrări			
n	***x***	***c***	***s***
1	a	a	y
		a	n
1	a	b	y
		b	n
20	abcdefghijklmnoprstu	a	y
		a	n
20	abcdefghijklmnoprstu	u	y
		u	n
20	abcdefghijklmnoprstu	z	y
		z	n
0			
21			

Partiționarea în categorii (*category-partition*)

- această metodă pentru testare de tip “cutie neagră” se bazează pe cele două anterioare.

- ea încearcă să genereze date de test care "acoperă" funcționalitatea sistemului și astfel, să crească posibilitatea de găsire a erorilor.

- cuprinde următorii 7 pași:

1. descompune specificația funcțională în unități (programe, funcții, etc.) care pot fi testate separat

2. pentru fiecare unitate, identifică parametrii și condițiile de mediu (d. ex. starea sistemului la momentul execuției) de care depinde comportamentul acesteia

Partiționarea în categorii - continuare

… continuare:

3. găsește categoriile (proprietăți sau caracteristici importante) fiecărui parametru sau condițiile de mediu.

4. partiționează fiecare categorie în alternative. O alternativă reprezintă o mulțime de valori similare pentru o categorie.

5. scrie specificația de testare. Aceasta constă din lista categoriilor și lista alternativelor pentru fiecare categorie.

6. creează cazuri de testare prin alegerea unei combinații de alternative din specificația de testare (fiecare categorie contribuie cu zero sau o alternativă).

7. creează date de test alegând o singură valoare pentru fiecare alternativă.

Exemplu - pașii 1-3

- Pentru exemplul nostru:

  1. descompune specificația în unități: avem o singură unitate
  2. identifică parametrii: *n, x, c, s*
  3. găsește categorii:
     - *n* : dacă este în intervalul valid 1..20
     - *x* : dacă este de lungime minimă, maximă sau intermediară
     - *c* : dacă ocupă prima sau ultima poziție sau o poziție în interiorul lui *x* sau nu apare în *x*
     - *s* : dacă este pozitiv (*y*) sau negativ (*n*)

Exemplu - pasul 4

4. Partiționează fiecare categorie în alternative:
   - *n* : <0, 0, 1, 2..19, 20, 21, >21
   - *x* : lungime minimă, maximă sau intermediară
   - *c* : poziția este prima, în interior sau ultima; sau *c* nu apare în *x*
   - *s* : y, n

Exemplu - pasul 5

5. Scrie specificația de testare

- *n*

  1. { *n* | *n* < 0 }
  2. 0
  3. 1 [ok, lungime1]
  4. 2..19 [ok, lungime_medie]
  5. 20 [ok, lungime20]
  6. 21
  7. { *n* | *n* > 21 }

- *x*

  1. { *x* | |*x*| = 1 } [if ok and lungime1]
  2. { *x* | 1 < |*x*| < 20 } [if ok and lungime_medie]
  3. { *x* | |*x*| = 20 } [if ok and lungime20]

Exemplu - pasul 5 - continuat

5. Scrie specificația de testare - continuare

- *c*
  1. { *c* | *c* se află pe prima poziție în *x* } [if ok]
  2. { *c* | *c* se află în interiorul lui *x* } [if ok and not lungime1]
  3. { *c* | *c* se află pe ultima poziție în *x* } [if ok and not lungime1]
  4. { *c* | *c* nu se află în *x* } [if ok]
- *s*
  1. *y* [if ok]
  2. *n* [if ok]

Exemplu - pasul 6 - observație

Din specificația de testare anterioră ar trebui să rezulte
7 x 3 x 4 x 2 = 168 de cazuri de testare.

Pe de altă parte, unele combinații de alternative nu au sens și pot fi eliminate. Acest lucru se poate face adăugând constrângeri acestor alternative.

Constrângerile pot fi: fie proprietăți ale alternativelor, fie condiții de selecție bazate pe aceste proprietăți. În acest caz, alternativele vor fi combinate doar dacă acele condiții de selecție sunt satisfacute.

În exemplul nostru (folosind constrângerile date anterior în paranteze […]), putem reduce numărul cazurilor de testare la 24.

Exemplu - pasul 6

6. Creează cazuri de testare (24 de teste)

n1
n2
n3x1c1s1
n3x1c1s2
n3x1c4s1
n3x1c4s2
n4x2c1s1
n4x2c1s2
n4x2c2s1
n4x2c2s2
n4x2c3s1
n4x2c3s2
n4x2c4s1
n4x2c4s2
n5x3c1s1
n5x3c1s2
n5x3c2s1
n5x3c2s2
n5x3c3s1
n5x3c3s2
n5x3c4s1
n5x3c4s2
n6
n7

Exemplu - pasul 7 (creează date de test)

Intrări			
n	***x***	***c***	***s***
-5			
0			
1	a	a	y
1	a	a	n
1	a	b	y
1	a	b	n
5	abcde	a	y
5	abcde	a	n
5	abcde	c	y
5	abcde	c	n
5	abcde	e	y
5	abcde	e	n
5	abcde	f	y
5	abcde	f	n
20	abcdefghijklmnoprstu	a	y
20	abcdefghijklmnoprstu	a	n
20	abcdefghijklmnoprstu	c	y
20	abcdefghijklmnoprstu	c	n
20	abcdefghijklmnoprstu	u	y
20	abcdefghijklmnoprstu	u	n
20	abcdefghijklmnoprstu	z	y
20	abcdefghijklmnoprstu	z	n
21			
25			

Avantaje și dezavantaje

- pașii de început (identificarea parametrilor și a condițiilor de mediu precum și a categoriilor) nu sunt bine definiți, bazându-se pe experiența celui care face testarea. Pe de alta parte, odată ce acești pași au fost realizați, aplicarea metodei este clară.

- este mai clar definită decât metodele “cutie neagră” anterioare și poate produce date de testare mai cuprinzătoare, care testează funcționalități suplimentare; pe de altă parte, datorită exploziei combinatorice, pot rezulta seturi de teste de foarte mari dimensiuni.