Cuprins

1. Testare funcțională

- Datele de test sunt generate pe baza specificației (cerințelor) programului, structura programului nejucând nici un rol.
- Tipul de specificație ideal pentru testarea funcțională este alcătuit din pre-condiții și post-condiții.
- Majoritatea metodelor funcționale se bazează pe o partiționare a datelor de intrare astfel încât datele aparținând unei aceeași partiții vor avea proprietăți similare (identice) în raport cu comportamentul specificat.

(a) Partiționare de echivalență (equivalence partitioning)

- 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 sunt tratate în mod identic.
- Cum toate valorile dintr-o clasă au specificat același comportament, se poate presupune că toate valorile dintr-o clasă vor fi procesate în același fel, fiind deci suficient să se aleagă câte o valoare din fiecare clasă.
- În plus, domeniul de ieșire va fi tratat în același fel, iar clasele rezultate vor fi transformate în sens invers (reverse engineering) în clase ale domeniului de intrare.
- Clasele de echivalență nu trebuie să se suprapună, deci orice clase care s-ar suprapune trebuie descompuse în clase separate.

- Dupa 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 vor fi procesate într-un mod identic.

Exemplu:

Se testează un program care verifică daca un caracter se află într-un șir de cel mult 20 de caractere. Mai precis, pentru un întreg *n* aflat între 1 și 20, se introduc caractere, iar apoi un caracter *c*, care este apoi căutat printre cele *n* caractere introduse anterior. Programul va produce o ieșire care va indica prima poziție din șir unde a fost găsit caracterul *c* sau un mesaj indicand că acesta nu a fost găsit. Utilizatorul are opțiunea să caute un alt caracter tastând *y* (yes) sau să termine procesul tastând *n* (no).

1. Domeniul de intrări:

Exista 4 intrări:

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

2. Domeniul de ieșiri

Constă din urmatoarele 2 răspunsuri:

- Poziția la care caracterul se găseste în șir
- Un mesaj care arată că nu a fost găsit

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}

Clasele de echivalență pentru întregul program (globale) se pot obține ca o combinatie a claselor individuale:

C_111 = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_1(x), s \in S_1}

C_112 = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_1(x), s \in S_2}

C_121 = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_2(x), s \in S_1}

C_122 = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_2(x), s \in S_2}

C_2 = { (n, x, c, s) | n \in N_2}

C_3 = { (n, x, c, s) | n \in N_3}

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

c_111: (3, abc, a, y)

c_112: (3, abc, a, n)

c_121: (3, abc, d, y)

c_122: (3, abc, d, n)

c_2: (0, _, _, _)

c_3: (25, _, _, _)

6 clase

Intrări				Rezultat afișat (expected)
n	**x**	**c**	**s**	
0				Cere introducerea unui întreg între 1 și 20
25				Cere introducerea unui întreg între 1 și 20
3	abc	a	y	Afișează poziția 1; se cere introducerea unui nou caracter
		a	n	Afișează poziția 1
3	abc	d	y	Caracterul nu apare; se cere introducerea unui nou caracter
		d	n	Caracterul nu apare

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void equivalencePartitioning() {
tester.main(new String[]{"0", null, null, null});
tester.main(new String[]{"25", null, null, null});
tester.main(new String[]{"3", "a", "b", "c", "a", "y"});
tester.main(new String[]{"3", "a", "b", "c", "a", "n"});
tester.main(new String[]{"3", "a", "b", "c", "d", "y"});
tester.main(new String[]{"3", "a", "b", "c", "d", "n"});
}
```

Avantaje:

- Reduce drastic numărul de date de test doar pe baza specificației.
- Potrivită pentru aplicații de tipul procesării datelor, în care intrările și ieșirile sunt ușor de identificat și iau valori distincte.

Dezavantaje:

- Modul de definire al claselor nu este evident (nu există nici o 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 că metodele funcționale trebuie aplicate împreună cu cele structurale.
- Mai puțin aplicabile pentru situații când intrările și ieșirile sunt simple, dar procesarea este complexă.

(b) Analiza valorilor de frontieră (boundary value analysis)

Analiza valorilor de frontieră este folosită de obicei împreuna cu partiționarea de echivalență. Ea se concentrează pe examinarea valorilor de frontieră ale claselor, care de obicei sunt o sursă importanta de erori.

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 gasească 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)

Deci, pentru C_111 și C_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 pozitia 20), iar pentru C_121 și C_122 câte 2 date de test (x de 1 caracter și x de 20 de caractere). În total vom avea 12 date de test.

C_111 : (1, a, a, y), (20, abcdefghijklmnoprstu, a, y), (20, abcdefghijklmnoprstu, u, y)

C_112 : (1, a, a, n), (20, abcdefghijklmnoprstu, a, n),
(20, abcdefghijklmnoprstu, u, n)

C_121: (1, a, b, y), (20, abcdefghijklmnoprstu, z, y)

C_122: (1, a, b, n), (20, abcdefghijklmnoprstu, z, n)

C_2 : (0, _, _, _)

C_3 : (21, _, _, _)

Intrari				Rezultat afișat (expected)
n	**x**	**c**	**s**	
0				
21				
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	

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void boundaryValueAnalysis () {
//…
}
```

Avantaje și dezavantaje:

- Aceleași ca la metoda anterioară.
- În plus, această metodă adaugă informații suplimentare pentru generarea setului de date de test și se concentrează asupra unei arii (frontierele) unde de regulă apar multe erori.

(c) Partiționarea în categorii (category-partitioniong)

Această metodă se bazează pe cele două anterioare. Ea caută să genereze date de test care "acoperă" funcționalitatea sistemului și maximizeaza posibilitatea de găsire a erorilor.

Cuprinde următorii pasi:

1. Descompune specificația funcțională în unitati (programe, functii, etc.) care pot fi testate separat.
2. Pentru fiecare unitate, identifică parametrii și condițiile de mediu (ex. starea sistemului la momentul execuției) de care depinde comportamentul acesteia.
3. Găseste categoriile (proprietăți sau caracteristici importante) fiecărui parametru sau condiții 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 în 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.

Pentru exemplul nostru:

1. *Descompune specificatia 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 sau negativ
4. *Partiționeaza 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

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

Din specificația de testare ar trebui să rezulte 7 * 3 * 4 * 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 sau proprietăți ale alternativelor sau condiții de selectie bazate pe aceste proprietăți. În acest caz, alternativele vor fi combinate doar

dacă condițiile de selecție sunt satisfacute. Folosind acest procedeu, în exemplul nostru vom reduce numărul cazurilor de testare la 24.

6. Creează cazuri de testare

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

24 cazuri de testare

7. Creează date de test

Intrari				**Rezultat afișat (expected)**
n	**x**	**c**	**s**	
-5				
0				
21				
25				
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	

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void categoryPartitioning () {
//…
}
```

Notă: O altă categorie poate fi considerată numarul de apariții a lui c în șirul x. Această categorie poate fi adaugată celor existente.

Avantaje și dezavantaje

- Pașii de inceput (identificarea parametrilor și a condițiilor de mediu precum și a categoriilor) nu sunt bine definiți și se bazează pe experienta celui care face testarea. Pe de altă parte, odata ce acesti pași au fost trecuti, aplicarea metodei este foarte clară.
- Este mai clar definită decat metodele funcționale anterioare și poate produce date de testare mai cuprinzatoare, care testează funcționalitati suplimentare; pe de altă parte, datorită exploziei combinatorice, pot rezulta date de test de foarte mare dimensiune.