Testing: a form of program validation

real data

satisfies specification

Testing method(step-by-step):generate Test Cases -> Develop& Run Tests

* unit testing: each module in isolation
* integration testing: a group of modules
* regression testing: after modifications -> re-run tests

Test Case (TC): combination {input data values} (of given test unit)

* exhaustive testing: *impractical*
* small representative set of Test Cases
* succeed with TCs as well as input

Test stand-alone procedures

* generate Test Cases: black-box white-box
* develop & run Test: Junit

*generate* Test Cases

*Approximation* (input domains)🡨🡪 representative Test Data Sets (TDSs)

* generate TDSs
* Combine TDSs -> form **Test Cases**

TDS: (values) subset of input domain

one input domain may contain several TDSs

* generate TDSs: identify input ranges

define a TDS per range (using representative data values)

**BBT**: black-box testing 🡪 program *specification*

**GBT**: glass-box or white-box testing 🡪 program *text*

easy -> TC generation + result interpretation

implementation (change) 🡨 robust against

specification: flaw/ incomplete/ not observed

* path incomplete

*input range criteria*

@requires: constraint expressions

@effects: condition output 🡺 consider relationships between inputs

exception

notation: semi-informal set

number range

*TDS formation criteria*

each range: data type (array-type 🡨🡪 String data type)

typical + atypical data type

iterator, data abstraction, type hierarchy

|  |  |  |
| --- | --- | --- |
|  | typical data | atypical data |
| numeric data type | few numbers in range | min + max values (of range) |
| array data type | few elements | null, empty  one-element  known values /\ specific indices |

/\*\*   
\* **@effects**

\* if p is a prime  
\* return true  
\* else  
\* return false  
\*/

**public static boolean** isPrime(**int** p)

*BBT*

|  |  |  |
| --- | --- | --- |
|  | *Ranges* | *TDSs* |
| primes | {2, 3, 5, …} | {2, 3, 5, 31, 65537} |
| non-primes | {4, 6, 8, 9, …} | {4, 32, 65538} |

/\*\*

\* **@requires** x >= 0 && .00001 < epsilon < .001

\* **@effects**  
\* return n such that x – epsilon < n^2 <= x + epsilon

\*/

**public static float** sqrt(**float** x, **float** epsilon)

|  |  |  |
| --- | --- | --- |
|  | *Ranges* | *TDSs* |
| x | [0, +∞) | {0, 0.001, 0.01, 0.09, 0.5, 1, 2, 10, 100, 2147483600} |
| epsilon | (.00001, .001) | {.00002, .0001, .0009} |

/\*\*

\* **@effects**  
\* if a is null  
\* throws NullPointerException  
\* else if x is not in a  
\* throws NotFoundException  
\* else  
\* returns i such that a[i] = x  
\*  
\*/

**public static int** search (**int**[] a, **int** x)

|  |  |  |
| --- | --- | --- |
|  | *Ranges* | *TDSs* |
| a | {null}  {[]}  {[x1, …, xn]| xi are integers} | {null, [], [1], [3, 1], [3, 1, 4], [3, 5, 1, 4]} |
| x | {y| y in a}  {y| y not in a} | {1, 2} |

specification: flaw/ incomplete/ not observed

* useful

aid -> code analysis + debug

implementation (change) 🡸 not robust against + require knowledge

*input range criteria*

logic paths: conditional, loop, recursion

conditional expression

number of iterations (loop, recursion)

*TDS formation criteria*

condition: data type

loop/ recursion: number of iteration & termination

* reasonable numbers of iterations:

loop: 0, 1, 2

recursion: (base) 0, 1

(inductive once) 2

(inductive twice) 3

Conditional

**if** P(x)

// do this

**else if** Q(x)

// do that

**else**

// do something else

*GBT*

Ranges:

* all x s.t P(x)
* all x s.t Q(x)
* all x s.t. P(x) /\ Q(x)

deterministic loop

**static int** someMethod(**int** n) {

**for** (**int** i = 1; i<= n; i++)

// do something

}

|  |  |  |
| --- | --- | --- |
|  | *Ranges* | *TDSs* |
| n | (, 0]  [1, ) | {0, 1, 2} |

non-deterministic loop

**static int** someMethod(**int** x) {

**while** (x > 0)

// do something with x

}

|  |  |  |
| --- | --- | --- |
|  | *Ranges* | *TDSs* |
| x | (, 0]  [1, ) | {0, 1, 2} |

recursion

**static int** fact(**int** n) {

**if** (n < 1)

**return** -1;

**else if** (n == 1)

**return** 1;

**else**

**return** n \* fact(n-1);

}

|  |  |  |
| --- | --- | --- |
|  | *Ranges* | *TDSs* |
| n | (, 1)  {1}  (1, ) | {-1, 1, 2, 3} |

conditional

**static int** maxOfThree (**int** x, **int** y, **int** z) {

**if** (x > y) {

**if** (x > z)

**return** x;

**else** // x <= z

**return** z;

}

**else** { // x <= y

**if** (y >= z)

**return** y;

**else** // y < z

**return** z;

}

}

|  |  |  |
| --- | --- | --- |
|  | *Ranges* | *TDSs* |
| x, y, z | x > y, z  y=z, y>z, y<z | { (3, 2, 1), (3, 2, 2), (3, 1, 2) } |
| z x > y  z=x, z>x | { (3, 2, 4), (3, 2, 3) } |
| x, z y   * x < z   🡪 x<z<= y   * x > z   🡪 z<x<=y   * x=z   🡪 x=z=y  x=z < y  y=z, y=x  x=y=z  z=x < y | { (1, 2, 2), (2, 2, 1), (1, 1, 1), (1, 2, 1) } |
| x y < z  x=y  x<y | { (1, 2, 3), (1, 1, 2) } |

condition & loop

**static int** someMethod(**int** x) {

**while** (x > 0) {

// checks x modulo 10

**if** (x % 10 == 5) **break**;

x--;

}

}

|  |  |  |
| --- | --- | --- |
|  | *Ranges* | *TDSs* |
| x | (, 0]  (0, )  {y| y = 10k + 5, k 0 }  {y| y 10k + 5, k 0} | {0, 5, 15, 1, 2} |

**static void** myMethod(**int** n, **int** m) {

**for** (**int** i = 1; i<= n; i++) {

**if** (pre (i \* m))

m++;

}

}

n= 2, i= {1, 2}

choose m to satisfy these: pred(m) && pred(2m + 2)

pred(m) && pred(2m + 2)

pred(m) && pred(2m)

pred(m) && pred(2m)