CS 5154: Software Testing

Implementing Input Space Partitioning

Instructor: Owolabi Legunsen

Fall 2021

Recall: what we do in MDTD

- Move from implementation level to abstraction level
- At the abstraction level, define test requirements and find input values that satisfy them
- Back in the implementation level: write, run, and evaluate tests for the inputs

How do we implement these steps for ISP?

- Step 1: Identify testable functions in your program
- Step 2: Find all input parameters
- Step 3: Model the input domain
- Step 4: Use a criterion to choose combination of values
- Step 5: Refine combinations of blocks into test inputs

The five ISP steps by example

- Consider method `triang()` from class `TriangleType` :
  - http://www.cs.gmu.edu/~offutt/softwaretest/java/Triangle.java
  - http://www.cs.gmu.edu/~offutt/softwaretest/java/TriangleType.java

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid }

/** side1, side2, and side3 are lengths of the sides of a triangle
 *  Returns the appropriate enum value
 **/
public static Triangle triang (int side1, int side2, int side3)
```

Step 1: Identify testable functions in TriangleType

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid }

/** side1, side2, and side3 are lengths of the sides of a triangle
 *  Returns the appropriate enum value
 **/
public static Triangle triang (int side1, int side2, int side3)
```

Identifying testable functions more generally

- Individual methods have one testable function
  - What if the method is private?
  - What if a method calls other methods?
- Each method in a class should be tested individually
- But methods in a class may share characteristics that you can reuse

Step 2: Find input parameters for triang()

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid }


/** side1, side2, and side3 are lengths of the sides of a triangle
 *  Returns the appropriate enum value
 **/
public static Triangle triang (int side1, int side2, int side3)
```

Finding input parameters for testable functions

- Simple step, but be careful to identify all parameters
- Remember to check if program state is an input parameter

```
add(E e) // add element e to Set
```

- Remember to check if data sources are input parameters

```
find(String key) // find location of key in a file
```

Step 3: Model the input domain for triang()

```
public static Triangle triang(int side1, int side2, int side3)
```

- Consider only parameter types or the functionality of triang()?
- How to combine values obtained from IDM of all parameters?
- What is the correct IDM for triang()?

Two approaches to IDM

- **Interface-based:** develop characteristics only from input parameters
  - e.g., triang() takes three ints

- **Functionality-based:** use behavioral view to develop characteristics
  - e.g., triang() returns a Triangle

- Which approach should we use?

Interface-based IDM: Example

```
/** side1, side2, and side3 are lengths of the sides of a triangle
 *  Returns the appropriate enum value
 **/

public static Triangle triang (int side1, int side2, int side3)
```

- Input domain:
- Partitioning characteristic:
    - Block 1:
    - Block 2:
    - Block 3:

Interface-based IDM: Pros and Cons

- ✓ easy to identify characteristics and translate to test cases
- ✓ almost mechanical to follow
- ✗ may not encode all the information that we know
- ✗ can miss tests if functionality depends on combination of values

Functionality-based IDM: Example

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid }

/** side1, side2, and side3 are lengths of the sides of a triangle
 *  Returns the appropriate enum value
 **/
public static Triangle triang (int side1, int side2, int side3)
```

- Input domain:
- Partitioning characteristic:
  - Block 1:
  - Block 2:
  - Block 3:

Functionality-based IDM: Pros and Cons

✓ allows incorporation of semantics or domain knowledge

✓ can be done earlier from requirement specifications

✗ harder to develop characteristics, e.g., large systems, missing specs

✗ can miss tests if functionality depends on combination of values

Poll: which approach should we use

- Interface-based
- Functionality-based
- Both
- None

Questions so far?

?

One question you may have

How does one design characteristics for the input domain?

Hints: designing functionality-based IDM characteristics

- Consider implicit or explicit preconditions

```
int choose() // select a value
```

- Consider implicit or explicit postconditions

```
// withdraw amount from balance
withdraw(double balance, double amount)
```

- Consider relationships among parameters

```
m(Object x, Object y)
```

Hints on designing characteristics (2)

- Consider factors other than parameters (e.g., “global variables”)

```
withdraw(double balance, double amount)
{ Database db = …; … }
```

- Consider using blocks with fewer characteristics
  - many blocks with few characteristics > few blocks with many characteristics

- As much as possible, do not use current code in your design.
  - Use domain knowledge, specification, etc.

In-Class Exercise

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          return true if element is in the list, false otherwise
```

Create two IDMs for findElement () :
1) Interface-based
2) Functionality-based

An interface-based IDM for findElement

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          return true if element is in the list, false otherwise
```

```
Two parameters : list, element
Characteristics :
   list is null (block1 = true, block2 = false)
   list is empty (block1 = true, block2 = false)
```

A functionality-based IDM for findElement

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          return true if element is in the list, false otherwise
```

Functionality-Based Approach

Two parameters : list, element

Characteristics :

- number of occurrences of element in list (0, 1, >1)
- element occurs first in list (true, false)
- element occurs last in list (true, false)

Compare and contrast the two IDMs?

Interface-Based IDM

Two parameters : list, element

Characteristics :

list is null (block1 = true, block2 = false)

list is empty (block1 = true, block2 = false)

Functionality-Based IDM

Two parameters : list, element

Characteristics :

number of occurrences of element in list (0, 1, >1)

element occurs first in list (true, false)

element occurs last in list (true, false)

Other questions you may be asking

How to create blocks from partitions?

How to select representative values from each block?

A checklist on choosing blocks and values

1. Does each partition allow all valid and invalid values? (completeness)
2. Can you further partition blocks to exercise different functionality?
3. Did you consider boundary values?
4. Does union of blocks in a characteristic cover the input space?
5. Does a value belong to more than one block for a characteristic?

Questions so far?

?

Characteristics can be refined to get more tests

- triang() characteristic: relation of each side to 0

Characteristic	b_1	b_2	b_3
q_1 = "Relation of Side 1 to 0"	positive	equal to 0	negative
q_2 = "Relation of Side 2 to 0"	positive	equal to 0	negative
q_3 = "Relation of Side 3 to 0"	positive	equal to 0	negative

- Max no. of tests: 3*3*3 = 27 (some are valid triangles, others are not)
- How can we refine this characteristic to obtain more tests?

A refinement that yields more tests

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Refinement of q_1"	greater than 1	equal to 1	equal to 0	negative
q_2 = "Refinement of q_2"	greater than 1	equal to 1	equal to 0	negative
q_3 = "Refinement of q_3"	greater than 1	equal to 1	equal to 0	negative

- Max. no. of tests is now: 4*4*4 = 64

Refinement should still yield correct partitioning!

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Refinement of q_1"	greater than 1	equal to 1	equal to 0	negative
q_2 = "Refinement of q_2"	greater than 1	equal to 1	equal to 0	negative
q_3 = "Refinement of q_3"	greater than 1	equal to 1	equal to 0	negative

- Assume that triangle sides were floating point numbers.
- Do you see a problem with this partitioning?
- Problem: No values between 0 and 1 will be chosen! (incomplete)

Choosing values after refinement

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Refinement of q_1"	greater than 1	equal to 1	equal to 0	negative
q_2 = "Refinement of q_2"	greater than 1	equal to 1	equal to 0	negative
q_3 = "Refinement of q_3"	greater than 1	equal to 1	equal to 0	negative

	Values for partition q_1			
Characteristic	b_1	b_2	b_3	b_4
Side1	**2**	1	0	**-1**

Test boundary conditions

Be careful with functionality-based IDM too!

A Geometric Characterization of *triang*()'s Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles	equilateral	invalid

What's wrong with this partitioning?

- Equilateral is also isosceles !
- We need to refine the example to make characteristics valid

Correct Geometric Characterization of *triang*()'s Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid

Choosing values for functionality-based IDM

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid

Possible values for geometric partition q_1

Characteristic	b_1	b_2	b_3	b_4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

Recall: IDM is a design activity!

<u>A Geometric</u> Characterization of *triang*()'s Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles	equilateral	invalid

Can you think of an alternative way to refine this partition?

An alternative refinement

- Break the geometric characterization into four characteristics

Characteristic	b_1	b_2
q_1 = “Scalene”	True	False
q_2 = “Isosceles”	True	False
q_3 = “Equilateral”	True	False
q_4 = “Valid”	True	False

- Then, impose constraints:
  - Equilateral = True implies Isosceles = True
  - Valid = False implies Scalene = Isosceles = Equilateral = False

One last question to answer on IDM

How to consider multiple partitions simultaneously?

What combination of blocks should we choose values from?