CS 5154: Software Testing

Implementing
Input Space Partitioning

Owolabi Legunsen
First, a review of some concepts from last class
Partitioning the input domain into blocks

• Decide on characteristics of your input domain to partition on

• Assumption: values in each block are equally useful for testing

• Example:

  Program: `void foo(String char) // “char” is a letter`
  Input domain: Alphabetic letters
  Partitioning characteristic: Case of letter
  • Block 1: upper case
  • Block 2: lower case
How to know that partitioning is “correct”?

• *Let the input domain be* $D$

• *Characteristic* $q$ *partitions* $D$ *into set of blocks,* $B_q = \{b_1, b_2, \ldots, b_Q\}$

• Each partition must satisfy two properties:
  1. Blocks must be *pairwise disjoint* (no overlap)
    
    
    
    $b_i \cap b_j = \emptyset, \forall i \neq j, b_i, b_j \in B_q$

    
    
    
    2. Together the blocks must *cover* the domain $D$ (complete)

    
    
    
    $\bigcup b = D, \forall b \in B_q$
Partitioning is simple but easy to do wrong

- Consider the characteristic “order of elements in list F”

\begin{align*}
  b_1 &= \text{sorted in ascending order} \\
  b_2 &= \text{sorted in descending order} \\
  b_3 &= \text{arbitrary order}
\end{align*}

but ... something’s fishy ...

What if the list is of length 0 or 1? 

The list \textbf{blocks} 

That is, disjointness is not satisfied

One solution: 

Two characteristics that each address 

\textit{Can you think of a solution?}

\begin{align*}
  C1: \text{List } F \text{ sorted ascending} \\
  &- c1.b1 = \text{true} \\
  &- c1.b2 = \text{false} \\
  C2: \text{List } F \text{ sorted descending} \\
  &- c2.b1 = \text{true} \\
  &- c2.b2 = \text{false}
\end{align*}
But, how does one implement ISP in practice?
Recall: steps 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 to 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/TriangleType.java](http://www.cs.gmu.edu/~offutt/softwaretest/java/TriangleType.java)

```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

```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)
```
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()

```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)
```
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

  findInFile(String key) // find key in a file
Step 3: Model the input domain for triang()

```java
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

```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)

• 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

✗ harder to generate tests; characteristics don’t map to one parameter
Poll: which approach should we use

• Interface-based

• Functionality-based

• Both

• None
Questions so far?
We started a systematic way of doing 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
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
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 for list :
list is null (block1 = true, block2 = false)
list is empty (block1 = true, block2 = false)

Characteristics for element :
element is null (block1 = true, block2 = false)
A functionality-based IDM for findElement

```java
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 for list:**
- 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)
One question that you may have

How does one design characteristics for the input domain?
Hints: designing functionality-based IDM characteristics

• Consider implicit or explicit preconditions

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

• Consider implicit or explicit postconditions

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

• Consider relationships among parameters

```java
m(Object x, Object y)
```
Hints on designing characteristics (2)

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

```java
Database db = ...;
withdraw(double balance, double amount)
{ ... // persist result to db }
```

• Characteristics that yield fewer blocks tend to be complete & disjoint
  • many characteristics with few blocks > few characteristics with many blocks

• As much as possible, do not use current code in your design.
  • Use domain knowledge, specification, etc.
Other questions that 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

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$ = “Relation of Side 1 to 0”</td>
<td>positive</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
<tr>
<td>$q_2$ = “Relation of Side 2 to 0”</td>
<td>positive</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
<tr>
<td>$q_3$ = “Relation of Side 3 to 0”</td>
<td>positive</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
</tbody>
</table>

- Max no. of tests: $3 \times 3 \times 3 = 27$ (some are valid triangles, others are not)
- How can we refine this characteristic to obtain more tests?
A refinement that yields more tests

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1 = \text{&quot;Refinement of } q_1\text{&quot;}$</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
<tr>
<td>$q_2 = \text{&quot;Refinement of } q_2\text{&quot;}$</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
<tr>
<td>$q_3 = \text{&quot;Refinement of } q_3\text{&quot;}$</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
</tbody>
</table>

- Max. no. of tests is now: $4 \times 4 \times 4 = 64$
Refinement should still yield correct partitioning!

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1 = \text{“Refinement of } q_1\text{”}$</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
<tr>
<td>$q_2 = \text{“Refinement of } q_2\text{”}$</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
<tr>
<td>$q_3 = \text{“Refinement of } q_3\text{”}$</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
</tbody>
</table>

- Suppose 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

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$ = “Refinement of $q_1$”</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
<tr>
<td>$q_2$ = “Refinement of $q_2$”</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
<tr>
<td>$q_3$ = “Refinement of $q_3$”</td>
<td>greater than 1</td>
<td>equal to 1</td>
<td>equal to 0</td>
<td>negative</td>
</tr>
</tbody>
</table>

Values for partition $q_1$

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Side1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

Test boundary conditions!
Be careful with functionality-based IDM too!

A Geometric Characterization of `triang()`’s Inputs

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$ = “Geometric Classification”</td>
<td>scalene</td>
<td>isosceles</td>
<td>equilateral</td>
<td>invalid</td>
</tr>
</tbody>
</table>

What’s wrong with this partitioning?

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

Corrected Geometric Characterization of `triang()`’s Inputs

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>$B_1$</th>
<th>$B_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$ = “Geometric Classification”</td>
<td>Scalene</td>
<td>isosceles, not equilateral</td>
<td>equilateral</td>
<td>invalid</td>
</tr>
</tbody>
</table>
Choosing values for functionality-based IDM

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b₁</th>
<th>b₂</th>
<th>b₃</th>
<th>b₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₁ = “Geometric Classification”</td>
<td>scalene</td>
<td>isosceles, not equilateral</td>
<td>equilateral</td>
<td>invalid</td>
</tr>
</tbody>
</table>

Possible values for geometric partition q₁

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>b₁</th>
<th>b₂</th>
<th>b₃</th>
<th>b₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle</td>
<td>(4, 5, 6)</td>
<td>(3, 3, 4)</td>
<td>(3, 3, 3)</td>
<td>(3, 4, 8)</td>
</tr>
</tbody>
</table>
Recall: IDM is a design activity

A Geometric Characterization of \textit{triang}()'s Inputs

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
<th>$b_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1 = \text{“Geometric Classification”}$</td>
<td>scalene</td>
<td>isosceles</td>
<td>equilateral</td>
<td>invalid</td>
</tr>
</tbody>
</table>

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

• Break the geometric characterization into four characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>( b_1 )</th>
<th>( b_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_1 = \text{“Scalene”} )</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>( q_2 = \text{“Isosceles”} )</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>( q_3 = \text{“Equilateral”} )</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>( q_4 = \text{“Valid”} )</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

• Then, impose constraint:
  • \text{Equilaterial = True} implies \text{Isosceles = True}
One last question to answer on IDM

How to consider multiple partitions simultaneously?

What combination of blocks should we choose values from?
We started a systematic way of doing 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
Next: finish a systematic way of doing 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