Software Testing Lecture 6
Input Domain Modelling

Justin Pearson

2019

Input Domains

- The *input domain* of a program is all the possible inputs to that program.
- Even for small programs the domain is infinite.
- Testing is fundamentally about *choosing finite sets* of values from the input domain.

Input Domains

- Input parameters define the scope of the input domain:
  - Parameters to a method.
  - Data read from a file.
  - Global variables.
  - user level inputs
- Domain for each parameter is partitioned into regions.
- At least one value is chosen from each region.

Don't forget the global state can be part of your test case.

Partitions Intuition

- Elements in each partition do the same thing; for some value of 'the same thing'.
- Simple mathematical theory of partitions, allows you to work out if you are on the right track.

Benefits of Inputs Space Partitions

- Can be applied at several levels of testing:
  - Unit
  - Integration
  - System
- Relatively easy to apply with no automation.
- Easy to adjust to get more of fewer tests.
- It is possible to do black box testing, you just need to know what the input space is.

Partitions or Equivalence relations

- We want to formalise the notion of being the same.
- Given a domain. D, there are two ways of looking at partitions:
  - A partition of D is a set of sets $B_1, \ldots, B_q$ that satisfy two properties:
    - Sets must be pairwise disjoint, all $i \neq j$ $B_i \cap B_j = \emptyset$.
    - Sets must cover the whole of D, that is $B_1 \cup \cdots \cup B_q = D$.
  - A binary relation $\equiv_B$ on D such that:
    - for all $d \in D$, $d \equiv_B d$.
    - for all $d, e \in D$, $d \equiv_B e \Leftrightarrow e \equiv_B d$.
    - For all d, e, f $d \equiv_B e \wedge e \equiv_B f$ implies $d \equiv_B f$.

Partitions or Equivalence relations

- Given a partition $B_1, \ldots, B_q$ of D then define:

$$d \equiv_B e \Leftrightarrow \exists i. d \in B_i \wedge e \in B_i$$

- Given an equivalence relation $\equiv_B$ on D then the partition consists of the sets:

$$[d]_{\equiv_B} = \{e \mid d \equiv_B e\}$$

Why is this important?

- Remember, *blocks* should:
  - be disjoint, no overlapping test cases
  - cover the domain, every element is tested.

Using Partitions – Assumptions

- Choose a value from each partition.
- Each value is assumed to be *equally useful* for testing.

It is up to the tester to work out what it means to be equally useful.

Partition design and test design

- There is no mathematical theorem that says that values in an equivalence class act the same way.
- When you design a partition as a test designer you are deciding want to treat as equal.
- Partitions also allow you mathematically define classes of tests that you believe all do the same amount of work.

Consider

```
if (x > 0 ) { .... } else { .... }
```

One of the most important thing about test design is documenting what you expect tests to do. Giving a partition is one way of saying that you the tester believe that values in equivalence classes behave the same.

How do we specify partitions?

Consider a function

```
int f(int x, int y)
```

- The input domain is the Cartesian product of the domains of all the input parameters.
- In theory you would do the partition on this domain.

$D = \{-2147483647 \ldots ...2147483648\} \times \{-2147483647 \ldots ...2147483648$

Sometimes you want to specify partitions on these sorts of domains, but it is often too complex.

Characteristics

Characteristics give a simple way specifying partitions.

- Application to testing:
  - Find *characteristics* in the inputs: parameters, semantic descriptions.
  - Partition each characteristic.
  - Choose tests by combining values from each characteristic.

Characteristics define blocks.

- Example Characteristics
  - Input X is null
  - Order of an input file (sorted, backwards, arbitary
  - Minimum separation of two aircrafts.
  - Input device type (DVD, CD, ...)

Choosing Partitions

- Choosing (or defining) partitions seems to easy, but it is equally easy to get it wrong.
- Consider the property the *order of the file*.
- Define the partition as follows:
  - $b_1 =$ files sorted in ascending order.
  - $b_2 =$ files sorted in descending order.
  - $b_3 =$ files in an arbitrary order.

Something is wrong with this. What about files of length 1.

Choosing Partitions

- Choosing (or defining) partitions seems to easy, but it is equally easy to get it wrong.
- Consider the property the order of the file.
- Define the partition as follows:
  - $b_1 =$ files sorted in ascending order.
  - $b_2 =$ files sorted in descending order.
  - $b_3 =$ files in an arbitrary order.

The blocks are not disjoint. Files of length 1 belong to b_1,b_2 and b_3. Hence for $i \neq j$ $b_i \cap b_j \neq \emptyset$.

Check your Partitions

- It is important to double check that you do have a partition.
- If you do not have a partition then you have not though carefully enough about your test case design.

Choosing Partitions

- You could just be careful.
- Or make your partitions binary and combine them. So, we have two separate partitions. Generate tests from all possible combinations of characteristics.
  - File is sorted ascending:
    - $b_1 =$ true
    - $b_2 =$ false.
  - File is sorted descending:
    - $b_1 =$ true
    - $b_2 =$ false.
- Here the combination is easy.

Properties of Partitions

- Don't forget that partitions should be complete and disjoint; if they are not then they have not been considered carefully enough.
- Review your partitions like any design attempt
- Different alternatives should be considered.

Modelling the input domain: Five stage plan

- **Step 1:** Identify testable functions
  - Individual methods
  - Functions
  - More complicated characteristics, hardware/software, etc. You might have to look beyond your code.
- **Step 2:** Find all parameters
  - Not rocket science. But be complete.
  - Don't forget global variables.
  - Don't forget the objects have states. Doing things to an empty list is different to doing things to a non-empty list.
  - Files, databases etc.

Modelling the input domain: Five stage plan

- **Step 3:** Model the input domain.
  - The domain depends on the parameters.
  - Define structure in terms of characteristics.
  - Get blocks and values.
  - This is where some creativity comes into play, you have to decide which characteristics are meaningful to you.

Modelling the input domain: Five stage plan

- **Step 4:** *Test criteria* for choosing combinations of values.
  - See later in the lecture, but you need to combine your characteristics.
  - Choosing all combinations is often infeasible.
- **Step 5:** Refine combinations of blocks into test inputs.
  - Choose appropriate values from each block.

Two Approaches to Input Domain Modelling

1. **Interface-based** approach.
   - Develop characteristics directly from individual input parameters.
   - Just use properties of the domains without considering the actual function under test.
2. **Functionality-based** approach
   - Develop characteristics from a behavioural view of the program under test.
   - Use your understanding of what the system should do.

Interface-Based vs Functionality-Based Approach Example

```
public boolean findElement(List list, Object element)
//if list or element is null throw NullPointerException
// else return true if element is in the list.
```

- Interface-Based Approach
  - Two parameters: `list`, `element`
  - Characteristics:
    - `list` is null (b_1 = true, b_2 = false)
    - `list` is empty (b_1 = true, b_2 = false)

Interface-Based vs Functionality-Based Approach Example

```
public boolean findElement(List list, Object element)
//if list or element is null throw NullPointerException
// else return true if element is in the list.
```

- Functionality-Based Approach
  - Two parameters: `list`, `element`
  - Characteristics:
    - number of occurrences of `element` in `list`

      $$(0, 1, > 1)$$

    - `element` occurs first in `list`

      $$(\text{true}, \text{false})$$

Sources for functionality-based characteristics

- The specification, or intended behaviour.
- Pre-conditions: tell what ranges and values you expect for input variables.
- Post conditions: often give you relationships between input and output parameters.

Step 3: Modelling the Input Domain

- Partitioning characteristics into blocks and values can be creative.
- More blocks means more tests.
- The partitioning often flows directly from the definition of the *characteristics*.
- Strategies for identifying values:
  - Include *valid*, *invalid* and *special* values.
  - Explore *boundaries* of domains.
  - Include values that represent normal use.
  - Don't forget that partitions must be *complete* **and** *disjoint*

Special Values

Some programming languages such as Java have special values such as `null`. Use these in your partitions.

TriType example

As shown in the book.

```
int Tritype(double i, double j, double k)
```

Should return the type of triangle with side length i, j and k.

- 3 = scalene (All sides different length)
- 2 = equilateral triangle
- 1 = isoceles triangle
- 0 = Not a triangle.

It is quite hard to get it right.

TriType Interface-Based Approach

- One testable function, and three integer inputs.

Characteristic	b_1	b_2	b_3
$q_1 =$ Relation of i to 0	> 0	$= 0$	< 0
$q_2 =$ Relation of j to 0	> 0	$= 0$	< 0
$q_3 =$ Relation of k to 0	> 0	$= 0$	< 0

- Pick one test from each square, a maximum of $3 * 3 * 3 = 27$ tests.
- You have to work out which are valid and which are invalid.

TriType Functional Based Appraoch

- We are classifying triangles.
- Geometric approach.

Characteristic	b_1	b_2	b_3	b_4
Geometric	scalene	isosceles	equilateral	invalid

- Do we have a partition?
- No, an equilateral triangle is also an isosceles triangle.

TriType Functional Based Appraoch

Characteristic Geometric	b_1 scalene	b_2 isosceles	b_3 equilateral	b_4 invalid

- Do we have a partition?
- No, an equilateral is also isosceles.
- So refine it to:

Characteristic Geometric	b_1 scalene	b_2 isosceles, not equilateral	b_3 equilateral	b_4 invalid

Example values $(4, 5, 6) \in b_1$, $(3, 3, 4) \in b_2$, $(3, 3, 3) \in b_3$, $(3, 4, 8) \in b_4$.

Choosing Combinations of Values

- We play the usual game.
- Lots of different characteristics, how do we combine them?
- Most obvious:
  - *All Combinations (ACoC)*: All combinations of blocks from all characteristics must be used.
- Number of tests equals the product of the number of blocks in each characteristic.

Choosing Combinations of Values

- Usual story, too many test cases!
- We should at least choose a value from each block at least once.
  - *Each Choice (EC)*: Each value from each block, for each characteristic, must be used in at least one test case.
- Yields few tests and is cheap, but maybe does not provide such good coverage.

Pair-Wise

- *Pair-Wise (PW)*: A value from each block, for each characteristic, must be combined with a value from every other block, for each other characteristic.
- Given three partitions $[A, B]$,$[q_1, q_2, q_3]$ and $[x, y]$. We would get the requirements:

Requirements:

(A, q_1)	(B, q_1)	(x, q_1)
(A, q_2)	(B, q_2)	(y, q_1)
(A, q_3)	(B, q_3)	(x, q_2)
(A, x)	(B, x)	(y, q_2)
(A, y)	(B, y)	(x, q_3)
		(y, q_3)

Combinations:

(A, q_1, x)	(B, q_1, y)
(A, q_2, x)	(B, q_2, y)
(A, q_3, x)	(B, q_3, y)
$(A, , y)$	$(B, , x)$

Constraints Among Characteristics

Back to the triangle type.

- Some combinations are infeasible. For example:
  - "Less than zero" and "scalene" are not possible at the same time.
- So when you combine tests from different blocks you have to be careful.
- As you might guess there are lots of possible ways of combining things.
- For what you are doing, think hard about your partitions and combinations. Generate quality test cases.

Input Space Partitioning Summary

- Fairly easy to apply
- By refining or merging partitions you can add or remove tests.
- Various strategies for combining characteristics.
- Based only on the input space of the program, not on the implementation.
- Even if are just looking at the data types and do things like "NULL", "Empty", ... you are using partitions.
- Designing good partitions can be a good way of generating good quality test cases.