**Software Testing, Quality Assurance and Maintenance** Winter 2010

Lecture 30 — March 19, 2010

*Patrick Lam* *version 1*

## Functionality-based Input Domain Modelling

Use the intended functionality of system to create tests.

**containsElement example.** Consider again the method `containsElement`, which takes a list `l` and an element `e`. It behaves as follows. If `l` or `e` is null, throw a `NullPointerException`. Otherwise, return `true` if `l` contains `e` and `false` otherwise.

Some possible characteristics:

Notes:

- (+) might yield better test cases due to domain knowledge;
- (+) can create these models from specifications; don't need implementations;
- (-) may be hard to identify values and characeristics;
- (-) harder to generate tests from such an IDM.

## Identifying Characteristics

Here are some possible sources of characteristics for functionality-based IDMs:

- preconditions and postconditions. e.g. 1) `Object.wait()` precondition: must hold lock; characteristic: lock held or not (the state includes the locks currently held). 2) `TriTyp` triangle classification; we might construct characteristics based on the return value.
- relationships between variables, e.g. parameter aliasing;
- specifications contain more concentrated domain knowledge, so it is easier to extract characteristics from them, if they are available.

Using fewer blocks usually gives a more tractable partition, and it is easier to ensure that the partition is disjoint and complete.

## Choosing Blocks and Values

This part is the most creative in input space testing. Here are some tips for finding good values to test with.

Be sure to include:

- valid values and invalid values;
- boundary values and non-boundary values;
- special values (0, `null`, empty set);

and choose your blocks and partitions appropriately; if you need more partitions, you can divide existing partitions, especially those containing valid values. Check that partitions are disjoint and complete.

**Trityp Example.** We discuss the standard triangle classification example again, which takes three side lengths as input.

Interface-based try 1:

| | | | |
|---|---|---|---|
| side 1 is: | $> 0$ | $< 0$ | $= 0$ |
| side 2 is: | $> 0$ | $< 0$ | $= 0$ |
| side 3 is: | $> 0$ | $< 0$ | $= 0$ |

We might refine some partitions, assuming that we take integers as parameters:

| | | | | |
|---|---|---|---|---|
| side 1 is: | $> 1$ | 1 | 0 | $< 0$ |
| side 2 is: | $> 1$ | 1 | 0 | $< 0$ |
| side 3 is: | $> 1$ | 1 | 0 | $< 0$ |

Generating tests for these partitions is straightforward.

Functionality-based try (wrong):

scalene isoceles equilateral invalid

Unfortunately, equilateral triangles are also isoceles, so we should instead use:

scalene isoceles, not equilateral equilateral invalid

Observe that it's not completely straightforward to generate tests from functionality-based partitions. We can, however, generate the following tests:

$(4, 5, 6)$ $(3, 3, 4)$ $(3, 3, 3)$ $(3, 4, 8)$

**Alternate approach.** Instead of multiple blocks (like 4 in the above example), use T/F characteristics (e.g. isEquilateral, etc). Then disjointness and completeness are guaranteed, but you get more characteristics.

**Multiple IDMs.** We can use one IDM for valid values and a second IDM for invalid values, then differentially refine or cover these IDMs.

**Verifying IDMs.** Some tips:

- "Are we missing anything we could include?"
- check completeness and disjointness; if using multiple IDMs, only the sum of the IDMs needs to be complete.