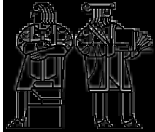


# 6.170 Lecture 8 Black-box testing



John Chapin  
MIT EECS  
2-15-2000

6.170

## Outline

### testing theory: partitioning techniques

- execution equivalence
- revealing subdomains
- heuristics

### black box testing

© John Chapin/John Guttag

2

6.170

## Part 1. Testing Theory

### validate: to increase confidence in program's correctness

correctness: program satisfies its specification

### 2 ways to validate:

verify: use proof techniques  
test: execute program for purpose of detecting errors

### Test $\neq$ debug

**debug: identify the mistake that causes an error**

test: compare input-output pairs to specification  
debug: study the steps that lead to a given error

© John Chapin/John Guttag

3

6.170

## What is hard about testing?

"just try it and see if it works..."

```
int procl(int x, int y, int z)
// requires: 1 <= x,y,z <= 1000
// effects:  computes some f(x,y,z)
```

exhaustive testing would require 1 billion runs

### Key problem: choosing the test suite

small enough to finish quickly (time == \$\$)  
large enough to validate the program

© John Chapin/John Guttag

4

6.170

## Approach: partition the input space

### Input space very large, program small

==> behavior is the same for sets of inputs

**Ideal test suite: try one input from each set**

### Two problems

1. Notion of the same behavior is subtle
  - Naive approach: execution equivalence
  - Better approach: revealing subdomains
2. Discovering the sets requires perfect knowledge
  - Use heuristics to approximate cheaply

© John Chapin/John Guttag

5

6.170

## Naive approach: execution equivalence

```
int abs(int x) {
  if (x < 0) return -x;
  else      return x;
}
```

all  $x < 0$  are execution equivalent:

program takes same sequence of steps for any  $x < 0$

all  $x \geq 0$  are execution equivalent

Suggests a test suite  $\{-3, 3\}$

### But...

1. Too many partitions
2. Doesn't work anyways

© John Chapin/John Guttag

6



### Why execution equivalence doesn't work

Consider the following bug:

```
int abs(int x) {
  if (x < -2) return -x;
  else      return x;
}
```

**{-3, 3} does not reveal the error!**

Two executions:

```
x < -2      x >= -2
```

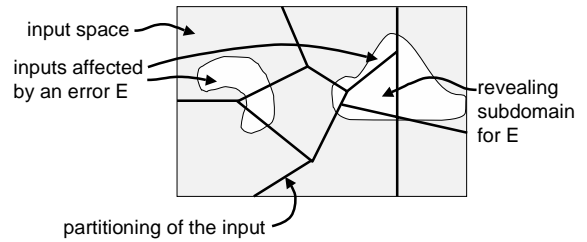
Three behaviors:

```
x < -2 (OK)  x = -2 or -1 (bad)  x >= 0 (OK)
```



### Better approach: revealing subdomains

A subdomain S is revealing for an error E when  
 E present and E affects any input in S →  
 All inputs in S produce incorrect output



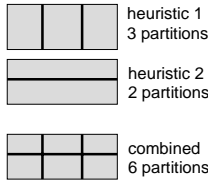
### Heuristics for designing test suites

A good heuristic gives:

- few partitions
- $\forall$  errors e in some class of errors E,
- high probability that some partition is revealing for e

**Different heuristics target different classes of errors**

In practice, combine multiple heuristics



### Part 2: Black box testing

**Heuristic: Partition using the specification**

Procedure or ADT is a black box, internals hidden  
 Liskov: "explore alternate paths through the specification"

**Example**

```
int max(int a, int b)
  // effects:  a > b => returns a
  //           a < b => returns b
  //           a = b => returns a
```

3 partitions, so 3 test cases:

```
(4, 3) => 4 (i.e. any input in the subdomain a > b)
(3, 4) => 4
(3, 3) => 3
```



### Your turn

Write test cases based on paths through the specification

```
int find(int[] a, int value) throws Missing
  // effects: returns the smallest i such
  //           that a[i] == value, unless
  //           value not in a => Missing
```

**2 obvious tests:**

```
( [4, 5, 6], 5 ) => 1
( [4, 5, 6], 7 ) => throw Missing
```

**1 more subtle test**

```
( [4, 5, 5], 5 ) => 1
```

**Must hunt for multiple cases in effects or requires**



### Heuristic: boundary testing

**Create partitions at the edges of other partitions**

**Why do this?**

- off-by-one bugs
- forget to handle empty container
- overflow errors in arithmetic
- aliasing of objects

**Small partitions at the edges of the "main" partitions have a high probability of revealing these errors**

