Testing

- Execute program on sample input data
  - Check if output correct

- Goals
  - Increase confidence program works correctly
    - Acceptance Testing
  - Find bugs in program
    - Debug Testing

Example (Black Box)

```
java TestServlet HelloWorld /FooBar/Test > out
HTTP/1.0 200
Content-Type: text/plain
Hello /FooBar/Test
diff out expectedOutput
```
Limitations of Testing

- Program runs on (very small) subset of input data
  - Exhaustive testing usually impossible
    - Too large input space (possibly infinite)

- Many situations hard to test
  - Multithreaded code
  - Hard-to-reach states

- Testing cannot prove absence of bugs
  - Especially a problem in security

Black Box Testing

- Pick subcomponent of program
  - Internals of component not considered
- Give it inputs
- Compare against correct outputs

The Test Case Generation Problem

- What tests will show that my program works?
  - Must consider “operational scenarios”
  - What is legitimate input?
  - What is the correct action or output?

- How can I make sure that all of the important behaviors of my program have been tested?
  - Usually, you can’t!

Test Cases via Specifications

```java
// Return true if x in a, else returns false
boolean contains(int[] a, int x);
```

- Two “paths” in specification
  - Test case where x is in a
  - Test case where x is not in a
Test Cases via Boundary Conditions

interface List {
    ...  
    inserts the specified element at the specified position in this list (optional operation). Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

    public void add(int index, Object element)
}

  • Test with empty list
  • Test with index at first/last element
  • Others?

Test Cases via Common Mistakes

// Appends l2 to the end of l1
void append(List l1, List l2);

  • Does append work if l1==l2?

  class A {
      ...
  ...
  }

      • Does equals work if operand is an object?

White/Glass Box Testing

  • Pick subcomponent of program
  • Give it inputs
    — Based on component code
  • Compare against correct outputs

Statement Coverage

int select(int[] a, int n, int x)
{
    int i=0;
    while (i<n && a[i] < x)
    {
        if (a[i]<0)
            a[i] = - a[i];
        i++;
    }
    return 1;
}

One test case (n=1, a[0]=-7, x=9) covers all statements
Faults handling positive values of a[i] not revealed
Must add test case (n=1, a[0]=7, x=9) to cover false branch of if
Faults handling positive values of a[i] revealed.
Faults exiting the loop with a[i] < x not revealed

Both i<n and a[i]<x must be false and true for different tests.
Must add tests that cause loop to exit for a value greater than x.
Faults that arise after several loop iterations not revealed.

• Adequacy criteria
  – If significant parts of program structure are not tested, testing is surely inadequate
• Control flow coverage criteria
  – Statement (node, basic block) coverage
  – Branch (edge) coverage
  – Condition coverage
• Attempted compromise between the impossible and the inadequate

• Unit testing
  – Individual components of a program are tested
    • Methods
    • Classes/packages
    • Processes of a distributed system
• Integration testing
  – Test case inputs to whole program, and outputs examined
White vs. Black box

- Black box
  - depends on spec
  - scales up
    - different techniques at different granularity levels
  - cannot reveal code coverage problems
    - same specification implemented with different modules

- White box
  - depends on control or data flow coverage
  - does not scale up
    - mostly applicable at unit and integration testing level
  - cannot reveal missing path errors
    - part of the specification that is not implemented

Testing Activities

- Test case execution is only a part of the process
- Must also consider
  - Test case generation
  - Test result evaluation
- Planning is essential
  - To achieve early and continuous visibility
  - To choose appropriate techniques at each stage
  - To build a testable product
  - To coordinate complementary analysis and testing

Alternatives to Testing

- Assume program works
  - You’re almost certainly wrong

- Formally prove program works
  - Tedious, really hard to get right

- Prove that your program works in a limited sense
  - E.g., type checking

You can’t always get what you want

- Correctness properties are undecidable
  - the halting problem can be embedded in almost every property of interest

![Decision Procedure Diagram]

- Property
- Program
- Decision Procedure
- Pass/Fail
The Halting Problem

- Haltp = true if and only if program P halts
- Halt() does not exist
  - Informal proof:
    - Suppose Halt() exists
    - Consider the following method
      ```java
      void foo() { while (halt("foo()")); }
      ```
    - Then foo() halts if Halt("foo()") false iff foo() does not halt
      - Contradiction, so Halt() does not exist

Examples of Undecidable Properties

- Does P terminate (halt)?
- Does x=0?
- Is f() ever called?
- Which foo method is invoked by b.foo()?
- and so on...

How Can We Check Anything?

- Halting problem == languages are too expressive
  - So reduce set of valid programs
- Example: Type checking
  ```java
class Foo { void f() { ... } }

Object o = new Foo();
o.g(); // not allowed, no g() method
o.f(); // not allowed, even though "works"
(Foo) o.f(); // ok; worked around type sys
```

Example: Unmatched Lock Operations

```java
if ( ... ) {
  synchronized($) {
    ... lock($);
    Checking for match ... undecidable ...
  }
  ... so Java prescribes a more restrictive, but
  unlock($);
  statically checkable constr...}
} else { ...
```