A complete PDF version of the text book is now available. The PDF version is an almost complete subset of the HTML version (where only a few, long program listings have been removed). See here.

54. Program Testing in General

54.1. Introduction to Program Testing

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

Testing according to Glen Myers book "*The art of Software Testing*"

- Program testing is the process of executing a program with the intent of finding errors
- A good test is one that has a high probability of finding an error
- Program testing cannot show the absence of errors
    - It can only show if errors are present
- It is possible to write the tests before the program

- *Test driven development*

Program testing is very resource and time consuming

It is not unusual to spend 40% of the total project efforts on testing

54.2. Overview of Program Testing

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

- Types of test
  - White box test
  - Black box test
- Levels of test
  - Unit test
  - Integration test
  - System test
- Repetition of test
  - Regression test

54.3. Testability

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

Testability refers to the software qualities that affect our ability to reveal errors

- Observability
  - The outcome of a test execution must be approachable and visible
- Controllability
  - The program input and state must be controllable prior to test execution
- Decomposability
  - A program must be broken into parts which can be tested individually
- Understandability
  - The specification of the program - *what is the correct program behaviour* - must be available

Design for Testability

54.4. Test Utopia

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

In the ideal world, all possible paths through a programs should be tested

A *total test* is a combinatorial complete test of all possible paths through a program

- Total test
  - All possible combinatoric combinations of commands and expressions should be executed relative to the control structures in a program
  - Even in small programs, the number of paths in a total test is very large
  - In many programs, there is no upper limit on the number of paths in a total test

In the real world, we need to select a subset of the paths in a total test which most likely will reveal possible program errors

54.5. From Program Test to Program Proof

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

A mathematical program proof is an alternative to program testing

- Program proofs
  - Prove that the postcondition of the program is true, given that
    - The precondition of the program is true
    - The program terminates
  - Relies on
    - A decoration of the program with mathematical assertions which reflect the intended result of the program relative to the initial program assumptions
    - Proof rules for all constructs of the program

Manually performed program proofs are likely to have more errors in the proof than in the program

Therefore only automatic program proofs will improve our confidence

54.6. White box testing

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

White box testing uses the control structures of a program unit to derive test cases

The aim of white box testing is to guarantee that all commands and expressions in a given program are executed *at least once*

- Wish list to white box testing:
  - All *independent paths* of a program unit has been executed at least once
  - All logical branches have been exercised
  - All loops and datastructures have been exercised at their boundaries

54.7. Basis Path Testing

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

Basis path testing is a white box testing technique that helps select a minimal set of paths through a program unit that covers all statements and conditions

A *path* through a program unit is a sequence of commands and conditions that starts at program entry and ends at program exit

An *independent path* is path that adds at least one new command or condition relative to already identified independent paths

- Overall approach
  - Draw a *flow chart* of the program unit
  - Abstract the flow chart to a *flow graph*
  - Find the *cyclomatic complexity* (say *n*) - a *test metric*
  - Identify *n test cases* that follow each of the independent paths

54.8. Cyclomatic Complexity - flow chart

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

```
1   public static void P(){
2     // Entry
3     while(A){
4       X;
5       if (B){
6         if(C)
7           Y;
8         else
9           Z;
10        // p
11      } else{
12         V; W;
13      }
14      // q
15    }
16    // Exit: r
17  }
```

Program 54.1 *A method P - emphasizing the program flow inside P.*

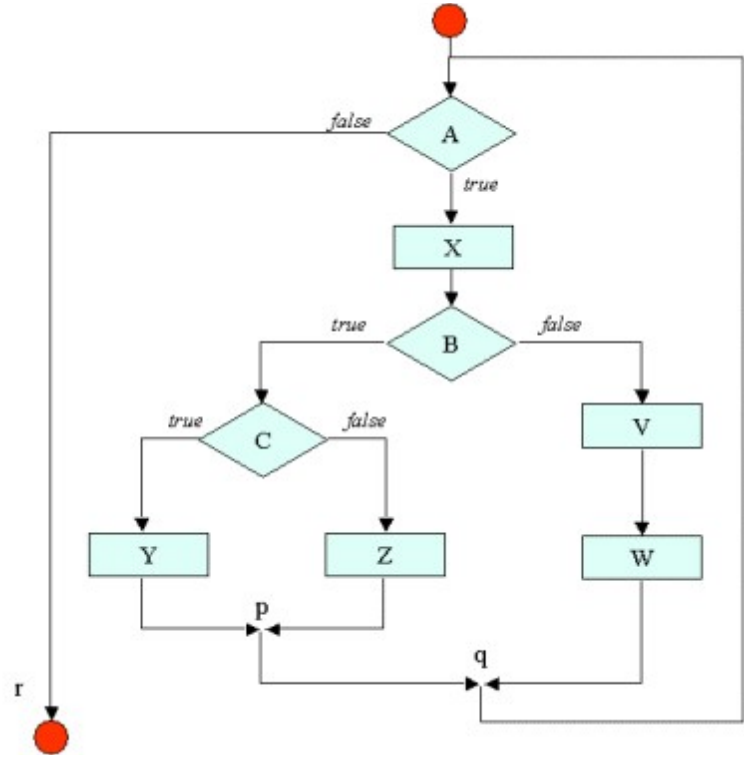

Figure 54.1 *The flow chart for the program unit P*

54.9. Cyclomatic Complexity - flow graph

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

A slight abstraction of the *flow chart* leads to a so-called *flow graph*

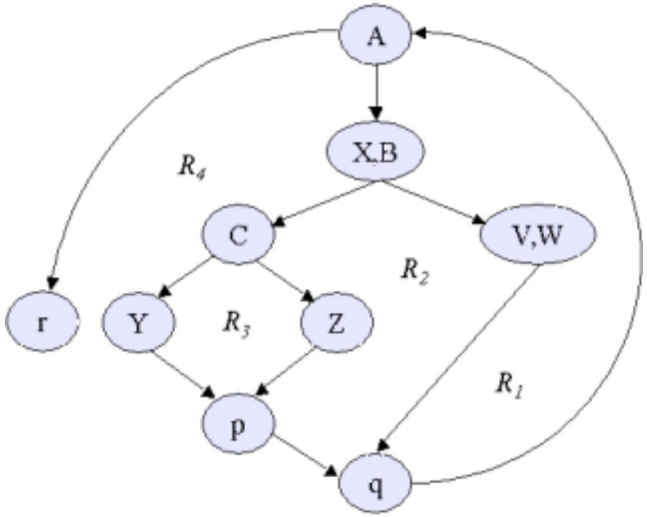

Figure 54.2 *The accompanying flow chart for the program unit P*

A, r	A, X, B, C, Y, p, q, A, r
A, X, B, C, Z, p, q, A, r	A, X, B, V, W, q, A, r

Table 54.1

54.10. Cyclomatic Complexity - Metric and Test Cases

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

Calculation of the test metric and finding test cases

- The cyclomatic complexity of the program unit P is determined
  - Can be done the flow graph - by use of simple graph theory concepts
    - The number of regions of the flow graph *or*
    - The number of *predicate notes* + 1
- Construct a test case for each independent path

- The test case should follow the control-flow of the path

Exercise 14.1. *Cyclomatic complexity of GCD*

Find the cyclomatic complexity of Euclid's, `gcd`, function in this C program.

If possible, find test cases that allow you to test each *independent path* of the `gcd` function.

Independent paths are defined here in course material.

How many test cases do you actually need to cover all source lines of the `gcd` function?

Solution

54.11. Black box testing

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

Black box testing uses the interface of a program unit to derive test cases

The aim of black box testing is to demonstrate that the program unit produces *correct* output on suitable input - relative to the *functional requirements*

- Wish list of black box testing
  - Locate functional errors
    - *Do we get the expected result on given inputs to a method?*
  - Locate interface errors
    - *Are data passed correctly to and from methods?*
  - Locate efficiency errors

- *Is the method fast enough?*

54.12. Input to a Black Box Test

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

It is not realistic - from a combinatoric point of view - to test a program unit on all possible inputs

Therefore, the choice of input values to black box testing is an important concern

An *equivalence partitioning* splits a set of input values in a (small) number of classes.

It is hypothesized that it is sufficient to test the program unit on a single *representative* from each class

- Include boundary representatives
  - Such as empty collections and full collections
- Include one or more representative for invalid input
  - Typecheck and use of preconditions may eliminate this need

54.13. Example of Equivalence Partitioning (1)

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

```
1   // Exchange element i and j in table
2   public void SwapElements<T>(T[] table, int i, int j){
```

```
3       ...
4   }
```

Program 54.2 *The method signature SwapElements.*

- Equivalence partitions - all combinations of
  - `table`
    - `table` is empty, `table` is singular, `table` with two or more elements
  - `i` and `j`
    - One of `i` and `j` are outside the bounds of `table`
      - `i` inside and `j` outside, `i` outside and `j` inside, both are outside
    - Both `i` and `j` are inside the bounds of `table`
      - `i < j`, `i > j`, `i = j`

Use of a stronger precondition limits the number of test cases

54.14. Example of Equivalence Partitioning (2)

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

```
1   // Find the largest element in table in between the indexes from and to.
2   // Return the index of this element.
3   // Precondition: 0 <= from <= to <= table.Length-1
4   public int FindMaxIndex<T>(T[] table, int from, int to)
5     where T: IComparable<T> {
6         ...
7   }
```

Program 54.3 *The method signature FindMaxIndex.*

- Equivalence partitions - all combinations of
  - `table`
    - `table` is empty/singular, largest element is first in `table`, largest element is last in `table`, largest element is in mid of `table`, all elements in `table` are equal
  - `from` and `to`
    - `from` and `to` are located at the boundaries of `table`
    - `from = to`
    - `from = to - 1`
    - `from` and `to` are not close to each other

A well-chosen precondition limits the number of test cases substantially

54.15. Regression testing

Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index

The purpose of regression test is to find regression errors

A regression error is an error in a previously correct program, which recently has been modified

A regression error is an error *which not used to be there*

- After modification of part P in a program Q
  - Test that the part P works correctly

- Test that the overall program Q has not been affected by the modification

- After modification of part P in a program Q
  - Test that the part P works correctly
  - Test that the overall program Q has not been affected by the modification

54.16. References

[-] Software Engineering - A Practitioner's Approach: Roger S. Pressman, McGraw-Hill, 1992

Generated: Monday February 7, 2011, 12:23:24