SOFTWARE QUALITY ASSURANCE

John Abbott College JPC

Test-Case Design

M. E. Kabay, PhD, CISSP
Director of Education, NCSA
President, JINBU Corp

Test-Case Design

- Design Philosophy
- Equivalence class analysis
- Boundary analysis
- Testing state transitions
- Testing race conditions and other time dependencies
- Function-equivalence testing
- Regression testing
- Error-guessing

Test-Case Design Philosophy

- Complete testing is impossible
- Therefore define subset of test cases likely to detect most (or at least many) errors
- Intuitive approach is “random-input testing”
  - sit at terminal
  - invent test data at random
  - see what happens
  - worst possible approach

Equivalence Partitioning

- “A group of tests forms an equivalence class if you believe that:
  - They all test the same thing.
  - If one test catches a bug, the others probably will, too.
  - If one test doesn’t catch a bug, the others probably won’t either.”

    -- p. 126
- Subjective process
- Goal is to reduce many redundant tests to a smaller number giving same information
- Focus especially on invalid inputs

Equivalence Partitioning

Must first identify the equivalence classes

- Range: below, within, above
- Number: fewer, valid, higher
- Set: all members & 1 non-member
- Requirement (set of 1): valid & invalid
- On doubt, split class

Equivalence Partitioning

Then define specific test cases

- At least one test case for every valid equivalence class
- At least one test case for every invalid equivalence class
- See Figure 7.1, p. 127 in text

Boundary-Value Analysis

- Cases at boundaries have high value for testing
- Select cases just below, at and just above limits of each equivalency class
- Some testers include mid-range value as well just for additional power of test

Testing State Transitions

- Every change in output is a state transition
- Test every option in every menu
- If possible, test every pathway to every option in every menu
- Interactions among paths
  - draw menu maps
  - identify multiple ways of reaching every state
  - keep careful records of what you test (can get confusing)

Testing Race Conditions and Other Time Dependencies

- Check different speeds of input
- Try to disrupt state transitions (e.g, press keys while program switches menus)
- Challenge program just before and just after time-out periods
- Apply heavy load to cause failures (not just poor performance)

Function-Equivalence Testing

- Use a program that produces known-good output
- Feed same inputs to both the standard program and the program under test
- Compare the outputs
- Automated testing techniques can help
  - for numerical and alphanumerical output
  - for real-time process-control applications

Regression Testing

- Did the bug get fixed?
  - Some programmers patch symptom
  - Few test effectively
- Check that you can produce bug at will in bad version of code
- Use same tests on revised code
  - Stop if bug reappears
  - Push the testing if bug seems to have been fixed

Error Guessing

- Need intuitive grasp of what is likely to go wrong in a program
- Look at typically difficult cases (e.g., wrong number of parameters)
- Examine cases that are not explicitly defined in specifications (assumptions by programmer)