Overview

- **Testing**
  - Running program against test input cases, chosen to reveal correct and incorrect behaviors

- **Debugging**
  - Identifying, localizing, and fixing problems
Test Case Design

- **Exhaustive**: Try every possible input
  - Usually NOT feasible

- **Black box**: Guess at “problem” inputs
  - Based only on problem description
  - Often misses problems

- **White box**: Use knowledge of program structure to design cases that exercise it
Example: GroupCheck

- Program supposed to check for balanced parentheses, using <> [] {} () as the groupings

- Report “location of an error”
  - `<pre>[html{}a href="works.html"]</pre> is not valid at position 11 because the ] is trying to close the {
  - 000000000011111111112
      012345678901234567890
      `<pre>[html{}a href="works.html"]</pre`

- Shows that 0-origin indexing is used for location, and that “first point at which error detected” is meant
GroupCheck (continued)

- **Exhaustive**
  - How many different strings are needed?

- **Guess at possible errors**
  - Other?
    - Mismatches: "{ }", "[ ]", "<}" ... How many?
    - What about "too many or too few parens"?
      - "{"", "}", "("", ")" Any more?

- Try complex correct strings, too
  - Each non-paren character as a single character String
  - Strings which nest parens deeply, with + w/o errors
Debugging

- **Identify error (GroupCheck example)**
  - Is error location correct? Carefully count!!
    - “Correct” is special case of location: −1.
- **Locate statements that produce this error**
  - Hypothesize, then verify hypothesis by getting more output from the program near error location
    - Use interactive debugger?
      - I haven’t found good way to get it to show variable values
    - Add System.out.println() statements
- **Modify program to fix error**
  - Try hard not to introduce new errors!
Interactive Debugger

- Allows you to stop program at “step” you choose “on the fly”
  - At a statement or method entrance
    - Only if some test on variables is true
    - After K executions of this statement
  - Whenever something changes some variable’s value (watchpoint)
- While stopped, lets you examine contents of any variables.
  - Variable values should be displayed in human-readable form
- Allows resuming execution, to reach next suspected “error site”
Eclipse Debugger

- Displays variable values as numeric pointer to an Object.
  - Not helpful
  - There’s some way to “evaluate expressions”, but those expressions have already be in the code.
    - Not flexible
    - I haven’t figured out how to do this

- Using println(), I’ve had to call Arrays.deepToString() to print contents of an array
Fixing Errors

- **Understand the program**
  - READ what it SAYS, not what you THOUGHT you wrote
  - Explain what it does to someone else who questions whether:
    - It does what you say it does
    - What you say it does will solve the problem
    - Other cases you haven’t considered exist
  - Use debugging statements to verify understanding, show what’s going on inside complex parts
  - Place debugging statements to show if you’ve found the source of erroneous output, follow flow of bad data backwards.
  - Add extra test cases to test hard-to-understand parts
Fixing Errors (2)

- Acid test of understanding: Try fixing what you think is wrong, verify:
  - That it fixed the problem you were working on
  - Rerun ALL test cases, verify they’re still OK

- I recommend fixing errors *in the order you find them*
Binary Search for some errors

- Use when you need to find which statement is changing some variable, and your debugger does not allow “watchpoints”

- Run program “half way”, see if the variable has changed
  - If not, run half the remaining steps
  - If so, start program over, stop ¼ way
  - Repeat till problem localized