13
The Whole and the Parts

The Whole and the Parts

- Designing the Bugs Out
- Component Debugging
- System Debugging

Bug-proofing the design

- Mismatched assumptions cause the worst bugs
- Brooks' strategies for ensuring conceptual integrity reduce mismatched assumptions
- Vyssotsky of Bell Labs stresses
  - Specifying the product completely
  - Testing the specification by separate staff

Top-down design

- Niklaus Wirth described top-down design in 1971
- Brooks applies it to systems, as well as programs
  - Design through refinement steps
  - Sketch top-level function
  - Break function into smaller sub-functions
  - Refine the algorithm and data in each step
  - Encapsulate related functions as modules
- Modularity determines adaptability to change

Benefits of top-down design

- Hierarchical structure clarifies the specification
- Modular partitioning reduces system bugs
- Detail suppression exposes high-level faults
- Progressive refinement permits staged testing

- Stepwise refinement does not eliminate errors
- But it does reveal gross design faults earlier, and
- Reduces the temptation to sustain a poor design

Structured programming

- Dijkstra applied Böhm and Jacobini's theory that shows all procedural programs are composition of
  - Sequence BLOCK, or semicolon (;)
  - Alternation IF-THEN
  - Iteration WHILE-DO
- Some authors specifically prohibit use of unstructured branching, e.g., GOTO
- Brooks emphasizes value of thinking about control structures instead of control statements

Structured programming

- Dijkstra applied Böhm and Jacobini's theory that shows all procedural program control comprises:
  - Sequence BLOCK, or semicolon (;)
  - Alternation IF-THEN
  - Iteration WHILE-DO
- Some authors specifically prohibit use of unstructured branching, e.g., GOTO
- Brooks emphasizes value of thinking about control structures instead of control statements

Component debugging

- Goal: Isolate faults to a few statements
- Plan: Correlate the input/output behaviour of the component under test with the progression of control flow through the program statements
- Strategy: Depends on resource constraints
  - Space, i.e., computer memory
  - Time, i.e., execution speed, input/output rates
  - Control, e.g., breakpoints, single-step
  - Effort

On-machine debugging

- Strategy: Examine machine state (e.g., registers) at predetermined points in the control flow
- Facilities:
  - Navigable object code
  - Breakpoint editing; Run control
  - Memory monitor
- Effort: Planning and inserting the breakpoints
- Today: Print statements and variable monitors
- Exclusions: Real-time control; Concurrency

Memory dumps

- Strategy: Examine the entire machine state at a single predetermined point in the control flow
- Facilities:
  - Core dumper with large offline storage
  - Hardcopy or interactive memory map
- Effort: Interpreting the large machine state
- Today: Very rarely used
- Exclusions: Large or distributed applications

Snapshots

- Strategy: Examine selected portions of the machine state at a single predetermined points
- Facilities:
  - Snapshot capture
  - Memory map
- Effort: Interpreting the large machine state
- Today: Stack traces
- Exclusions: Large or distributed applications

Interactive debugging

- Strategy: Execute the subject program under the control of a supervisory program
- Facilities:
  - Multitasking operating system
  - Run/Stop/View controls
- Effort: Resisting temptation to not think (Brooks suggests 1:1 desk-to-lab time)
- Today: Symbolic debuggers, Interpreters
- Exclusions: Real-time control; Concurrency

System debugging

- Will take longer than expected
- Requires a plan

Use debugged components

- Isolate faults as much as possible
- Resist temptation to:
  - Use the system as its own test scaffolding
  - Proceed with documented, uncorrected faults
- These are rationalizations for delayed schedules

Build plenty of scaffolding

- Brooks suggests up to half is scaffolding
- Dummy component
  - Compliant interface with nonsensical data
- Miniature file
  - Compliant but limited data, e.g., handwritten
- Dummy file
  - Limiting case for miniature file
  - Empty source and sink, e.g., /dev/null on Unix

Control changes

- Centralize the authority for the system build
- Stage the version progression from individual programmers towards the central build
- Stage the approval progression from central build towards release, e.g., DEV, TEST, PROD
- Clearly demarcate the official versions
  - E.g., 0.0.0 (major.minor.build) version scheme
  - E.g., CVS tag

Quantize updates

- Balance responsiveness with stability
  - Build frequently to include valuable changes
  - But not so frequently as to confuse developers