Hierarchical decomposition

- In well-designed code:
  - Bottom level code units are methods/functions (~1–100 LOC)
  - Modules have up to a couple of dozen ops
  - At most a couple of dozen modules to implement related functionality
- Top-level modules scale to ~10k LOC progs
- Modularity alone isn’t enough for large systems—need hierarchy of modules
Hierarchical decomposition

- Divide and conquer: must break large modules into smaller modules
- Multiple levels of hierarchy
- Good design if: only need to think about one module, one level at a time
- How to manage large-scale design?

Modular structure

- Program is composed of modules
- One module depends on another if it uses a value, function, or type from it

- Module Dependency Diagram (MDD) helps understand large-scale program structure
Keeping dependencies simple

- Too many dependencies or cycles: harder to debug, maintain, extend software

Example

Box size determined by source size
Example 2

(generated from OCaml source by dep2dot)

Bottom-up development

• Bottom-up: develop modules before the modules that depend on them

• Advantage: catch key technology/performance issues early

• Advantage: always working code, unit testing

• Disadvantage: catch large-scale design flaws late
Top-down development

- Top-down: develop using modules before modules they depend on

- **Advantage:** get high-level design right from start, do integration testing
- **Advantage:** easier to design interfaces well, quickly spec out system

- **Disadvantage:** harder to test until program complete

Unit/component testing

- Test modules through their interfaces
- Test each implementation against interface separately
- Write *test harness* to test each module
- Good match to bottom-up development
System and integration testing

- **Integration testing**: test many modules together
  - Do modules compose successfully?
- **System testing**: test entire system
  - May also validate requirements specification: is this what we want?
  - Check high-level structure, UI of program
  - Good for top-down development
- **Stubs**: low-cost replacements of missing module implementations
  - May partially simulate functionality
  - May be slow/simple/cheap implementations of functionality

Top-down or bottom-up?

- Depends on the project!
- **Goal**: avoid huge redesign cost
  - Minimize risk: resolve uncertainties early
  - UI/high-level design: top-down
  - Core technology/performance: bottom-up
- Usually some mix of both strategies & both unit and integration testing
Waterfall model

- An abstraction of different activities in software projects
- Not always this neat!

Validation:
- are requirements right?

Verification:
- does impl meet spec?
  - Formal verification
  - Testing
  - Assurance: reasonable confidence that right system has been built, correctly

Testing

- Goal is assurance that system works
- (Completely) working system is free of faults:
  - Errors in requirements
  - Errors in specifications
  - Errors in implementation

Strategy: build a set of test cases that if passed give assurance
- Test: compare actual to expected outputs
- Test case: inputs to program/component, and expected outputs
- Collection of test cases: test suite
Coverage

• How can finite test cases give strong assurance?
• Key: test cases that have good coverage of possible faults
• Exhaustive testing:
  – Test all possible inputs (against spec; against other, simpler, obviously correct implementation)
  – Usually infeasible (input space too large)
    `val plus : int->int->int` has $2^{64}$ inputs (584 years at 1/ns)
  – Sometimes can exhaustively test up to some input “size” -- faults usually have small counterexamples.
• Random testing:
  – Generate inputs randomly. Idea: if all tests pass, unlikely to see faults in production
  – Problem: only works if inputs have same random distribution as “in nature”. Hard!
• Usually must design test cases -- an art

Black-box testing

• Idea: test cases achieve coverage based on specification only
  – Aka “closed-box”
• Idea: specification divides space of possible inputs into different regions.
  – Test boundary values and corner cases
• Examples:
  ```
  plus : int->int->int
  lmax : int list -> int
  ```
• Advantages:
  – Can write test cases before implementation
  – Can write test cases independently
  – Helps find problems in specifications
• Disadvantages:
  – May not test all code or test code thoroughly
• A good place to start designing tests
Glass-box testing

• Using the implementation to design test cases.
  Some approaches:

1. Use the AF and RI to identify interesting parts of input space
2. Statement coverage: ensure every statement or expression is evaluated in some test case
3. Branch (or condition) coverage: ... every branch is tested both ways
4. Entry/exit coverage: ... every function entry/exit is tried
5. Path coverage: ... every path is followed

Regression testing

• Test suites are valuable!
• 1/3 of bug fixes introduce a new bug
• Regression testing:
  1. record outputs to entire test suite
  2. on changes to system, check that outputs haven’t changed (meaningfully)
• Push-button automation is key. Lots of tools to help: expect, JUnit, ...
  • Test early and often!