CS 2110: Implementation & Testing

Interface spec clauses (audience: client)

- Returns / creates: postconditions
- Requires / Checks: preconditions
  - Checks: promises to check for precondition violation & halt program (for debugging)
    - Checks: x ≥ 0
- Effects / Modifies: side effects
  - ↳ unspecified effect

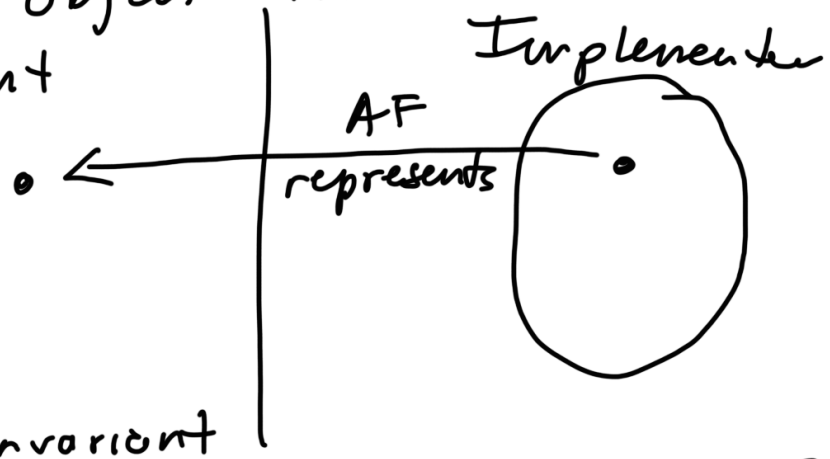
Default conditions (for brevity)

- arguments, instance vars, results not null
- assumed non-null unless stated otherwise.

Documenting implementation (audience: maintainers)

Key information:

1. Represents clause (abstraction function)
   How does client view interpret object state



2. Class invariant
   - which object states are legal?
3. Specs for private/protected methods
4. Algorithm explanations
   - inside method body
   - minimize interleaving
   - together at top of method body

Module dependency

Control code dependencies (loose coupling)

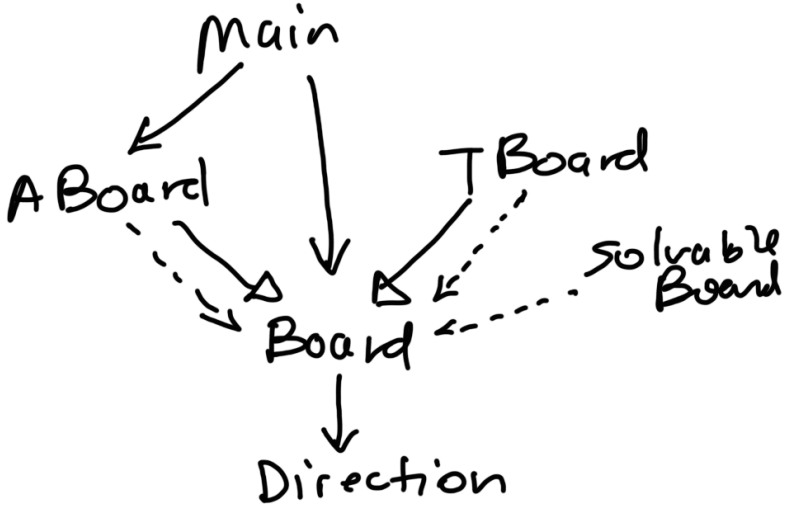
- clients depend on impl.
- subclasses depend on superclasses.

Module dependency diagram (MDD)

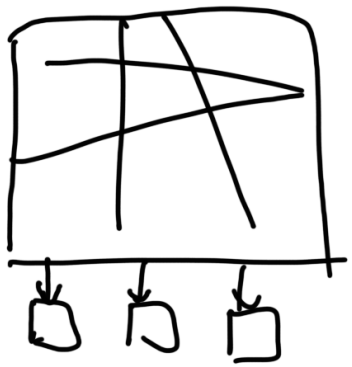
Graph: edges = dependencies (
node = modules (classes)

UML notation

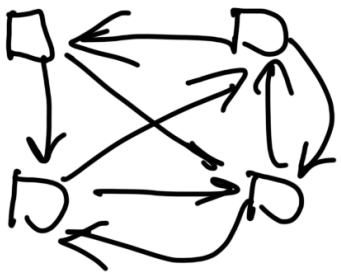
- depends on (uses) →
- subclass of —▷
- subtype of - - - -▷
- - - - - - ->



MDD design



"big ball of mud"
— no separation of concern



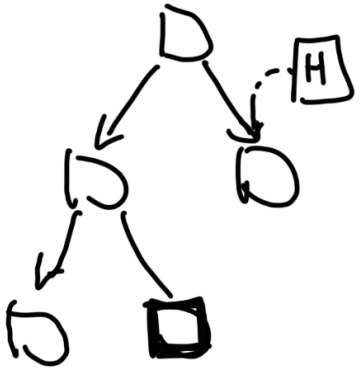
cycles : tight coupling

Implementation Strategies

Top-down

- start at root of MDD work downward.

+ high-level design early
+ always demoable
+ test w/ stubs



Bottom-up

- only use complete modules.

+ unit testing (with harness)
+ know tech. feasibility early

Which? Reduce risk

UI risk ⇒ top-down
performance ⇒ bottom-up

Key: plan, agree

Testing

Untested = broken.
⇒ early, continuous testing

Develop tests before, during impl.

Early tests:
- test specs
- test tests
- test asserts

Test automation

- testing should be easy & reproducible.
- JUnit: your friend
- Test suite is key code.

Regression testing: 1/3 bug fixes create new bugs.
⇒ keep testing whole test suite.

Test design

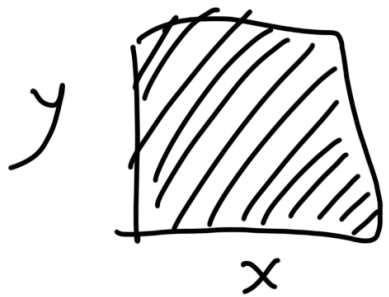
Finite # tests ⇒ confidence over all inputs?

Problem: too many inputs.

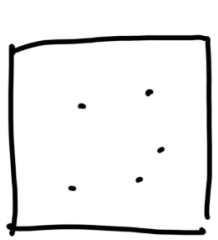
Testing: x/y on doubles : 2^{128} tests

Can't exhaustively (Intel FDIV bug)

How achieve coverage of possible inputs?

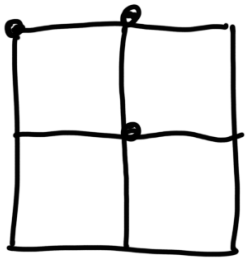


Idea 1: choose tests for coverage of inputs



"typical inputs"

+



edge/corner cases
"where are the edges"
usual mistake: skip these

Black box testing

- Use specs to guide coverage.

can write tests before code!

Ex/ remove element from array,
typical: remove an element.
+ elem not in array
+ elem is first/last in array
+ empty array
+ occurs twice
+ single-elem array

Glass-box testing

- use impl. code to guide coverage. (& define edges)
  - cover all code
  - all methods
  - all lines & all paths
  - all data structures