Modular Design

(Introduction through examples)

Eugene Register Guard, 10 April 1999, Pg. 5B:

Washington driver's manual, test flawed

In recent months, the department has used a computerized test that was known to have some answers that conflicted with state law or the state-issued study guide. […] Some 97,414 tests were administered during the three months, with 36,391 failures reported.

Some computers used to administer the tests are still marking correct answers as wrong, said department spokeswomen Suzanne Taylor. **The problem lies in getting updated software for the the computers' testing program, she said.**

How hard can this be?

- How can it happen that Washington has to wait for updated test software with the correct answers?
- How would you design that system? Would your design be vulnerable to this problem?

Information Hiding

- Information =
  - Design decisions (data structures, algorithms, …)
  - Device characteristics
  - Platform
  - …
- Hiding =
  - Isolating dependence
  - Permitting independent change

Hide what?

Information hiding is fundamentally a strategy of "design for change"

- Hide what may change
  - Over time
  - As the design is refined
  - Between versions, platforms, etc.
- Hide so that it can change *independently*

What will you hide?

Exercises

- Application: Electronic ignition
  - Delco electronics, ignitions for GM cars
  - Same basic design: gather sensor data, adjust timing forward or back according to driving conditions
- Application: Mosaic web browser
  - X, Mac, Windows versions
  - Evolution of: http, html, media types
- Application: Tektronix printer engine
  - Various ink jet and laser hardware, color and monochrome

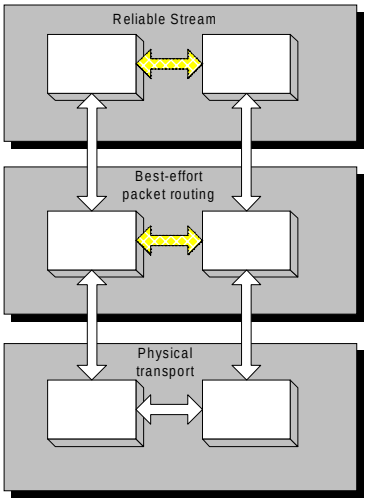
Hiding is not Free

- Hiding means pretending not to know
  - Not taking advantage of information that is "hidden" in other modules
  - Not using the faster special-case (optimizing)
  - Coding for all cases, not just those that can actually occur
- So we hide some things and reveal others
  - Fundamental assumptions = unlikely to change
    - and we pay the price if they do
  - Pay the price of hiding for what is most likely to change

Language Support

- OO ≠ Information Hiding
  - Helpful, but neither necessary nor sufficient
- Scope rules help "enforce" secrets
  - Prevent access to representations we have chosen to keep secret, e.g., data structures
  - Object-based and object-oriented languages are good at keeping data structures secret, ***if*** the object interfaces are well designed
    - abstract = more than one possible instance
- Limited usefulness for some secrets
  - e.g., concurrency control strategy

Non-OO Info Hiding Example (1)



- Layered protocol structure
  - Each layer hides the the layers below
- True abstractions
  - Can substitute ATM for Ethernet, etc.
- Implemented since 70s in straight C

Non-OO Info Hiding Example (2)

- Consider the structure of XML (or HTML, SGML)
  - Basic tag syntax <foo> … </foo>
  - Particular application-specific tags
- Hiding:
  - Particular tag-sets hidden from XML parser and many generic XML tools
  - Concrete syntax hidden from application-specific tools
    - parse/unparse components hide parsing, reveal tree structure
- Similar “thin tree, fat tree” division in compilers

Exercise: E-Card Exchange

- Application: Electronic exchange of "business card" information
  - Exchange format + program support for sending and receiving contact information
- Requirement: Interoperate with off-the-shelf PIMS (contact managers)
  - "product line" must interoperate with several
- Problem: Conversion between native format and various PIM formats