CSE503: SOFTWARE ENGINEERING

DESIGN

David Notkin
Spring 2011

Today

2

- Very brief project #1 descriptions
- Software design introduction

Functional decomposition

Already done ☺

3

- Divide-and-conquer based on functions
  - **`input; compute; output`**
- Then proceed to decompose compute
- This is stepwise refinement (Wirth, 1971)
  - In essence, refining until implementable directly in a programming language (or on an architecture)
- There is an enormous body of work in this area, including many formal calculi to support the approach
  - Closely related to proving programs correct
- More effective in the face of stable requirements

Information hiding

4

- Information hiding is perhaps the most important intellectual tool developed to support software design [Parnas 1972]
  - *Makes the anticipation of change a centerpiece in decomposition into modules*
- Provides the fundamental motivation for abstract data type (ADT) languages
  - And thus a key idea in the OO world, too
- The conceptual basis is key

Basics of information hiding

- Modularize based on anticipated change
  - Fundamentally different from Brooks' approach in OS/360 (see old and new MMM)
- Separate interfaces from implementations
  - Implementations capture decisions likely to change
  - Interfaces capture decisions unlikely to change
  - Clients know only interface, not implementation
  - Implementations know only interface, not clients
- Modules are also work assignments

Anticipated changes

- The most common anticipated change is "change of representation"
  - Anticipating changing the representation of data and associated functions (or just functions)
  - Again, a key notion behind abstract data types
- Ex:
  - Cartesian vs. polar coordinates; stacks as linked lists vs. arrays; packed vs. unpacked strings

Classic Parnas example: KWIC

Key Word in Context

- Example input

  One flew over
  the cuckoo's nest
  but it wasn't me

- Example output

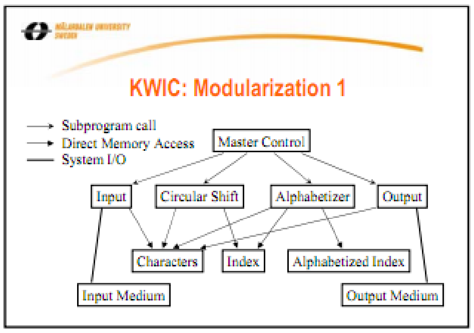
  but it wasn't me
  cuckoo's nest the
  flew over One
  it wasn't me but
  me but it wasn't
  nest the cuckoo's
  One flew over
  over One flew
  the cuckoo's nest
  wasn't me but it

Functional decomposition

- A traditional would decompose based on the four basic functions performed: **input, shift, alphabetize, output,** coordinated by **main**
- Data communication is through shared storage and can be an unconstrained read-write protocol because of **main**
- Parnas argues some serious drawbacks in terms of its ability to handle changes
  - In particular, a change in data storage format will affect almost all of the modules.
  - Similarly changes in algorithm and enhancements to system function are not easily handled
- Finally, reuse is not well-supported because each module of the system is tied tightly to this particular application
- Strengths of this decomposition?

http://www.idt.mdh.se/kurser/cdt413/V08/lectures/l2.pdf

9



Information hiding decomposition

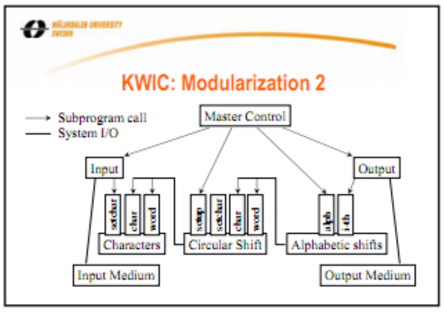
ADT-oriented

10

- Data representations not shared computational components
- Instead, each module provides an interface that hides the data representation, allowing only access through the interface
- Change is easier
  - Both algorithms and data representations can be changed in individual modules without affecting others modules
  - Reuse is better supported because modules make fewer assumptions about the others with which they interact

http://www.idt.mdh.se/kurser/cdt413/V08/lectures/l2.pdf

11



Anticipating change?

12

- A fundamental assumption of information hiding is the ability to anticipate change
- Can we do this effectively?
- If not, is information hiding unreasonable to pursue?

From a Case for Extreme Programming

- … UC-Berkeley political scientist Aaron Wildavsky … lists two categories of risk management, anticipation and resilience.

  "Anticipation is a mode of control by a central mind; efforts are made to predict and prevent potential dangers before damage is done. Resilience is the capacity to cope with unanticipated dangers after they have become manifest, learning to bounce back. … Anticipation seeks to preserve stability: the less fluctuation, the better. Resilience accommodates variability; … The positive side of anticipation is that it encourages imagination and deep thought. And it is good at eliminating known risks. It can build confidence. But anticipation doesn't work when the world changes rapidly, and in unexpected ways. It encourages two types of error: hubristic central planning and overcaution. (Postrel)"

Is representation change less common?

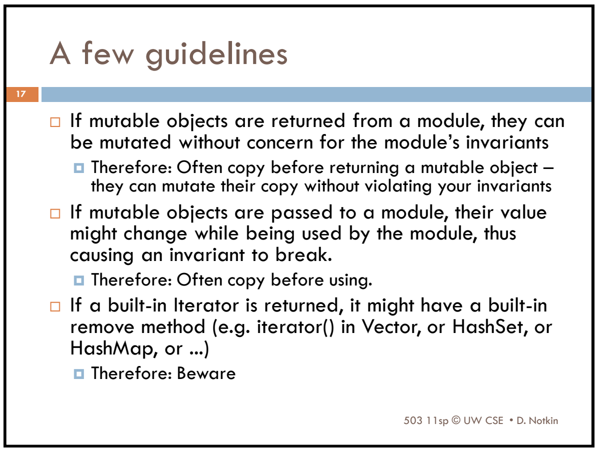
- We have significantly more knowledge about data structure design than we did 25 years ago
- Memory is less often a problem than it was previously, since it's much less expensive
- Therefore, should we think twice about anticipating that representations will change – after all, there is an opportunity cost

Module semantics remain unchanged

- The semantics of the module remain unchanged when implementations are changed: the client should only care if the interface is satisfied
  - But what captures the semantics of the module? The signature of the interface? Performance? What else?
- Who cares more about this issue? The clients of the module or the implementors of the module?

Representation exposure

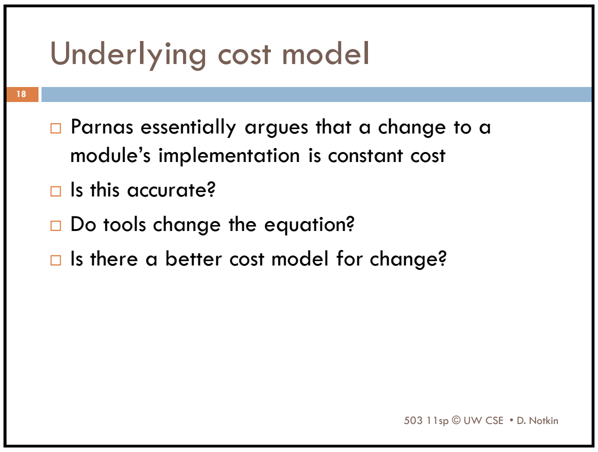
- Representation exposure occurs (in ADTs specifically, but in information hiding more generally) when a module interface and implementation allow a client to learn "more than they should" about an implementation
- This can, of course, lead to both unexpected consequences and also a dependence by the client on the specific implementation
- It is generally the case that careful assessment and reasoning about the abstraction function and the representation invariant can identify exposures



A few guidelines

17

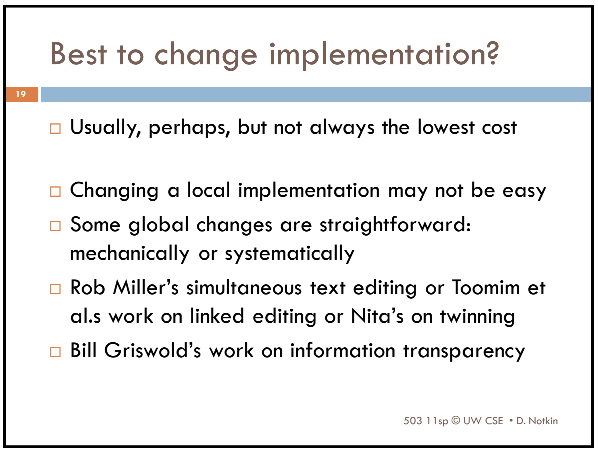
- If mutable objects are returned from a module, they can be mutated without concern for the module's invariants
  - Therefore: Often copy before returning a mutable object – they can mutate their copy without violating your invariants
- If mutable objects are passed to a module, their value might change while being used by the module, thus causing an invariant to break.
  - Therefore: Often copy before using.
- If a built-in Iterator is returned, it might have a built-in remove method (e.g. iterator() in Vector, or HashSet, or HashMap, or ...)
  - Therefore: Beware



Underlying cost model

18

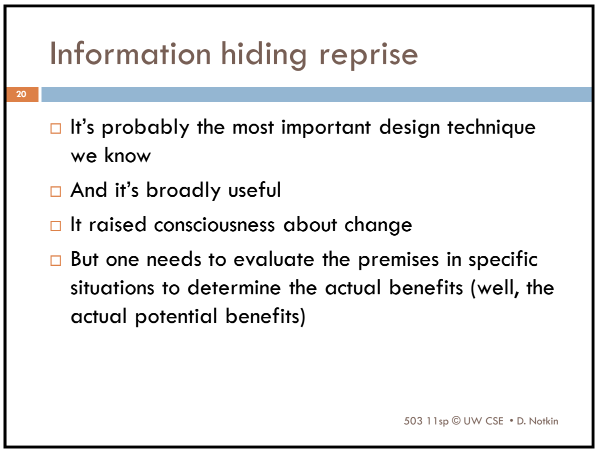
- Parnas essentially argues that a change to a module's implementation is constant cost
- Is this accurate?
- Do tools change the equation?
- Is there a better cost model for change?



Best to change implementation?

19

- Usually, perhaps, but not always the lowest cost
- Changing a local implementation may not be easy
- Some global changes are straightforward: mechanically or systematically
- Rob Miller's simultaneous text editing or Toomim et al.s work on linked editing or Nita's on twinning
- Bill Griswold's work on information transparency



Information hiding reprise

20

- It's probably the most important design technique we know
- And it's broadly useful
- It raised consciousness about change
- But one needs to evaluate the premises in specific situations to determine the actual benefits (well, the actual potential benefits)

Aside: Information Hiding and OO

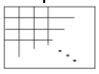
21

- Are these the same? Not really
  - OO classes are chosen based on the domain of the problem (in most OO analysis approaches)
  - Not necessarily based on change
- But they are obviously related (separating interface from implementation, e.g.)
- What is the relationship between sub- and super-classes?

Dependence on implementation

22

- Gregor Kiczales et al.: clients indeed depend on some aspects of the underlying implementations in a broad variety of domains and situations
- What happens when the implementation strategy for a module depends on how it will be used?
- Aren't we supposed to separate policy from mechanism?
- Example: spreadsheet via many small windows?



```
for i = 1 to 100
  for j = 1 to 100
    mkwindow(100, 100, i*100, j*100);
  end
end
```

Poor performance often leads to…

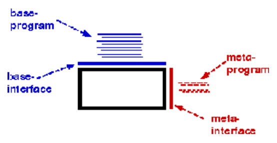
"hematomas of duplication"

"coding between the lines"

Open implementation

24

- Decompose into base interface (the "real" operations) and the meta interface (the operations that let the client control aspects of the implementation)
- Arose from work in (roughly) reflection in the Meta-Object protocol (MOP) and led to the development of aspect-oriented programming (which we will look at next week, from a modularization point of view)



Meta interface examples

25

- C's **`register`** storage class
  - "A declaration of an identifier for an object with storage-class specifier **`register`** suggests that access to the object be as fast as possible."
- Unix **`nice`**
- High-Performance Fortran
  -
```
        REAL A(1000,1000),B(998,998)
  !HPF$ ALIGN B(I,J) WITH A(I+1,J+1)
  !HPF$ DISTRIBUTE A(*,BLOCK)
```
- ...many many more! Quick examples from you?

Layering [Parnas 79]

26

- A focus on information hiding modules isn't enough
- One may also consider abstract machines
  - In support of program families, which are systems that have "so much in common that it pays to study their common aspects before looking at the aspects that differentiate them"
- Still a focus on anticipated change

The **`uses`** relation

27

- A program **A** uses a program **B** if the correctness of **A** depends on the presence of a correct version of **B**
- Requires specification and implementation of **A** and the specification of **B**
- Again, what is the "specification"? The interface? Implied or informal semantics?

`uses` vs. **`invokes`**

28

- These relations often but do not always coincide
- Invocation without use: name service with cached hints
- Use without invocation: examples?

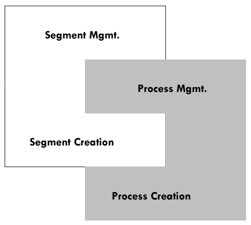
```
ipAddr := cache(hostName);
if wrong(ipAddr,hostName) then
   ipAddr := lookup(hostName)
endif
```

Parnas' observation

- A non-hierarchical uses relation makes it difficult to produce useful subsets of a system
- So, it is important to design the **`uses`** relation using these criteria
  - **A** is essentially simpler because it uses **B**
  - **B** is not substantially more complex because it does not use **A**
  - There is a useful subset containing **B** but not **A**
  - There is no useful subset containing **A** but not **B**

Modules and layers interact?

- Information hiding modules and layers are distinct concepts
- How and where do they overlap in a system?



Language support?

- We have lots of language support for information hiding modules
  - C++ classes, Ada packages, etc.
- We have essentially no language support for layering
  - Operating systems provide support, primarily for reasons of protection, not abstraction
  - Big performance cost to pay for "just" abstraction
- General observation: design ideas not encoded in a language are less likely to be used