CSE584: Software Engineering

Lecture 2 (April 8, 1997)

David Notkin
Dept. of Computer Science & Engineering
University of Washington
www.cs.washington.edu/homes/notkin

UW CSE
CSE584: Software Engineering

Notkin (c) 1997 1

Lecture 2, Outline [approximate minutes]

- Administrivia (rescheduling) [5]
- Basic design concepts (coupling, cohesion, etc.) [20]
- Information hiding [20]
- Layered systems [20]
- Break [10]
- Roundtable:
  - Design problems you face [30]
  - How you do design [15]
- Implicit invocation & mediator-based design [30]
- Wrap-up and slop [20]

Notkin (c) 1997 2

Design: management of complexity

- We have to decompose large systems to be able to build them
  - » The "modern" problem of composing systems from pieces will be equally or more important
- For software, we have decomposition techniques that are distinct from those used in physical systems
  - Very few constraints are imposed by the material

Notkin (c) 1997 3

Design, design, design

- Design is a continuous activity in software development
  - High-level (architectural) design
    - » What pieces? How connected?
  - Low-level design
    - » Should I use a hash table or binary search tree?
  - Very-low-level design
    - » Variable naming, specific control constructs, etc.

Our primary focus

Notkin (c) 1997 4

Which decomposition?

- How do we select a decomposition?
  - We determine the desired criteria
  - We select a decomposition (design) that will achieve those criteria
- In theory, that is; but in practice, it's hard to
  - Determine the desired criteria with precision
  - Tradeoff among various conflicting criteria
  - Figure out if a design satisfies given criteria

Notkin (c) 1997 5

Structure

- The focus of most design approaches is *structure*
- What are the components and how are they put together?
- Behavior is important, but less so than structure (during architectural design)

Notkin (c) 1997 6

So what happens?

- People often buy into a particular approach or methodology
  - Ex: functional decomposition, data decomposition, object-oriented programming, information hiding, layering, JSD, Hatley-Pirbai, etc.
- "Beware a methodologist who is more interested in his methodology than in your problem." (Michael Jackson)

UW CSE CSE584: Software Engineering Notkin (c) 1997 7

Properties of design [Bergland]

- Cohesion
- Coupling
- Complexity
- Correctness
- Correspondence

- Makes designs "better", one presumes
- Worth paying attention to

UW CSE CSE584: Software Engineering Notkin (c) 1997 8

Cohesion

- The reason that elements are found together in a module
  - Ex: coincidental, temporal, functional, …
- The details aren't critical, but the intent is useful
- During maintenance, one of the major structural degradations is in cohesion
  - Need for "logical remodularization"

UW CSE CSE584: Software Engineering Notkin (c) 1997 9

Coupling

- "Strength of interconnection between modules"
- Hierarchies are touted as a wonderful coupling structure, limiting interconnections
- Coupling also degrades over time
  - "I just need *one* function from that module…"
  - Low coupling vs. no coupling
- Can't live without coupling

UW CSE CSE584: Software Engineering Notkin (c) 1997 10

It's easy to...

- ..reduce coupling by calling a system a single module
- …increase cohesion by calling a system a single module

- No satisfactory measure of coupling
  - Either across modules or across a system

UW CSE CSE584: Software Engineering Notkin (c) 1997 11

Complexity

- Well, yeah
- Bergland essentially said, "design for test" under his discussion of complexity
  - There may be a lesson here from model checking in hardware
    - Properties of a finite state space can often be checked even where there is enormous complexity
- Again, no useful measures exist

UW CSE CSE584: Software Engineering Notkin (c) 1997 12

Correctness

- Well, yeah
- Even if you "prove" modules are correct, composing the modules' behaviors to determine the system's behavior is hard

Correspondence

- "Problem-program mapping"
- The way in which the design is associated with the requirements
- The idea is that the simpler the mapping, the easier it will be to accommodate change in the design when the requirements change

Functional decomposition

- Divide-and-conquer based on functions
  - `input`
    `compute`
    `output`
- More effective in the face of stable requirements

Question

- To what degree do you consider your systems
  - as having modules?
  - as consisting of a set of files?

Physical structure

- Almost all the literature focuses on logical structures in design
- But physical structure plays a big role in practice
  - Sharing
  - Separating work assignments
  - Degradation over time
- Why so little attention paid to this?

Information hiding

- Information hiding [Parnas 1972] is perhaps the most important intellectual tool developed to support software design
- Provides the fundamental motivation for abstract data type languages
  - And thus a key idea in the object-oriented world, too
- The conceptual basis is key (IMHO)

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

UW CSE CSE584: Software Engineering Notkin (c) 1997 19

Capturing anticipated changes

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

UW CSE CSE584: Software Engineering Notkin (c) 1997 20

Claim

- We less frequently change representations than we used to
  - 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, we should think twice about anticipating that representations will change
  - This is important, since we can't simultaneously anticipate all changes

UW CSE CSE584: Software Engineering Notkin (c) 1997 21

Other anticipated changes?

- Information hiding isn't only ADTs
- Algorithmic changes
  - Monolithic to incremental algorithms
  - Improvements in algorithms
- Replacement of hardware sensors
  - Ex: better altitude sensors
- More?

UW CSE CSE584: Software Engineering Notkin (c) 1997 22

Central premise I

- We can effectively anticipate changes
  - Unanticipated changes require changes to interfaces or (more commonly) simultaneous changes to multiple modules
- How accurate is this premise?
  - We have no idea; there is essentially no research about whether anticipated changes happen (and v.v.)

UW CSE CSE584: Software Engineering Notkin (c) 1997 23

Central premise II

- Changing an implementation is the best change, since it's isolated
- This may not always be true
  - Changing an implementation may not be simple, even if localized
  - Some global changes are straightforward
    - » Mechanically or systematically
  - VanHilst and Notkin have an alternative
    - » Using parameterized classes with a deferred supertype [ISOTAS, FSE, OOPSLA]

UW CSE CSE584: Software Engineering Notkin (c) 1997 24

Central premise III

- The semantics of the module must remain unchanged when implementations are replaced
  - Specifically, the client should not care how the interface is implemented by the module
- But what captures the semantics of the module?
  - The signature of the interface? Performance? What else?

UW CSE CSE584: Software Engineering Notkin (c) 1997 25

Central premise IV

- One implementation can satisfy multiple clients
  - Different clients of the same interface that need different implementations would be counter to the principle of information hiding
    - » Clients should not care about implementations, as long as they satisfy the interface
  - Next week: Kiczales' work on open implementations

UW CSE CSE584: Software Engineering Notkin (c) 1997 26

Central premise V

- It is implied that information hiding can be recursively applied
- Is this true?
- If not, what are the consequences?

UW CSE CSE584: Software Engineering Notkin (c) 1997 27

Information hiding reprise

- It's probably the most important design technique we know
- 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

UW CSE CSE584: Software Engineering Notkin (c) 1997 28

Information Hiding and OO

- 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?

UW CSE CSE584: Software Engineering Notkin (c) 1997 29

Layering [Parnas 79]

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

UW CSE CSE584: Software Engineering Notkin (c) 1997 30

The `uses` relation

- 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?
  - Can `uses` be mechanically computed?

Notkin (c) 1997 31

`uses` vs. `invokes`

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

Notkin (c) 1997 32

Parnas’ observation

- A non-hierarchical uses relation makes subsetting difficult
  - It also makes testing difficult
  - (What about upcalls?)
- So, it is important to design the `uses` relation

Notkin (c) 1997 33

Criteria for `uses(A,B)`

- 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

Notkin (c) 1997 34

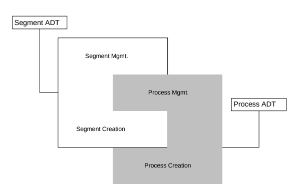
Layering in Dijkstra’s THE OS

- OK, those of you who took OS
- How was layering used, and how does it relate to this work?

Notkin (c) 1997 35

Modules and layers interact?

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



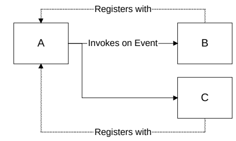
Notkin (c) 1997 36

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 cost to pay for "just" abstraction

Implicit invocation

- Components announce events that other components can choose to respond to
  - (Roughly, event-based programming)
  - The `invokes` relation is the inverse of the `names` relation



Implicit invocation mechanisms

- Field [Reiss], DEC FUSE, HP Softbench, etc.
  - Components announce events as ASCII messages
  - Components register interest using regular expressions
  - Centralized multicast message server
- Smalltalk's Model-View-Controller
  - Registering with objects
  - Separating UI views from internal models
  - May request permission to change
- Others? (COM's model?)

Not just indirection

- There is often confusion between implicit invocation and indirect invocation
  - Calling a virtual function is a good example of indirect invocation
    - » The calling function doesn't know the precise callee, but it knows it is there and that there is only one
    - » Not true in general in implicit invocation
- An announcing component should not `use` any responding components

Mediators

- One style of using implicit invocation is the use of mediators [Sullivan & Notkin]
- This approach combines events with entity-relationship designs
- The intent is to ease the development and evolution of integrated systems
  - Management the coupling and isolate behavioral relationships between components

Experience

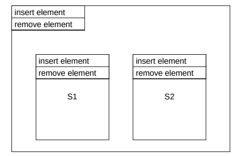
- I'll show a small (academic) example
- However, a radiation treatment planning (RTP) system (Prism) was designed and built using this technique
  - By a radiation oncologist [Kalet]
  - A third generation RTP system
  - In clinical use at UW and several other major research hospitals

Example

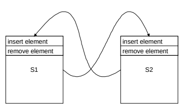
- Two set components, `S1` and `S2`
- Ensure that the sets maintain the same elements
  - Can add or delete elements from either set
- What changes might you anticipate?

ADT design

- To ensure that no client changes one set but not the other, encapsulate both in a third component
  - Promote hidden operations
- This outer component is not there for information hiding reasons



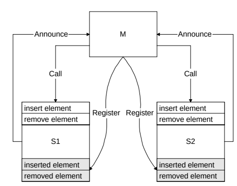
Hardwiring



- Modify the implementations of the sets
- Clients simply call functions on either `S1` or `S2`

Mediators

- Create separate component to represent relationship
- When either set changes, it announces an event
  - Events are defined in the interface, like methods
- The mediator registers with and responds to those events
  - Must avoid circularity
- Neither set knows it is part of the relationship
  - Clients see `S1` and `S2`

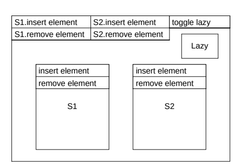


Change: lazy equivalence

- What if we later decided to maintain the equivalence of the sets lazily
  - For instance, one set might be represented in a hidden window, and there's no reason to maintain equivalence at all times

ADT design

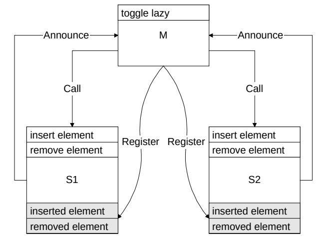
- Put the lazy bit inside the encapsulating component
- Expand the interface
- Where is the code that re-establishes the equivalence relation when lazy toggles off?
  - Requires iterator, too



Hardwired design

- Handling the lazy change with the hardwired result leads to a pretty ugly (highly coupled) design

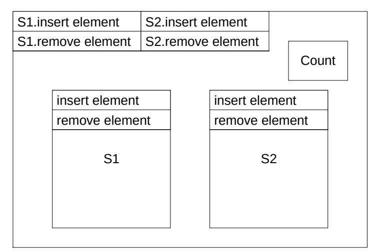
Mediator: with lazy update



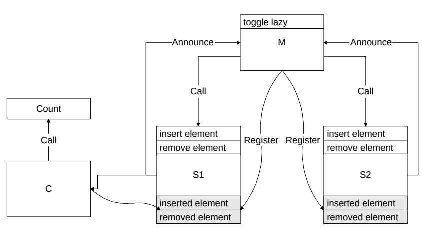
Another change: size of `S1`

- Suppose we now want to keep track of the size of one of the sets (say, `S1`)
- Should be able to query the size
  - In some variants, you can directly increment or decrement the size directly

ADT design



Mediators



Assessment

- For some classes of systems and changes, mediator-based designs seem attractive
- Lots of outstanding issues
  - Circularities in relations
  - Ordering of mediators
  - Distributed and concurrent variants
  - New component models
    - » COM, etc.