

## CSE584: Software Engineering

### Lecture 4: Design (A)

David Notkin  
Computer Science & Engineering  
University of Washington  
<http://www.cs.washington.edu/education/courses/584/>

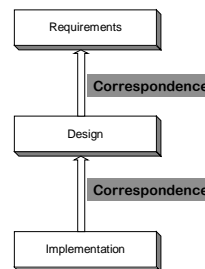
## Design

- **First lecture (tonight)**
  - Basic issues in design, including some historical background
  - Well-understood techniques
    - Information hiding, layering, event-based techniques
- **Second lecture (next week)**
- **More recent issues in design**
  - Problems with information hiding (and ways to overcome them)
  - Architecture, patterns, frameworks

## Outline

- Basic concepts
- Information hiding
- Layered systems
- Design problems you face
- Implicit invocation design

## Very High-Level View



- Requirements define the clients' view
  - What the system is supposed to do
  - Focuses on external behavior
- Design captures the developers' view
  - How the requirements are realized
  - Defines the internal structure of the solution
- But: "What" vs. "How"
- Also, reminiscent of the Brian Cantwell Smith diagram in Jackson's video

## Complexity

"Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into one... In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound." —Brooks, 1986

## Continuous & iterative

- 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.
  - About 1000 design decisions at various levels are made in producing a single page of code

## Multiple design choices

- There are multiple (perhaps unbounded) designs that satisfy (at least the functional) aspects of a given set of requirements
- How does one choose among these alternatives?
  - How does one even identify the alternatives?
  - How does one reject most bad choices quickly?
  - What criteria distinguish good choices from bad choices?

## What criteria?

- In general, there are three high level answers to this question: and, it is very difficult to answer precisely
  1. Satisfying functional and performance requirements
    - Maybe this is too obvious to include
    - Often not achieved, though: but because of design choices? Garlan: yes!
  2. Managing complexity
  3. Accommodating future change

## 1. Managing complexity

- “The technique of mastering complexity has been known since ancient times: *Divide et impera* (Divide and Rule).” —Dijkstra, 1965
- “...as soon as the programmer only needs to consider intellectually manageable programs, the alternatives he is choosing from are much, much easier to cope with.” —Dijkstra, 1972
- The complexity of the software systems we are asked to develop is increasing, yet there are basic limits upon our ability to cope with this complexity. How then do we resolve this predicament?” —Booch, 1991

## Divide and conquer

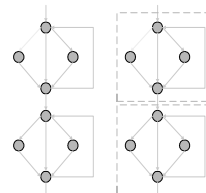
- We have to decompose large systems to be able to build them
  - The “modern” problem of composing systems from pieces is equally or more important
    - It’s not modern, though: we’ve had to compose for as long as we have decomposed
  - And closely related to decomposition in many ways
- For software, decomposition techniques are distinct from those used in physical systems
  - Fewer constraints are imposed by the material
  - Shanley principle?



## Composition

- “Divide and conquer. Separate your concerns. Yes. But sometimes the conquered tribes must be reunited under the conquering ruler, and the separated concerns must be combined to serve a single purpose.”  
—M. Jackson, 1995
- Jackson’s view of composition as printing with four-color separation
- Remember, composition in programs is not as easy as conjunction in logic

## Benefits of decomposition



- Decrease size of tasks
- Support independent testing and analysis
- Separate work assignments
- Ease understanding
- In principle, can significantly reduce paths to consider by introducing one interface

## 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; 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
  - Find a better one that satisfies more criteria
- In practice, it's easy to
  - Build something designed pretty much like the last one
  - This has benefits, too: understandability, properties of the pieces, etc.

## Structure

- The focus of most design approaches is structure
- What are the components and how are they put together?
- Behavior is important, but largely indirectly
  - Satisfying functional and performance requirements

## So what happens?

- People often buy into a particular approach or methodology
  - Ex: structured analysis and design, object-oriented design, JSD, Hatley-Pirbair, etc.
- “Beware a methodologist who is more interested in his methodology than in your problem.” —M. Jackson

## Conceptual integrity

- Brooks and others assert that conceptual integrity is a critical criterion in design
  - “It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.” —Brooks, MMM
- Such a design often makes it far easier to decide what is easy and reasonable to do as opposed to what is hard and less reasonable to do
  - This is not always what management wants to hear

## 2. Accommodating change

- “...accept the fact of change as a way of life, rather than an untoward and annoying exception.” —Brooks, 1974
- Software that does not change becomes useless over time. —Belady and Lehman
- Internet time makes the need to accommodate change even more apparent

## Anticipating change

- It is generally believed that to accommodate change one must anticipate possible changes
  - Counterpoint: Extreme Programming
- By anticipating (and perhaps prioritizing) changes, one defines additional criteria for guiding the design activity
- It is not possible to anticipate all changes

## Rationalism vs. empiricism

Brooks' 1993 talk  
"The Design of Design"

- **rationalism** — the doctrine that knowledge is acquired by reason without resort to experience [WordNet]
- **empiricism** — the doctrine that knowledge derives from experience [WordNet]

## Examples

### Life

- Aristotle vs. Galileo
- France vs. Britain
- Descartes vs. Hume
- Roman law vs. Anglo-Saxon law

### Software (Wegner)

- Prolog vs. Lisp
- Algol vs. Pascal
- Dijkstra vs. Knuth
- Proving programs vs. testing programs

## Brooks' view

- Brooks says he is a "thoroughgoing, died-in-the-wool empiricist."
- "Our designs are so complex there is no hope of getting them right first time by pure thought. To expect to is arrogant."
- "So, we must adopt design-build processes that incorporate evolutionary growth ...
  - "Iteration, and restart if necessary"
  - "Early prototyping and testing with *real users*"
- Maybe this is more an issue of requirements and specification, but I think it applies to design, too
  - "Plan to throw one away, you will anyway."

## Properties of design

- Cohesion
- Coupling
- Complexity
- Correctness
- Correspondence
  
- Makes designs "better", one presumes
- Worth paying attention to

## 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" (in a future paper we'll read)

## Coupling

- Strength of interconnection between modules
- Hierarchies are touted as a wonderful coupling structure, limiting interconnections
  - But don't forget about composition, which requires some kind of coupling
- Coupling also degrades over time
  - "I just need one function from that module..."
  - Low coupling vs. no coupling

## Unnecessary coupling hurts

- Propagates effects of changes more widely
- Harder to understand interfaces (interactions)
- Harder to understand the design
- Complicates managerial tasks
- Complicates or precludes reuse

## 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

## Complexity

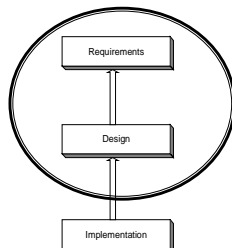
- Well, yeah, simpler designs are better, all else being equal
- But, again, no useful measures of design/program complexity exist
  - Although there are dozens of such measures
  - My understanding is that, to the first order, most of these measures are linearly related to “lines of code”

## Correctness

- Well, yeah
- Even if you “prove” modules are correct, composing the modules' behaviors to determine the system's behavior is hard
- Leveson and others have shown clearly that a system can fail even when each of the pieces work properly
  - Many systems have “emergent” properties

## 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
- M. Jackson: problem frames
  - In the style of Polya



## Functional decomposition

- Divide-and-conquer based on functions

```
input;
compute;
output
```
- Then proceed to decompose compute
- This is stepwise refinement (Wirth, 1971)
- 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

## Question

- To what degree do you consider your systems
  - as having modules?
  - as consisting of a set of files?
- This is a question of physical vs. logical structure of programs
  - In some languages/environments, they are one and the same
  - Ex: Smalltalk-80

## 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 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

## 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
  - Ex: Changing the representation of null-terminated strings in Unix systems wouldn't be sensible
    - And this doesn't represent a stupid design decision

## Other anticipated changes?

- Information hiding isn't only ADTs
- Algorithmic changes
  - (These are almost always part and parcel of ADT-based decompositions)
  - Monolithic to incremental algorithms
  - Improvements in algorithms
- Replacement of hardware sensors
  - Ex: better altitude sensors
- More?

## Ubiquitous computing domain

- Portolano is a UW CSE project on this topic
  - Devices everywhere, handhelds, on-body devices, automated laboratories, etc.
- The set of anticipated changes is significantly different than in many other domains
  - Data is more stable than computations
  - Must accommodate diversity in communication speed, reliability, etc.
- Interesting domain for information hiding

## 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
  - Nor do we have disciplined ways to figure out how to better anticipate changes

## The A-7 Project

- In the late 1970's, Parnas led a project to redesign the software for the A-7 flight program
  - One key aspect was the use of information hiding
- The project had successes, including a much improved specification of the system and the definition of the SCR requirements language
- But little data about actual changes was gathered

## Central premise II

- Changing an implementation is the best change, since it's isolated
- This may not always be true
  - Changing a local implementation may not be easy
  - Some global changes are straightforward
    - Mechanically or systematically
  - VanHilst's work showed an alternative
    - Using parameterized classes with a deferred supertype [ISOTAS, FSE, OOPSLA]
  - Griswold's work on information transparency

## 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?

### 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
  - Kiczales' work on open implementations

### Central premise V

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

### Information hiding reprise

- 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)

### 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?

### 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

### 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?

## uses vs. invokes

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

## Criteria for `uses` (A, B)

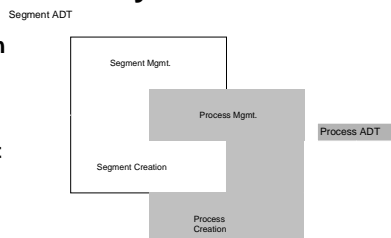
- A is essentially simpler because it uses B
- B is not substantially more complex because it does not use A
- There is a useful subset containing B but not A
- There is no useful subset containing A but not B

## Layering in THE (Dijkstra's layered OS)

- OK, those of you who took OS
- How was layering used, and how does it relate to this work?
- (For thinking about off-line, or for email discussion)

## Modules and layers interact?

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

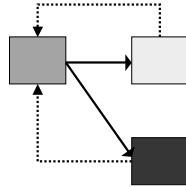


## Language support

- 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

## Implicit invocation

- Components announce events that other components can choose to respond to
- Yellow and red register interest in an event from blue
  - When blue announces that event, yellow and red are invoked
- In implicit invocation, the *invokes* relation is the inverse of the *names* relation
- Invocation does not require ability to name



## Old II 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

## New II mechanisms: or extensive uses of them

- JDK's
  - Different versions have somewhat different event models
- Java beans, Swing, ...
- CORBA and COM

## Objective

- **Most, if not all, of you are at least comfortable with using events**
  - Probably primarily in the context of existing components and frameworks
- **So, what's the issue to cover?**
- **Several**
  - Thinking of implicit invocation as more than "just" events
  - Identifying some concrete software engineering reasons to use it
  - Identifying some limitations

## Not just indirection

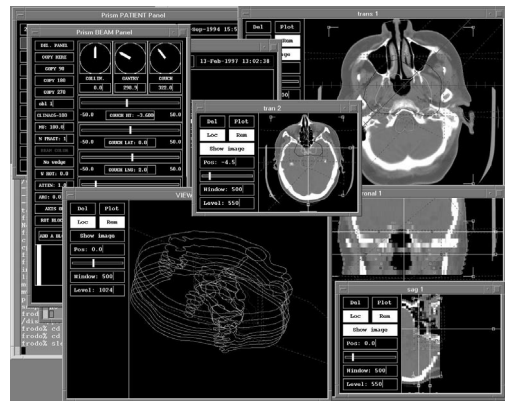
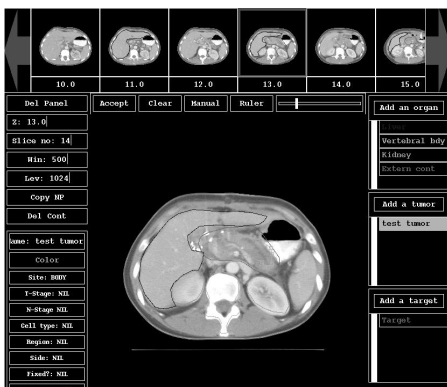
- **There is often confusion between implicit invocation and indirect invocation**
  - Calling a virtual function is a good example of indirect invocation
    - The calling function doesn't know the precise callee, but it knows it is there and that there is only one
    - Not true in general in implicit invocation
- **An announcing component should not use (in the Parnas sense) any responding components**
  - This is extremely difficult to define precisely
  - Roughly, the postcondition of the announcing component should not depend on any computation of the implicitly invoked 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**
  - Manage the coupling and isolate behavioral relationships between components

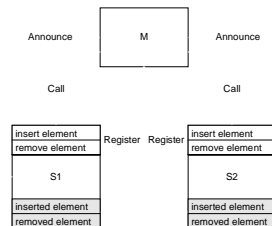
## Experience

- 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
  - <http://www.radonc.washington.edu/physics/prism/>
  - See the screenshots on next slides



## Example from paper

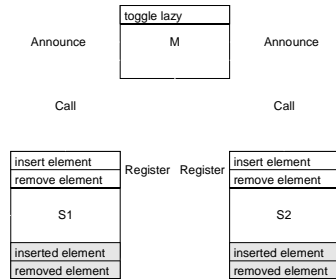
- Two set components, S1 and S2
- Ensure that the sets maintain the same elements
  - Can add or delete elements from either set
- On right is the mediator approach
- ADT and hardwired alternatives discussed in paper



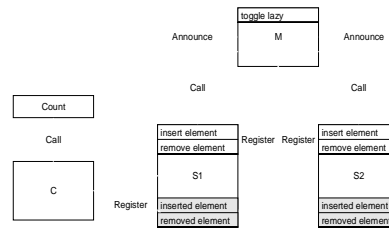
## Mediator issues

- Must avoid circularity
- Events are first-class elements in interfaces
  - “interface” and “outface”
- Makes many changes easier
  - lazy equivalence
  - allow size of the sets to be changed directly
  - ...

## Mediator: with lazy update



## Mediators: lazy and count



## 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
  - Reasoning (even informally) about systems built with implicit invocation
    - Even “just” debugging

## Design I wrap-up

- High-level issues in design
  - Managing complexity, accommodating change, conceptual integrity
- Information hiding
- Layering
- Implicit invocation
  - Mediator-based design
- Problems you face

## Next week

- Open implementation
  - Aspect-oriented programming
- Software architecture
- Patterns
- Frameworks
- A bit on composition of systems