Design is not just what it looks like and feels like.
Design is how it works. –Steve Jobs

CSE403: Software Engineering

David Notkin
Winter 2009

Design: word association

- Shout out any phrases, tools, approaches, issues, etc. that you think of when I say "software design"

Design topics

- Basic issues in design, including historical background
- Well-understood techniques
  - Information hiding, layering, event-based techniques
  - …
- Neo-modern issues
  - Problems with information hiding
  - Aspect-oriented design
  - Architecture, patterns, frameworks
  - …

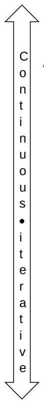
Properties of software design

- Complexity
- Multi-level, continuous, iterative
- Broad potential solution space (in most cases)
- Relatively unclear criteria for selecting solution

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?
  - What does it mean to have conceptual integrity?
- 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
  - Satisfying functional and performance requirements
  - Managing complexity
  - Accommodating future change
- Well, also reliability, safety, understandability, compatibility, robustness, …

Managing complexity

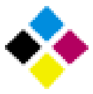
- The technique of mastering complexity has been known since ancient times: Divide et impera (Divide and Rule). —Dijkstra, 1965
- …as soon as the programmer only needs to consider intellectually manageable programs, the alternatives he is choosing from are much, much easier to cope with. —Dijkstra, 1972
- 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

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 an interface

Which decomposition?

- How do we select a decomposition?
  - We determine the desired criteria
  - We select a decomposition (design) that will achieve those criteria
  - Question: whether do the potential decompositions even come from?
- 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

Alan Perlis quotations: aside

- If you have a procedure with 10 parameters, you probably missed some.
- One man's constant is another man's variable.
- There are two ways to write error-free programs; only the third one works.
- When someone says "I want a programming language in which I need only say what I wish done," give him a lollipop.
- Simplicity does not precede complexity, but follows it.

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

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

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"

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

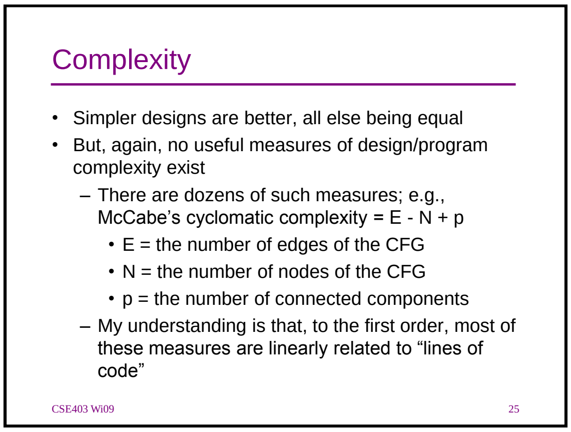
Unnecessary coupling hurts

- Propagates effects of changes more widely
- Harder to understand interfaces (interactions)
- Harder to understand the design
- Complicates managerial tasks
- Complicates or precludes reuse
- May lead to "clone and own"

It's easy to...

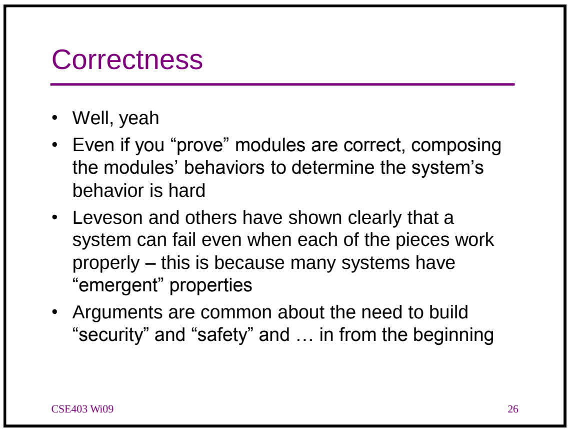
- ...reduce coupling by calling a system a single module
- …increase cohesion by calling a system a single module

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



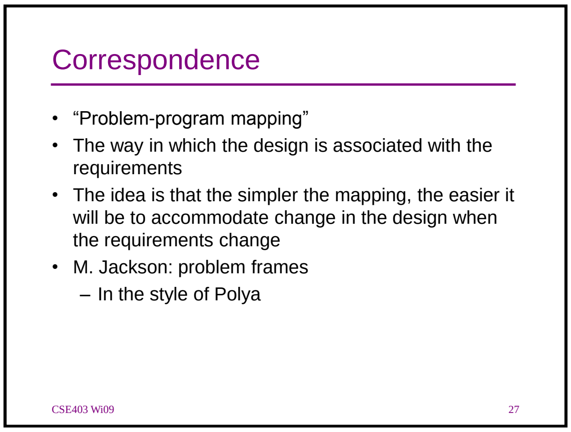
Complexity

- Simpler designs are better, all else being equal
- But, again, no useful measures of design/program complexity exist
  - There are dozens of such measures; e.g., McCabe's cyclomatic complexity = E - N + p
    - E = the number of edges of the CFG
    - N = the number of nodes of the CFG
    - p = the number of connected components
  - My understanding is that, to the first order, most of these measures are linearly related to "lines of code"



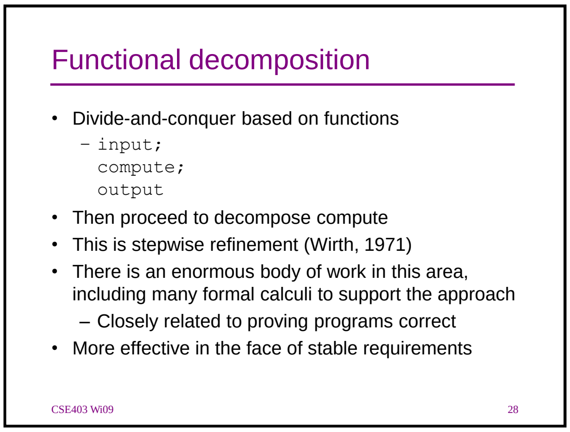
Correctness

- 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 – this is because many systems have "emergent" properties
- Arguments are common about the need to build "security" and "safety" and … in from the beginning



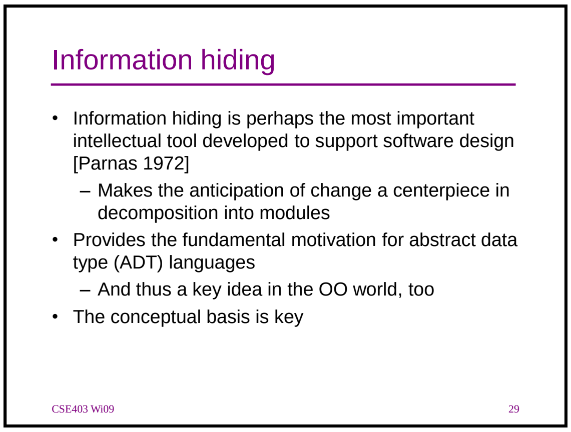
Correspondence

- "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



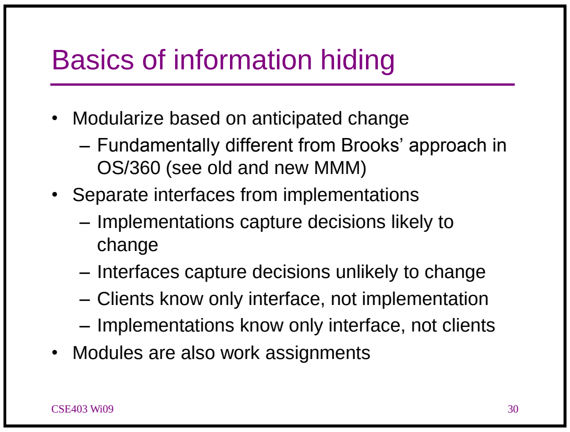
Functional decomposition

- 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



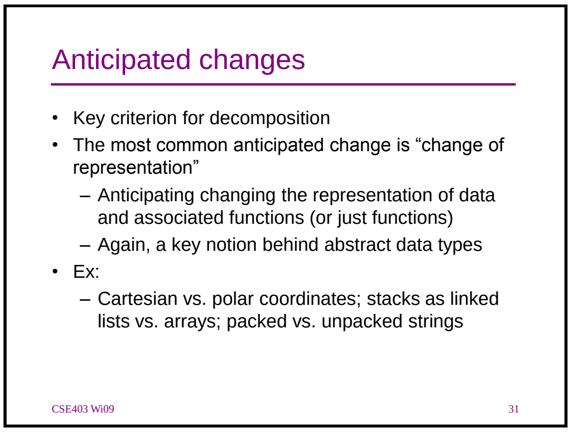
Information hiding

- 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

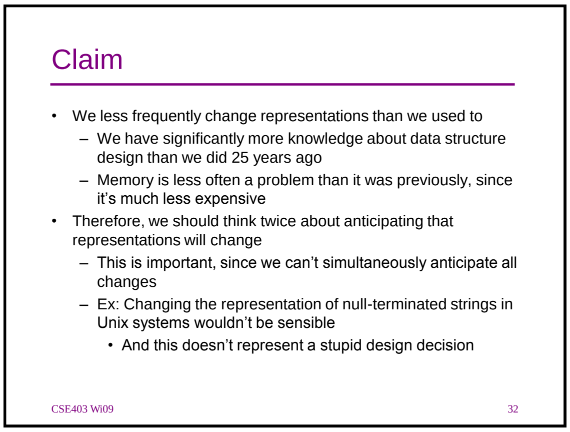
- Modularize based on anticipated change
  - Fundamentally different from Brooks' approach in OS/360 (see old and new MMM)
- Separate interfaces from implementations
  - Implementations capture decisions likely to change
  - Interfaces capture decisions unlikely to change
  - Clients know only interface,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



Anticipated changes

- Key criterion for decomposition
- 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

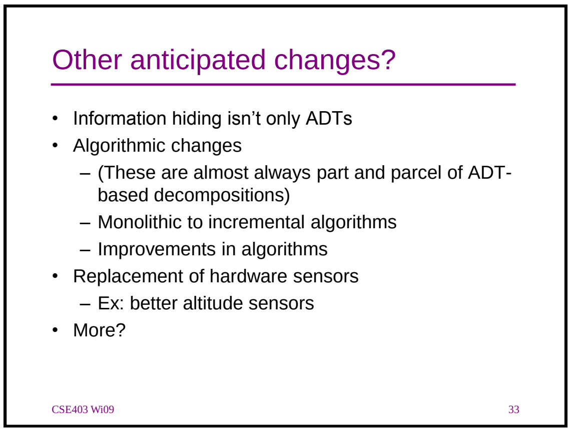
CSE403 Wi09 31



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

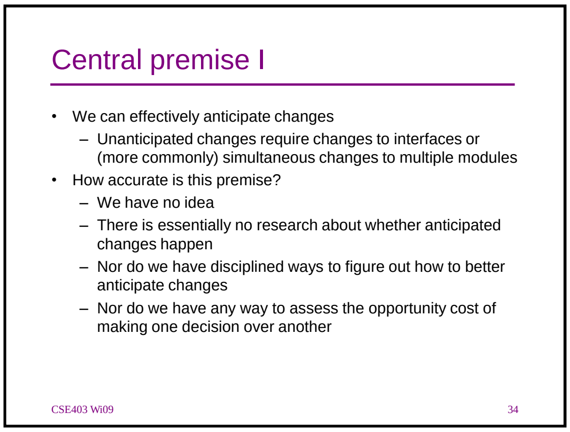
CSE403 Wi09 32



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?

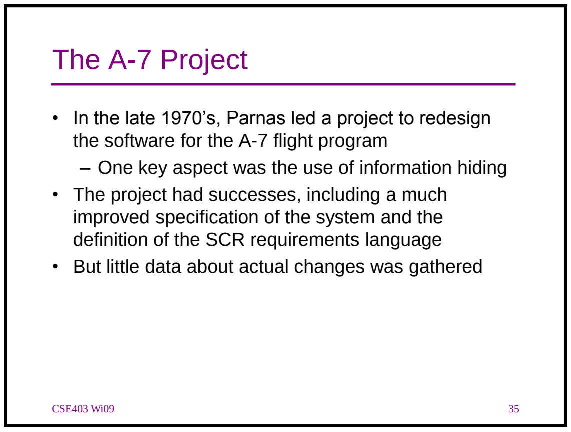
CSE403 Wi09 33



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
  - Nor do we have any way to assess the opportunity cost of making one decision over another

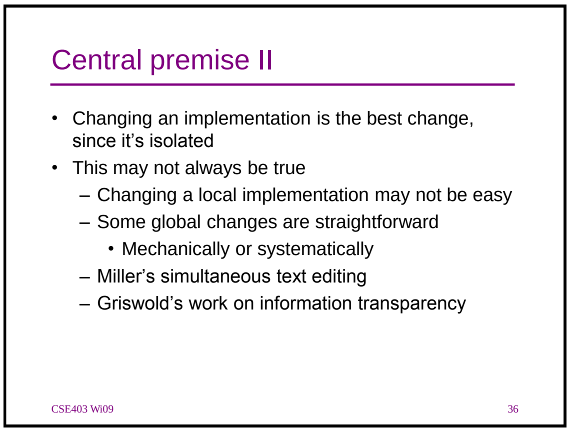
CSE403 Wi09 34



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

CSE403 Wi09 35



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
  - Miller’s simultaneous text editing
  - Griswold’s work on information transparency

CSE403 Wi09 36

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

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)

Questions?