CSE503: Software Engineering

David Notkin
University of Washington
Department of Computer Science & Engineering
Winter 2002

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

Today

- Some general background on design
  - Decomposition, composition
  - Managing complexity, anticipating change
  - Etc.
- Monday
  - Information hiding
- Wednesday
  - Layering/ "uses" relation

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
  2. Managing complexity
  3. Accommodating future change

7

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

8

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?

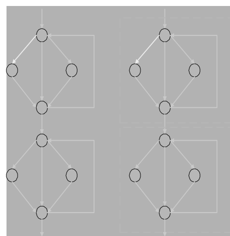
9

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
- Composition in programs is not as easy as conjunction in logic

10

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

11

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.

12

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

13

So what happens?

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

14

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

15

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

16

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

17

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]

18

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

19

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."

20

Properties of design

- Cohesion
- Coupling
- Complexity
- Correctness
- Correspondence

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

21

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

22

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

23

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

24

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

25

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"

26

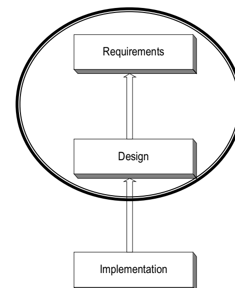
Correctness

- 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

27

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



28

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

29

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

30

Aside: 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?

31

Wrap-up

- High-level, but needed for basic discussion on design
- Monday: read the Parnas information hiding paper

32