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

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

What criteria?

- But how do we make design choices (at any level)?
- 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
  - 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
- For software, decomposition techniques are distinct from those used in physical systems
  - Fewer constraints are imposed by the material

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.

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.”
**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
KWIC: “hello world” of information hiding

“The KWIC index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a list of all circular shifts of all lines in alphabetical order.”

- In groups, sketch a module-level design for KWIC – down to the level of modules and methods

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?

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

Unchanged semantics

- 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?
Questions?