Major results in software design: an historical overview

- Managing complexity
- Stepwise refinement and top-down design
  - Relatively brief tangent: proofs of correctness
- Coupling, cohesion
- Information hiding
- Layering

Managing complexity: Dijkstra

- The competent programmer is fully aware of the limited size of his own skull.
- Software is so complex that our poor head cannot cope with it at all. Therefore, we have to use all possible means and methods to try to control this complexity.
- The technique of mastering complexity has been known since ancient times: Divide et impera (Divide and Rule).
- …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.

Managing complexity: others

- Brooks
  - 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.
- Booch
  - 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?
- Perlis
  - If you have a procedure with 10 parameters, you probably missed some.
  - There are two ways to write error-free programs; only the third one works.
  - Simplicity does not precede complexity, but follows it.

Stepwise refinement and top-down design [Dijkstra, Wirth, Hoare, et al.]

- Pseudo-code is repeatedly expanded until the translation into programming language code is obvious
- Define top-level module
  - Choose a module to be decomposed
  - Use stepwise refinement to decompose into smaller modules
  - Repeat until all modules are easily understood
  - Use stepwise refinement to complete low-level modules

```
• while not sorted do
      find inversion;
      swap
  end


• lex;
  parse;
  check semantics;
  generate code
```

Structured programming

- Dijkstra's concerns about the `goto` statement were based largely on the notion that the static (syntactic) structure of the program and the dynamic (execution) structure of the program were wildly different, making understanding and reasoning difficult
- An oft-cited theorem, due to Böhm and Jacopini, is that the `goto` statement is not needed – all programs can be written using sequencing, conditionals, and loops as the control structures
  - The proof is constructive, converting arbitrary flow charts to structured flow charts
- However, the proof does not genuinely address Dijkstra's concern

Basics of program correctness

- Very closely related to stepwise refinement: make precise the meaning of programs
- In a logic, write down (this is often called the specification)
  - the effect of the computation that the program is required to perform (the postcondition `Q`)
  - any constraints on the input environment to allow this computation (the precondition `P`)
- Associate precise (logical) meaning to each construct in the programming language (this is done per-language, not per-program)
- Reason (usually backwards) that the logical conditions are satisfied by the program `S`
- A Hoare triple is a predicate `{P}S{Q}` that is true whenever `P` holds and the execution of `S` guarantees that `Q` holds

Examples

```
• {true}
    y := x * x;
  {y >= 0}

• {x <> 0}
    y := x * x;
  {y > 0}

• {x > 0}
    x := x + 1;
  {x > 1}
```

More examples

-
  ```
  {x = k}
    if (x < 0) x := -x endif;
  {    ?    }
  ```

-
  ```
  {    ?    }
    x := 3;
  { x = 8 }
  ```

Strongest postconditions
[example from Aldrich and perhaps from Leino]

The following are all valid Hoare triples

- `{x = 5} x := x * 2 { true }`
- `{x = 5} x := x * 2 { x > 0 }`
- `{x = 5} x := x * 2 { x = 10 || x = 5 }`
- `{x = 5} x := x * 2 { x = 10 }`

- Which is the most useful, interesting, valuable? Why?

Weakest preconditions
[example from Aldrich and perhaps from Leino]

Here are a number of valid Hoare Triples

- `{x = 5 && y = 10} z := x / y { z < 1 }`
- `{x < y && y > 0} z := x / y { z < 1 }`
- `{y ≠ 0 && x / y < 1} z := x / y { z < 1 }`

- The last one is the most useful because it allows us to invoke the program in the most general condition
- It is called the *weakest precondition,* `wp(S,Q)` of `S` with respect to `Q`
  - If `{P} S {Q}` and for all `P'` such that `P' => P`, then `P` is `wp(S,Q)`

Sequential execution

- What if there are multiple statements
  - `{P} S1;S2 {Q}`
- We create an intermediate assertion
  - `{P} S1 {A} S2 {Q}`
- We reason (usually) backwards to prove the Hoare triples
- A formalization of this approach essential defines the ; operator in most programming languages

-
  ```
  {x > 0}
    y := x*2;
    z := y/2
  {z > 0}
  ```

-
  ```
  {x > 0}
    y := x*2;
  {y > 0}
    z := y/2
  {z > 0}
  ```

Conditional execution

- `{P}`
```
   if C then S1
         else S2
   endif
 {Q}
```
- Must consider both branches
- Ex: compute the maximum of two variables **x** and **y**

```
{true}
  if x >= y then
    max := x
  else
    max := y
  fi
{(max >= x ∧ max >= y)}
```

Hoare logic rule: conditional

```
{P} if C then S1 else S2 {Q}
  ≡
{P ∧ C}S1{Q} ∧ {P ∧ ¬ C}S2{Q}
```

Be careful!

- `{true}`
```
   max := abs(x)+abs(y);
 {max >= x ∧ max >= y}
```
- This predicate holds, but we don't "want" it to
  - The postcondition is written in a way that permits satisfying programs that don't compute the maximum
  - In essence, every specification is satisfied by an infinite number of programs and vice versa
- The "right" postcondition is
  - `{(max = x ∨ max = y)`
    `∧ (max >= x ∧ max >= y)}`

Assignment statements

- We've been highly informal in dealing with assignment statements
- What does the statement **`x := E`** mean?
  - `{Q(E)} x := E {Q(x)}`
  - If we knew something to be true about **E** before the assignment, then we know it to be true about **x** after the assignment (assuming no side-effects)

Examples

```
{y > 0}
  x := y
{x > 0}


{x > 0}    [Q(E) ≡ x + 1 > 1 ≡ x > 0 ]
  x := x + 1;
{x > 1}    [Q(x) ≡ x > 1]
```

More examples

```
{    ?    }
  x := y + 5
{x > 0}

{x = A ∧ y = B }
  t := x;
  x := y;
  y := t
{x = B ∧ y = A }
```

Loops

- `{P} while B do S {Q}`
- We can try to unroll this into
  - `{P ∧ ¬ B} S {Q} ∨`
    `{P ∧ B} S {Q ∧ ¬B} ∨`
    `{P ∧ B} S {Q ∧ B} S {Q ∧ ¬B} ∨ …`
- But we don't know how far to unroll, since we don't know how many times the loop can execute
- The most common approach to this is to find a loop invariant, which is a predicate that
  - is true each time the loop head is reached (on entry and after each iteration)
  - and helps us prove the postcondition of the loop
  - It approximates the fixed point of the loop

Loop invariant for `{P} while B do S {Q}`

- Find `I` such that
  - `P ⇒ I` -Invariant is correct on entry
  - `{B ∧ I} S {I}` -Invariant is maintained
  - `{¬B ∧ I} ⇒ Q` -Loop termination proves Q
- Example

```
{n > 0}
  x := a[1];
  i := 2;
  while i <= n do
    if a[i] > x then x := a[i];
    i := i + 1;
  end;
{x = max(a[1],…,a[n])
```

Termination

- Proofs with loop invariants do not guarantee that the loop terminates, only that it does produce the proper postcondition **if** it terminates – this is called *weak correctness*
- A Hoare triple for which termination has been proven is *strongly correct*
- Proofs of termination are usually performed separately from proofs of correctness, and they are usually performed through well-founded sets
  - In this example it's easy, since `i` is bounded by `n`, and `i` increases at each iteration
- Historically, the interest has been in proving that a program does terminate: but many important programs now are intended not to terminate

Correctness of data structures

- Primarily due to Hoare; figures from Wulf *et al*.
- Prove the specifications on the abstract operations (e.g., $\texttt{Push}_a$)
- Prove the specifications on the concrete operations (e.g., $\texttt{Push}_c$)
- Prove the relation between abstract and concrete operations (e.g., `R`), the representation mapping

$$\begin{tikzcd}
\langle x_1,x_2\rangle \arrow[r, bend left, "\text{Push}_a(S,x)"] & \langle x,x_1,x_2\rangle \\
\begin{array}{c}S.sp = 2\\ S.v =\\ {[x_2,x_1,?,?,\ldots]}\end{array} \arrow[u, bend left, "R"] \arrow[r, bend left, "\text{Push}_c(S,x)"] & \begin{array}{c}S.sp = 3\\ S.v =\\ {[x_2,x_1,x,?,\ldots]}\end{array} \arrow[u, bend left, "R"]
\end{tikzcd}$$

Example

```
{¬full(S_a)}           {¬full(R(S_c))}
  Push_a(S_a,x)          Push_c(S_c,x)
{S_a=<x>||S'_a}        {R(S_c) = <x>||
                                R(S'_c)}
```

So what?

- It lays a foundation for
  - Thinking about programs more precisely
  - Applying techniques like these in limited, critical situations
  - Development of some modern design, specification and analysis approaches that seem to have value in more situations

Slight aside: 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

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

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

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

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

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?

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

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
  - Miller’s simultaneous text editing
  - 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)`

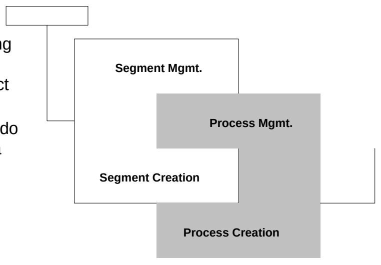
- `A` is essentially simpler because it uses `B`
- `B` is not substantially more complex because it does not use `A`
- There is a useful subset containing `B` but not `A`
- There is no useful subset containing `A` but not `B`

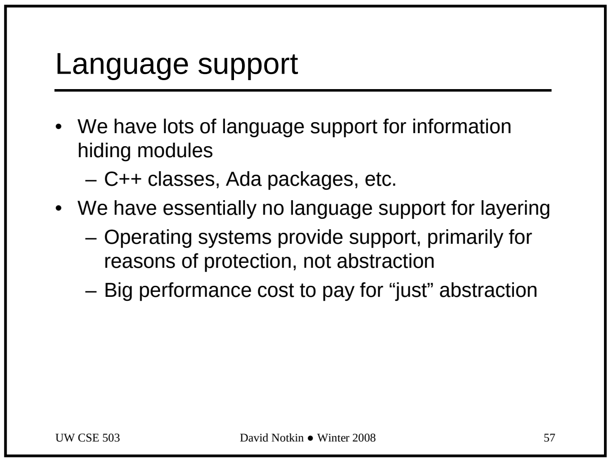
Layering in THE
(Dijkstra's layered OS)

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

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