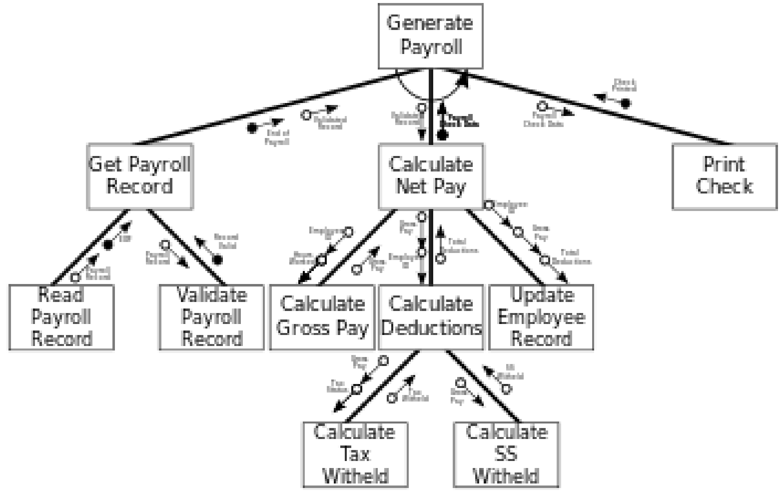


Structured Design for Assignment 2

*CIS*2750*

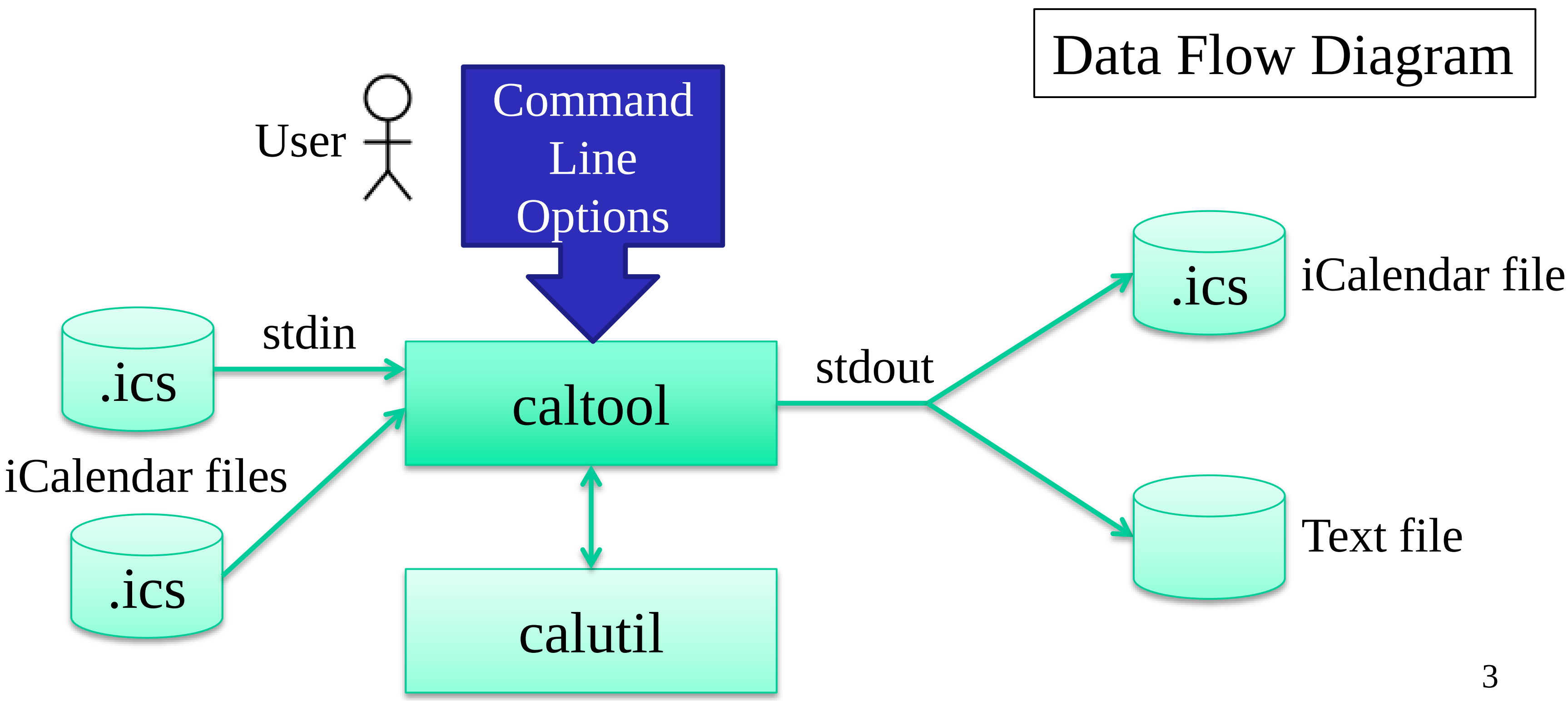
Software Development as a Profession

Outline of Lecture

1. Introduction to Assignment 2
2. Design Models
3. Caltool Design

A Command Line Application

- A2 builds on A1's calutil utilities



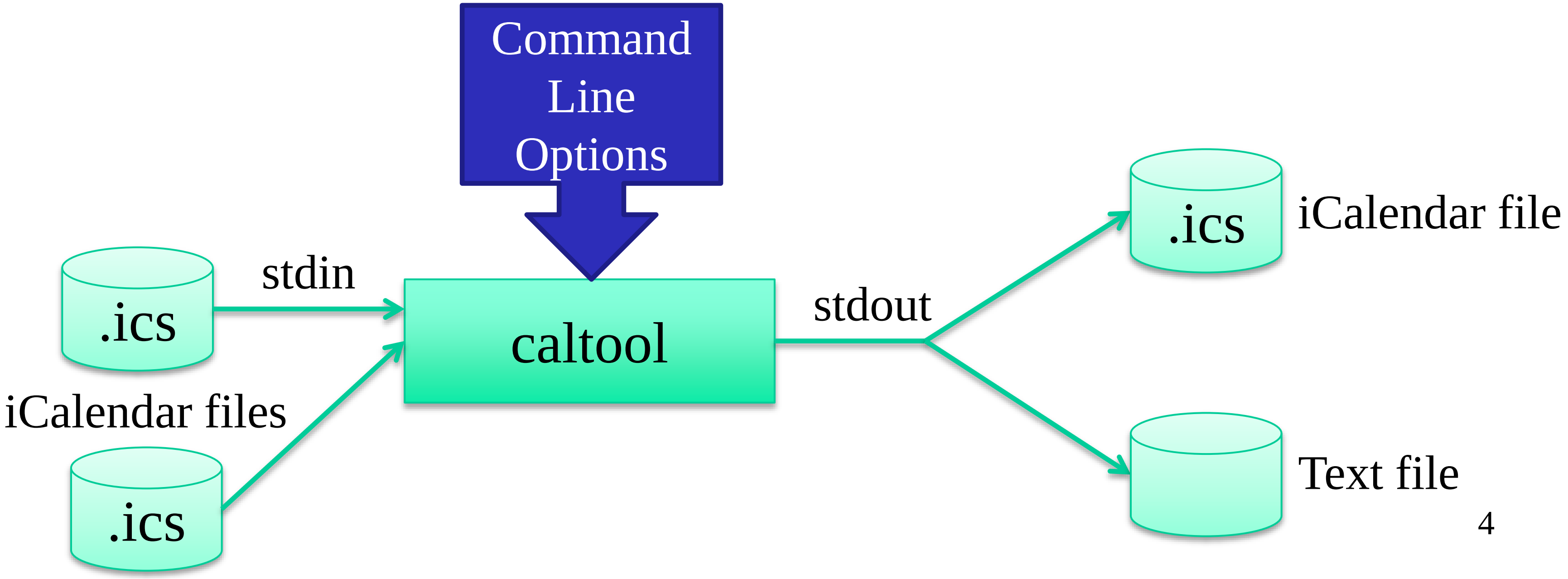
4 Command Options

-info prints statistics

-extract *kind* extracts kinds of data

-filter *content* [**from** *date*] [**to** *date*]

-combine *icsfile* combines two files



Take Advantage of Shell Features

- **Pipe:** | routes stdout of left to stdin of right

  `foo | bar`

- **File redirection:**

  > captures stdout of left into file

  < reads from file into stdin of left

  `foo < infile > outfile`

- You do not program these! They are already supplied by bash, sh, csh, and other shells.

Examples of Command Usage

- cat A.ics | **caltool *–option*** > B.ics
- **caltool *–option*** < A.ics > B.ics
- **caltool *–option*** < A.ics | **caltool –info**
- **caltool *–option*** < A.ics | tee B.ics | **caltool -info**

Smart Design

- How do we design a SW program?
- Structured design for caltool
- Principles of modularity
- From design model to code

How do we design a SW program?

- *Concept:* Build a **design model** first!
  - Can critique, refine till satisfied
  - Compare architectural drawing of building, sketch of sculpture
  - Use design model as basis for coding/testing

Tool for Building Design Model

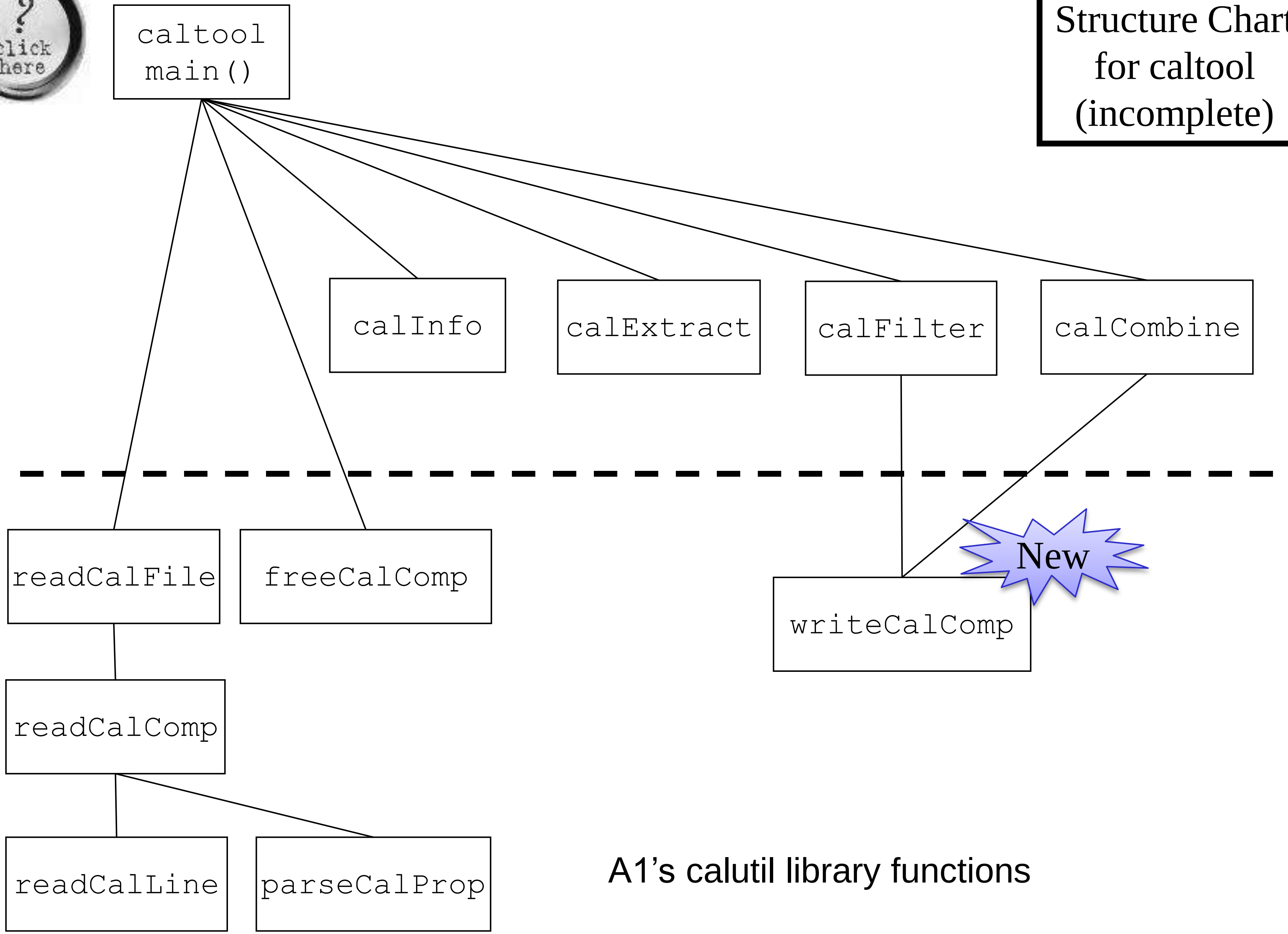
- **Structure chart**
  – Represents *static* structure of SW program in its modular form
- 3 essential features of **notation**

1. Box = module (one C function)

If module A calls module B…

2. Must be line from A to B
3. A must be higher (on the page) than B

Structure Chart for caltool (incomplete)



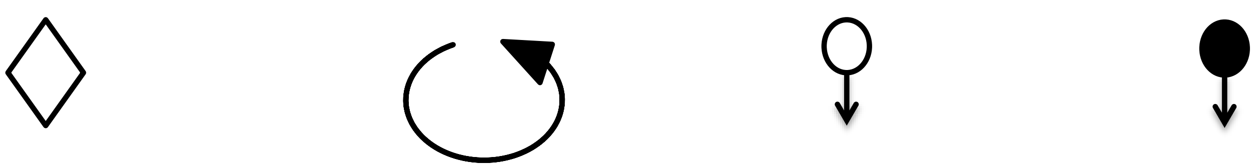
Structure Chart

- Should be able to "reverse engineer" existing code to draw a structure chart *or* draw a chart from verbal description of program
- Benefits
  - Use for design review (easy to visualize/discuss proposed SW architecture)
  - Easier/cheaper to change chart than code!
  - Helps find problems at design stage

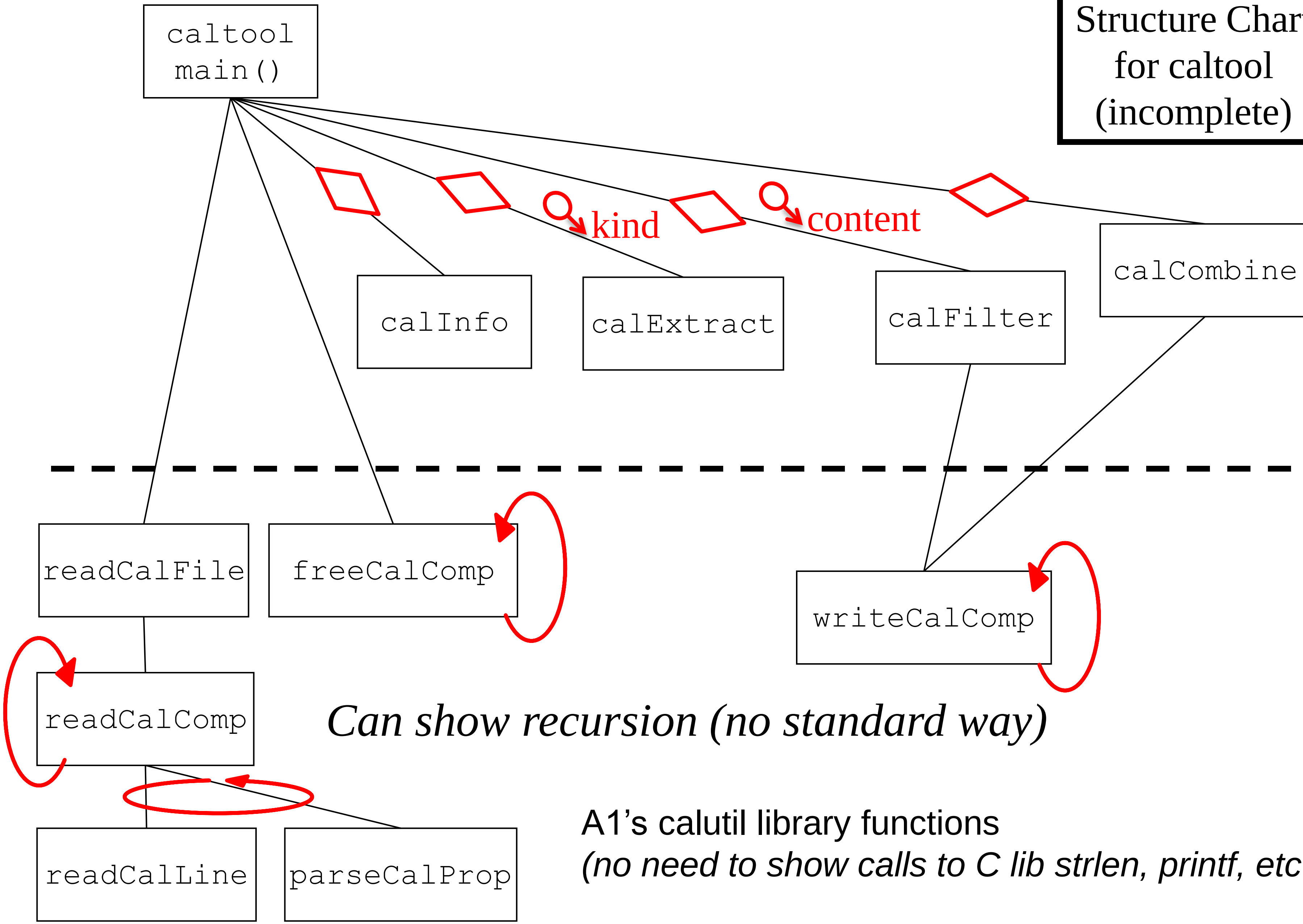
Reading caltool's Structure Chart

- "Top down"
  - Start with main()
  - Then major functions (corresponding to command options)
  - Helper functions
  - Finally utilities
- Optional "decorations"

Conditional Iteration Argument Flag/switch



Structure Chart for caltool (incomplete)



Can show recursion (no standard way)

A1’s calutil library functions
(no need to show calls to C lib strlen, printf, etc.)

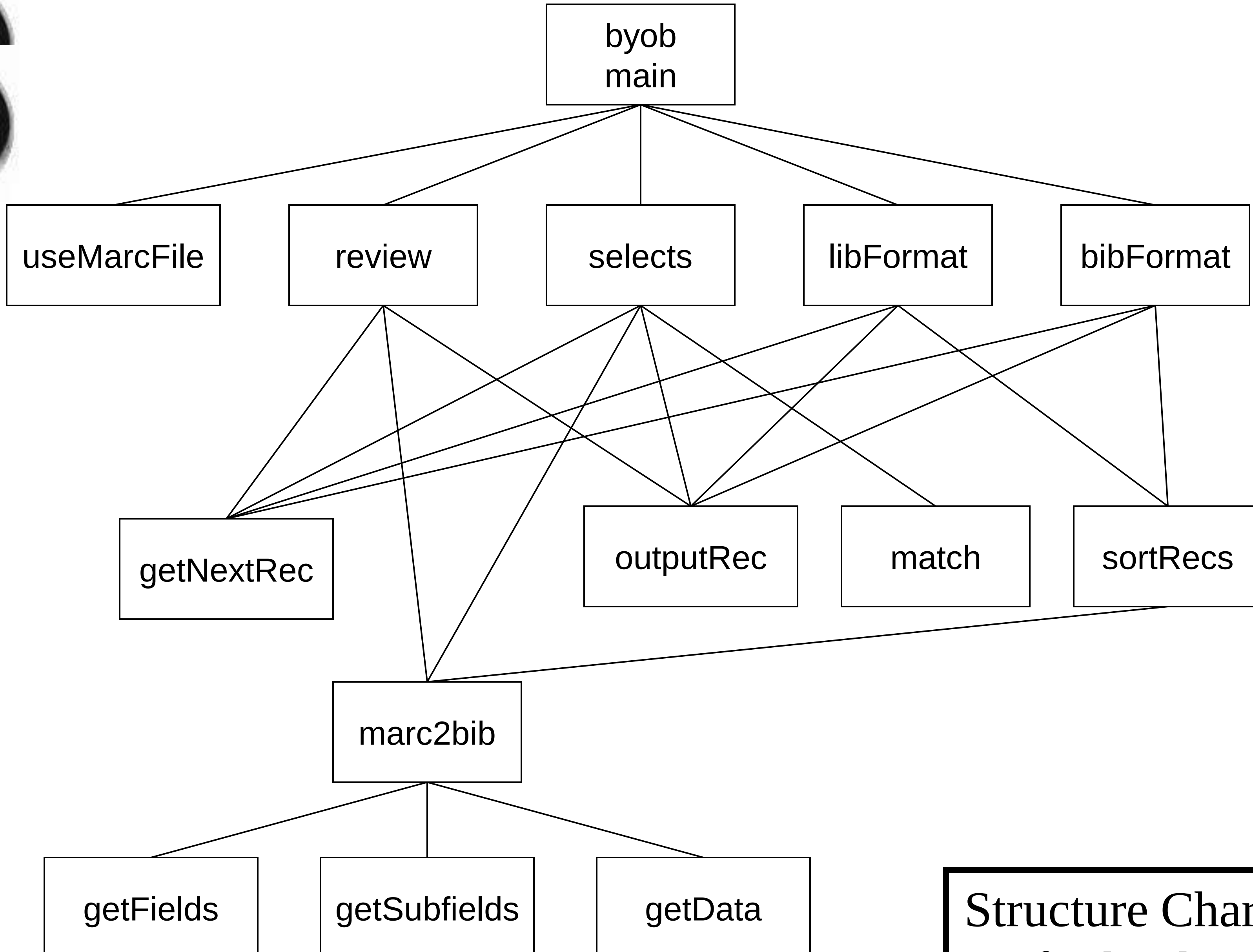
Properties of Modularity

How do we decide what to put in each module? How much "stuff" is enough? Two classic properties, cohesion & coupling*:*

- **Cohesion** should be high (not low)
  - Module should be dedicated to single easy-to-describe purpose
- Modules with high reusability *may* have high **fanin** (no. callers); compare **fanout** (no. callees)

Properties of Modularity

- **Coupling** should be low (loose, weak) not high (tight, strong)
  - Degree of data sharing between modules
  - Ideally, just parameter passing!
  - No global variables!
  - “Sequential coupling” can be tolerable
    - Where modules expect to be called in certain order
    - E.g., readCalLine() has “reset mode” due to internal state; also readCalComp() due to call depth variable



Structure Chart
for byob

Properties of Modularity

- Paying attention to cohesion & coupling yields modules easier to **understand** and **maintain**
  - Changes unlikely to have unforeseen side effects (in remote parts of program)
  - If you find a common operation occurring in >1 modules, can break out new helper function (called “refactoring”)

From Design Model to Code

- Read off the structure chart
  - Each box on chart ➔ 1 module (C function)
  - Also need spec for **interface** (parameter list, prototype, pre/postconditions) and **pseudocode**
  - Implement top-down (using stubs) or bottom-up (using drivers)
- **Structured analysis** = method for going from problem description to structure chart