The paper compares two methods of breaking a program into components: the first is to divide the program into steps (or rather, a flowchart) and the second is modularize related behaviors [1]. These strategies roughly correspond to imperative and functional programming.[1] The argument is that the second method—of encapsulating design decisions and presenting abstract interfaces—is better from an engineering standpoint and sufficently motivating to inspire work on efficient compilation.

Strengths

- Identifies and clearly states an important problem: how should we organize large software projects?
- The *Changeability* section raised interesting points. Design decisions 2, 4, and 5 are still relevant (time/space tradeoffs).

Weaknesses

- The worked example is nice, but I would have liked more anecdotes from larger projects.
- The paper does not deeply consider modularizing across a language, libraries, and user applications. (Looking back, it's obvious that the programming language should provide abstractions for strings, character sets, and I/O. User applications should *never* provide these things. At the very least, they belong in a library outside the "system".)

It's hard to imagine taking over a week to write a KWIC index. Also, why is "KWIC index" a keyword for the paper?

References

[1] D.L. Parnas. On the criteria to be used in decomposing systems into modules. In *Programming Techniques*, 1972.

[1] There are hints of object-oriented programming, but the core idea is just of a module with a clear API of functions.