

## Class 06: Hoare's Examples, Revisited

**Held:** Friday, February 2, 2007

**Summary:** Today we continue to consider the code examples from Hoare and the points he intended to make with them. We also consider other global design issues.

### Related Pages:

- EBoard.
- Reading: C.A.R. Hoare - Hints on Programming Language Design.

### Due

- Homework 2: Permutation with Repetitions.

### Assignments

- Homework 3: Removing Recursion.

### Notes:

- I do not have Wednesday's reading yet. Sorry.

### Overview:

- Examples from Hoare.
- Choosing a language.
- Key design criteria.
- More notes from the readings.

## Examples from Hoare

- What's the point of the Fortran example?
- How does his case statement seem to work.
- What point does he make in comparing  $x := y$  and  $x := y+1$ , where  $x$  is a reference variable?
- What's wrong with `multiply(A, A, A)` (assume C)?

## Criteria for Evaluating Languages

- Suppose you needed to decide whether a language is appropriate for some task. What are some of the issues that you might consider? Many people look at four basic criteria.
- **Readability:** How easy is it to read (and comprehend) a program (or portion thereof) written in the language?

- **Writability:** How easy is it to write programs (or particular kinds of programs) in the language?
- **Reliability:** How easy is it to write robust programs? Does the language help prevent errors?
- **Cost:** How expensive is it to develop, use, and maintain programs written in the language?

## Additional Criteria

- There are also a number of other, “more primitive” criteria that help contribute to the four basics.
- *Simplicity.* A language should not have too many features nor too many ways to express the same concept.
  - However, too much simplicity can lead to too much complication if the language has no direct support for something the programmer wants to do.
- *Orthogonality and Uniformity.* Primitive operations can be combined freely to provide more powerful operations. The same operation can always be used in the same when. Also a lack of "special cases".
- It should be easy to work with *abstractions* of subproblems and data, rather than always dealing with underlying representation or implementation.
  - Rich *Control Structures*.
    - And perhaps even support for developing new ones.
  - Support for defining and using *Data Structures*.
    - Some would also include *overloading* to support common syntax for operations on those data structures.
- A useful *Typing System*.
  - And other instances of *compile-time error detection*.
- A clear, understandable, and reasonable *Syntax*.
  - The legendary "NASA Fortran DO typo" is an example of bad syntax, as is the `=-` operator in the original C.
  - Some theorists say “If it’s difficult to write a program to parse it, it’s probably difficult for a programmer to get it right.”
- A language with *standards* for writing programs often has greater readability.
  - For example, Java has standards for naming, commenting, and capitalization.
  - Standards for libraries also enhance maintainability.
- The list can also consider fairly detailed issues. For example, does the language support exception handling? Does it permit aliasing?

## Hoare and Gabriel on the Design of Languages

- What do these two distinguished authors identify as the most important attributes of languages?
- Your good questions on Hoare:
  - Hoare states, "I fear that each reader will find some of my points wildly controversial". What points would have been considered "wildly controversial" at the time the paper was published.
  - From the hints that Hoare gives about, it seems that it would be almost unacceptable to have object-oriented programming languages around. He mentions that modularity does not mean simplicity, and readability and program structures are important design considerations. All these seem to be against the object-oriented paradigm that was in an infant state when he wrote the

paper. Yet, the OOP languages proved to be widely accepted in later years. Can this be explained by the acceptance model of Gabriel and the advantage of easy program debugging?

- The second reading was very specific in its requirements for a good programming language. It was my impression, that he thought of the issue as too straight forward and cut dry. I always thought that there had to be some tradeoffs in designing a computer language. I understand his focus on simplicity, but is it not a tradeoff for the power of your language? I'm not sure what I'm trying to say here. But if everything was as simple as he said, why wouldn't people just do things his way? For example, what are the advantages of using pointers and references (they seem quite important and widely used in C)?
- Your good questions on Gabriel:
  - The author predicted that object oriented programming would fail perhaps because it requires users to have "mathematical sophistication" (such as inheritance and polymorphism). Why has OO programming flourished contrary to Gabriel Richard's prediction?
  - This paper was authored about three decades after Hoare published his "Hints on Programming Language Design" but it seems that little has changed since then. The qualities of machine independence, use of familiar notations and existing popularity among others still dominate the programmer's decision on which language to use. So much so that all the languages that Hoare deemed to be well designed such as ALGOL and FORTRAN are either "dead or moribund" according to Richard. How useful then are Hoare's hints?
  - The author says that to a large extent the sophistication of a programming language is limited by the sophistication of the hardware. At the same time he says that a language must not be a resource hog, but should also be on a popular platform. I don't quite understand how we can hope to make a language popular, easily acceptable without too much innovation and also not use a lot of innovation?
- Hoare makes some subtle points with his examples. We will return to them on Friday.