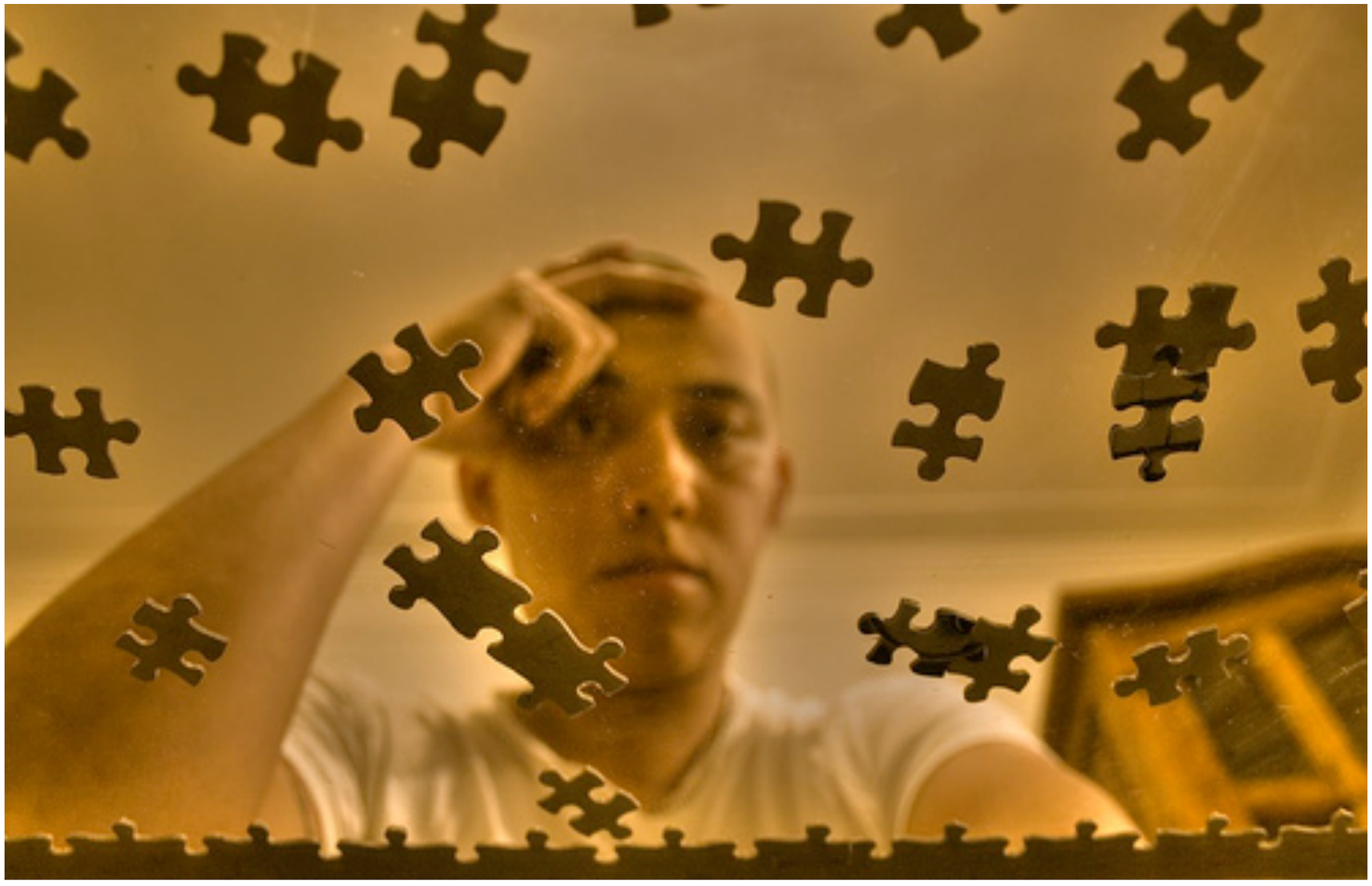
Instead of a conclusion

Tijs van der Storm

What we've been doing

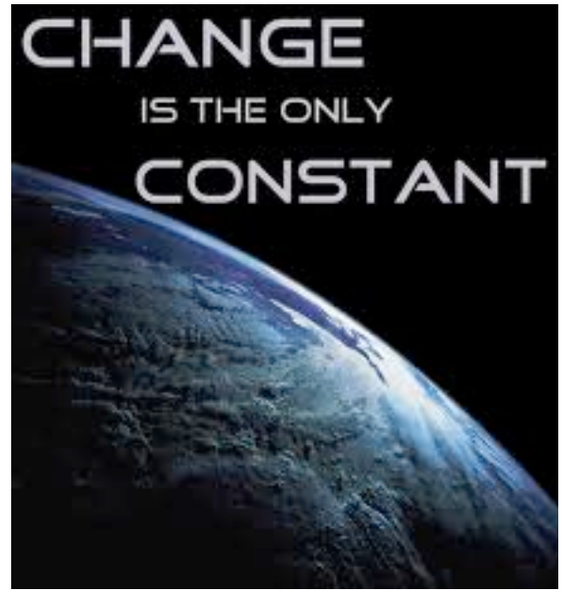
- Dealing with abstractions: classes, objects, methods etc.
- Discussing trade-offs wrt how to organize (e.g. visitor vs interpreter)
- Getting rid of fluff, boiler-plate, ugliness...
- Finding the right balance

How does it all fit together?




CHANGE
AHEAD


CHANGE IS THE ONLY CONSTANT


CHANGE
IS THE ONLY
CONSTANT


CHAN
IS THE
ONLY
CONSTANT


Change is
the only
Constant.


CHANGE
IS THE ONLY CONSTANT.


the only
constant
in life
is change.


Change is the only
CONSTANT


Change is the
Only Constant

Code that is easy to change

- Clean, readable and understandable
- Encapsulates design decisions
- Once and only once (DRY)
- Strong cohesion
- Weak coupling
- Open for extension

- Minimize confusion
- Leverage language's concepts
- Appropriate use of patterns
- Only talk to interfaces

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

Key Words and Phrases: software, modules, modularity, software engineering, KWIC index, software design

CR Categories: 4.0

Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

A Brief Status Report

The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code, but the systems most often used as examples of problem systems are highly-modularized programs and make use of the techniques mentioned above.

[1] Reprinted by permission of Prentice-Hall, Englewood Cliffs, N.J.

Information hiding

- Put design decisions behind walls
- Isolate change
- “Talk to interfaces”
- Make dependencies explicit

is everywhere...

- modules, classes, methods, etc.
- List vs. ArrayList etc.
- Encapsulate something that may *change*

On Understanding Data Abstraction, Revisited

William R. Cook

University of Texas at Austin

wcook@cs.utexas.edu

Abstract

In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper called "On understanding types, data abstraction, and polymorphism". Their work kicked off a flood of research on semantics and type theory for object-oriented programming, which continues to this day. Despite 25 years of research, there is still widespread confusion about the two forms of data abstraction, *abstract data types* and *objects*. This essay attempts to explain the differences and also why the differences matter.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects

General Terms Languages

Keywords object, class, abstract data type, ADT

1. Introduction

What is the relationship between *objects* and *abstract data types* (ADTs)? I have asked this question to many groups of computer scientists over the last 20 years. I usually ask it at

So what is the point of asking this question? Everyone knows the answer. It's in the textbooks. The answer may be a little fuzzy, but nobody feels that it's a big issue. If I didn't press the issue, everyone would nod and the conversation would move on to more important topics. But I do press the issue. I don't say it, but they can tell I have an agenda.

My point is that the textbooks mentioned above are wrong! Objects and abstract data types are not the same thing, and neither one is a variation of the other. They are fundamentally different and in many ways complementary, in that the strengths of one are the weaknesses of the other. The issues are obscured by the fact that most modern programming languages support both objects and abstract data types, often blending them together into one syntactic form. But syntactic blending does not erase fundamental semantic differences which affect flexibility, extensibility, safety and performance of programs. Therefore, to use modern programming languages effectively, one should understand the fundamental difference between objects and abstract data types.

While objects and ADTs are fundamentally different, they are both forms of *data abstraction*. The general concept of data abstraction refers to any mechanism for hiding the implementation details of data. The concept of data ab-

ADT vs OO

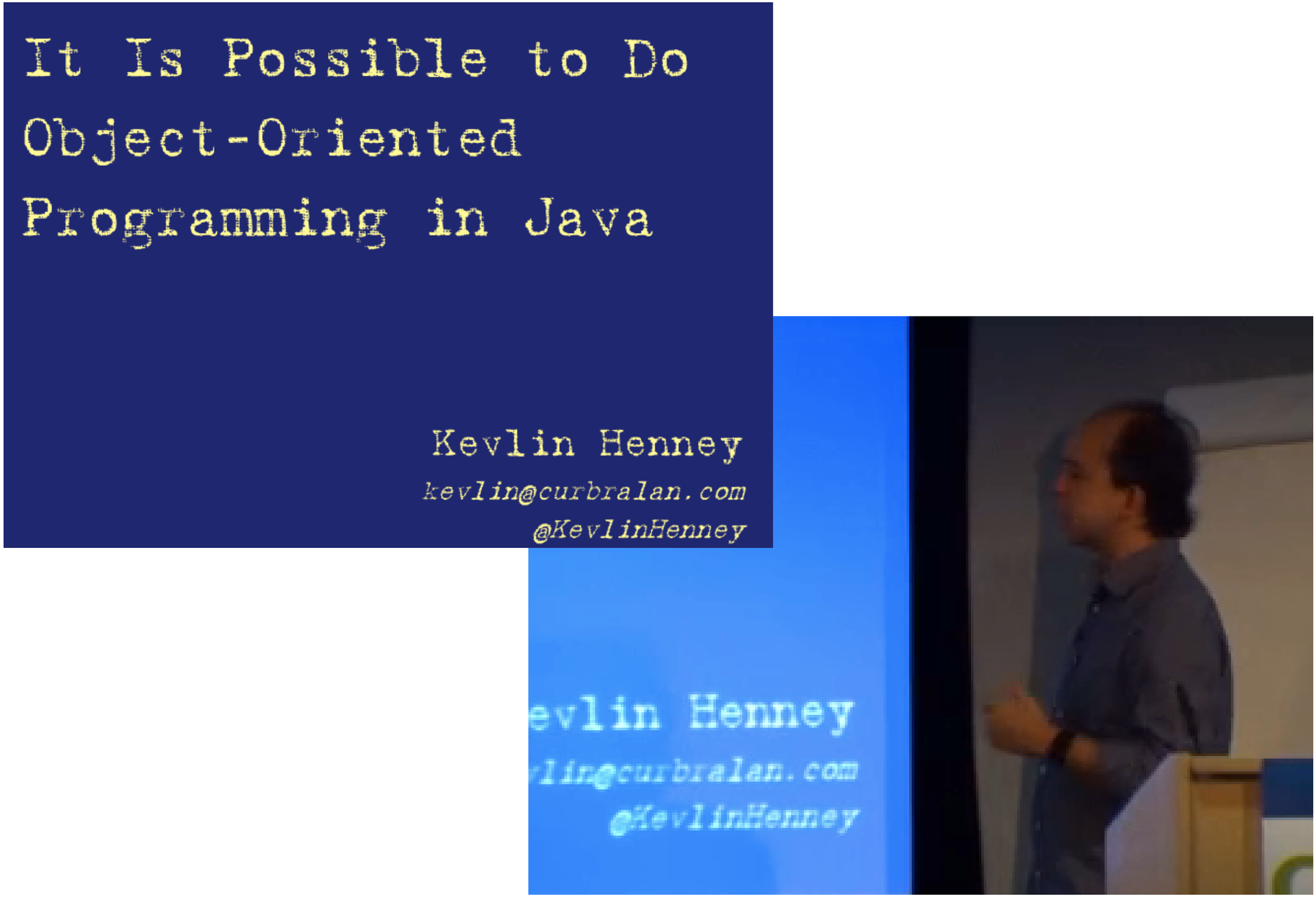
- Dual perspectives on *data abstraction*
- Different trade-offs

ADT

- Public type + operations
- Hidden, *priviliged* representation
- Facilitates optimization and verification
- Need static types

OO

- Objects are *behavioral* abstractions
  - "modules" that respond to messages
- Often easier to extend
  - objects simulating other objects
- Can be dynamically typed



http://www.infoq.com/presentations/It-Is-Possible-to-Do-OOP-in-Java

Policy Sciences 4 (1973), 155–169

Dilemmas in a General Theory of Planning*

HORST W. J. RITTEL
Professor of the Science of Design, University of California, Berkeley

MELVIN M. WEBBER
Professor of City Planning, University of California, Berkeley

ABSTRACT

The search for scientific bases for confronting problems of social policy is bound to fail, because of the nature of these problems. They are "wicked" problems, whereas science has developed to deal with "tame" problems. Policy problems cannot be definitively described. Moreover, in a pluralistic society there is nothing like the undisputable public good; there is no objective definition of equity; policies that respond to social problems cannot be meaningfully correct or false; and it makes no sense to talk about "optimal solutions" to social problems unless severe qualifications are imposed first. Even worse, there are no "solutions" in the sense of definitive and objective answers.

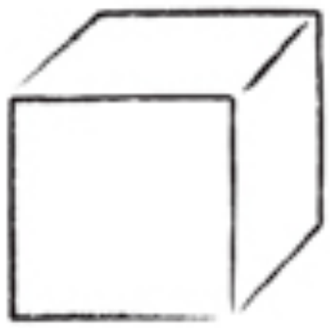
George Bernard Shaw diagnosed the case several years ago; in more recent times popular protest may have already become a social movement. Shaw averred that "every profession is a conspiracy against the laity." The contemporary publics are responding as though they have made the same discovery.

Few of the modern professionals seem to be immune from the popular attack—whether they be social workers, educators, housers, public health officials, policemen, city planners, highway engineers or physicians. Our restive clients have been telling us that they don't like the educational programs that schoolmen have been offering, the redevelopment projects urban renewal agencies have been proposing, the law-enforcement styles of the police, the administrative behavior of the welfare agencies, the locations of the highways, and so on. In the courts, the streets, and the political campaigns, we've been hearing ever-louder public protests against the professions' diagnoses of the clients' problems, against professionally designed governmental programs, against professionally certified standards for the public services.

It does seem odd that this attack should be coming just when professionals in

* This is a modification of a paper presented to the Panel on Policy Sciences, American Association for the Advancement of Science, Boston, December 1969.

TAMED



Well–defined problem

Algorithmic solution

[scientific management]

a more complex world

WICKED

Indefinable problem

Heuristic solution

[design management]

- There is no definitive formulation of a wicked problem
- Wicked problems have no stopping rule.
- Solutions to wicked problems are not true-or-false, but better or worse.
- There is no immediate and no ultimate test of a solution to a wicked problem.
- ...

The software construction course is wicked...

Fundamental trade-offs

- Flexibility vs safety
- Abstraction vs readability
- Simplicity vs genericity
- Scattering vs tangling
- Performance vs reuse
- ADT vs Objects
- ...

Kent Beck
@KentBeck

Following

first you learn the value of abstraction, then you learn the cost of abstraction, then you're ready to engineer

Reply Retweet Favorite More

97

Collective Wisdom
from the Experts

97 Things Every Programmer Should Know

O'REILLY®

Edited by Kevlin Henney

Contributions Appearing in the Book

1. Act with Prudence by Seb Rose
2. Apply Functional Programming Principles by Edward Garson
3. Ask "What Would the User Do?" (You Are not the User) by Giles Colborne
4. Automate Your Coding Standard by Filip van Laenen
5. Beauty Is in Simplicity by Jørn Ølmheim
6. Before You Refactor by Rajith Attapattu
7. Beware the Share by Udi Dahan
8. The Boy Scout Rule by Uncle Bob
9. Check Your Code First before Looking to Blame Others by Allan Kelly
10. Choose Your Tools with Care by Giovanni Asproni
11. Code in the Language of the Domain by Dan North
12. Code Is Design by Ryan Brush
13. Code Layout Matters by Steve Freeman
14. Code Reviews by Mattias Karlsson
15. Coding with Reason by Yechiel Kimchi
16. A Comment on Comments by Cal Evans
17. Comment Only What the Code Cannot Say by Kevlin Henney
18. Continuous Learning by Clint Shank
19. Convenience Is not an -ility by Gregor Hohpe
20. Deploy Early and Often by Steve Berczuk
21. Distinguish Business Exceptions from Technical by Dan Bergh Johnsson

Other Edited Contributions

1. Abstract Data Types by Aslam Khan
2. Acknowledge (and Learn from) Failures by Steve Berczuk
3. Anomalies Should not Be Ignored by Keith Gardner
4. Avoid Programmer Churn and Bottlenecks by Jonathan Danylko
5. Balance Duplication, Disruption, and Paralysis by Johannes Brodwall
6. Be Stupid and Lazy by Mario Fusco
7. Become Effective with Reuse by Vijay Narayanan
8. Better Efficiency with Mini-Activities, Multi-Processing, and Interrupted Flow by Siv Fjellkårstad
9. Code Is Hard to Read by Dave Anderson
10. Consider the Hardware by Jason P Sage
11. Continuous Refactoring by Michael Hunger
12. Continuously Align Software to Be Reusable by Vijay Narayanan
13. Data Type Tips by Jason P Sage
14. Declarative over Imperative by Christian Horsdal
15. Decouple that UI by George Brooke
16. Display Courage, Commitment, and Humility by Ed Sykes
17. Dive into Programming by Wojciech Rynczuk
18. Don't Be a One Trick Pony by Rajith Attapattu
19. Don't Be too Sophisticated by Ralph Winzinger
20. Don't Reinvent the Wheel by Kai Tödter
21. Don't Use too Much Magic by Mario Fusco
22. Done Means Value by Raphael Marvie
23. Execution Speed versus Maintenance Effort by Paul Colin Gloster

Acknowledge (and Learn from) Failures