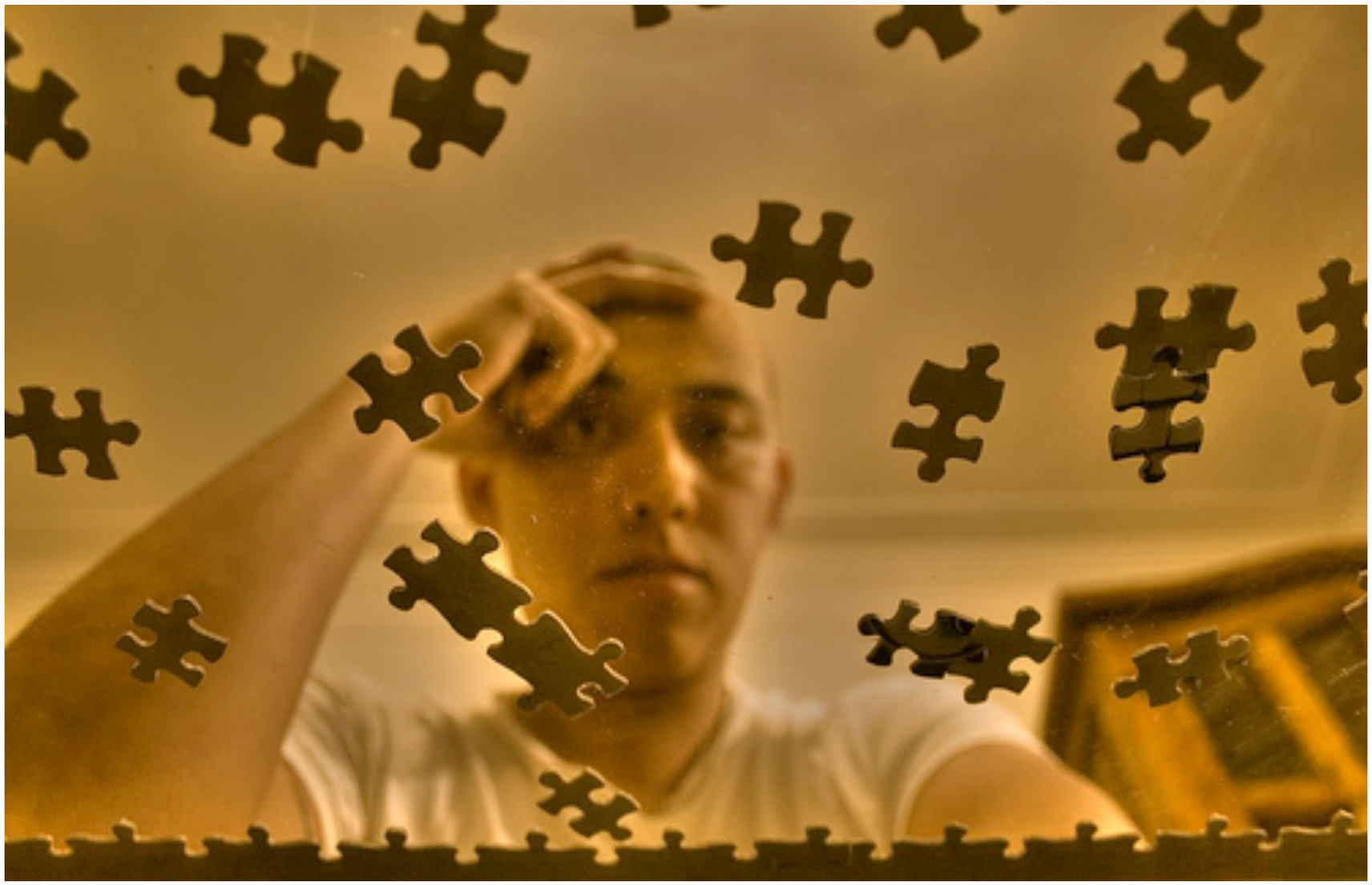
Instead of a conclusion

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*

Chapter 9

Modularity in the Design of Complex Engineering Systems

Carliss Y. Baldwin
Kim B. Clark
Harvard Business School
Boston, MA 02163

1. Introduction

In the last decade, the concept of modularity has caught the attention of engineers, management researchers and corporate strategists in a number of industries. When a product or process is "modularized," the elements of its design are split up and assigned to modules according to a formal architecture or plan. From an engineering perspective, a modularization generally has three purposes:

- To make complexity manageable;
- To enable parallel work; and
- To accommodate future uncertainty.

Modularity accommodates uncertainty because the particular elements of a modular design may be changed after the fact and in unforeseen ways as long as the design rules are obeyed. Thus, within a modular architecture, new module designs may be substituted for older ones easily and at low cost.

This chapter will make three basic points. First, we will show that *modularity is a financial force* that can change the structure of an industry. Then, we will explore the *value* and *costs* that are associated with constructing and exploiting a modular design. Finally we will examine the ways in which modularity shapes organizations and the risks that it poses for particular firms.[1]

---

[1] Some of the arguments and figures in this paper are taken from Baldwin and Clark, 2000. The figures are reprinted by permission.

Option value

- Modules are options
  - explore multiple, take best
  - increase parallelism
  - evolution without breaking the whole systems

THE ARCHITECTURE OF COMPLEXITY

HERBERT A. SIMON*

Professor of Administration, Carnegie Institute of Technology

(*Read April 26, 1962*)

A NUMBER of proposals have been advanced in recent years for the development of "general systems theory" which, abstracting from properties peculiar to physical, biological, or social systems, would be applicable to all of them.[1] We might well feel that, while the goal is laudable, systems of such diverse kinds could hardly be expected to have any nontrivial properties in common. Metaphor and analogy can be helpful, or they can be misleading. All depends on whether the similarities the metaphor captures are significant or superficial.

It may not be entirely vain, however, to search for common properties among diverse kinds of complex systems. The ideas that go by the name of cybernetics constitute, if not a theory, at least a point of view that has been proving fruitful over a wide range of applications.[2] It has been useful to look at the behavior of adaptive systems in terms of the concepts of feedback and homeostasis, and to analyze adaptiveness in terms of the theory of selective information.[3] The ideas of feedback and information provide a frame of reference for viewing a wide range of situations, just as do the ideas of evolution, of relativism, of axiomatic method, and of operationalism.

In this paper I should like to report on some things we have been learning about particular kinds of complex systems encountered in the behavioral sciences. The developments I shall discuss arose in the context of specific phenomena, but the theoretical formulations themselves make little reference to details of structure. Instead they refer primarily to the complexity of the systems under view without specifying the exact content of that complexity. Because of their abstractness, the theories may have relevance—application would be too strong a term—to other kinds of complex systems that are observed in the social, biological, and physical sciences.

In recounting these developments, I shall avoid technical detail, which can generally be found elsewhere. I shall describe each theory in the particular context in which it arose. Then, I shall cite some examples of complex systems, from areas of science other than the initial application, to which the theoretical framework appears relevant. In doing so, I shall make reference to areas of knowledge where I am not expert—perhaps not even literate. I feel quite comfortable in doing so before the members of this society, representing as it does the whole span of the scientific and scholarly endeavor. Collectively you will have little difficulty, I am sure, in distinguishing instances based on idle fancy or sheer ignorance from instances that cast some light on the ways in which complexity exhibits itself wherever it is found in nature. I shall leave to you the final judgment of relevance in your respective fields.

I shall not undertake a formal definition of

* The ideas in this paper have been the topic of many conversations with my colleague, Allen Newell. George W. Corner suggested important improvements in biological content as well as editorial form. I am also indebted, for valuable comments on the manuscript, to Richard H. Meier, John R. Platt, and Warren Weaver. Some of the conjectures about the nearly decomposable structure of the nucleus-atom-molecule hierarchy were checked against the available quantitative data by Andrew Schoene and William Wise. My work in this area has been supported by a Ford Foundation grant for research in organizations and a Carnegie Corporation grant for research on cognitive processes. To all of the above, my warm thanks, and the usual absolution.

[1] See especially the yearbooks of the Society for General Systems Research. Prominent among the exponents of general systems theory are L. von Bertalanffy, K. Boulding, R. W. Gerard, and J. G. Miller. For a more skeptical view—perhaps too skeptical in the light of the present discussion—see H. A. Simon and A. Newell, Models: their uses and limitations, *in* L. D. White, ed., *The state of the social sciences,* 66–83, Chicago, Univ. of Chicago Press, 1956.

[2] N. Wiener, *Cybernetics,* New York, John Wiley & Sons, 1948. For an imaginative forerunner, see A. J. Lotka, *Elements of mathematical biology,* New York, Dover Publications, 1951, first published in 1924 as *Elements of physical biology.*

[3] C. Shannon and W. Weaver, *The mathematical theory of communication,* Urbana, Univ. of Illinois Press, 1949; W. R. Ashby, *Design for a brain,* New York, John Wiley & Sons, 1952.

Hierarchy

- Related to option value, information hiding
- Hierarchical systems are easy to evolve
- “Layers of abstraction”

The watches the men made consisted of about 1,000 parts each. Tempus had so constructed his that if he had one partly assembled and had to put it down—to answer the phone say—it immediately fell to pieces and had to be reassembled from the elements. The better the customers liked his watches, the more they phoned him, the more difficult it became for him to find enough uninterrupted time to finish a watch.

The watches that Hora made were no less complex than those of Tempus. But he had designed them so that he could put together subassemblies of about ten elements each. Ten of these subassemblies, again, could be put together into a larger subassembly; and a system of ten of the latter subassemblies constituted the whole watch. Hence, when Hora had to put down a partly assembled watch in order to answer the phone, he lost only a small part of his work, and he assembled his watches in only a fraction of the man-hours it took Tempus.

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 dinner, or over drinks. The typical response is a variant of "objects are a kind of abstract data type".

This response is consistent with most programming language textbooks. Tucker and Noonan [57] write "A class is itself an abstract data type". Pratt and Zelkowitz [51] intermix discussion of Ada, C++, Java, and Smalltalk as if they were all slight variations on the same idea. Sebesta [54] writes "the abstract data types in object-oriented languages... are called classes." He uses "abstract data types" and "data abstraction" as synonyms. Scott [53] describes objects in detail, but does not mention abstract data types other than giving a reasonable discussion of opaque types.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

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 abstraction has existed long before the term "data abstraction" came into existence. In mathematics, there is a long history of abstract representations for data. As a simple example, consider the representation of integer sets. Two standard approaches to describe sets abstractly are as an *algebra* or as a *characteristic function*. An algebra has a sort, or collection of abstract values, and operations to manipulate the values[1]. The characteristic function for a set maps a domain of values to a boolean value, which indicates whether or not the value is included in the set. These two traditions in mathematics correspond closely to the two forms of data abstraction in programming: algebras relate to abstract data types, while characteristic functions are a form of object.

In the rest of this essay, I elaborate on this example to explain the differences between objects and ADTs. The

[1] The sort, or carrier set, of an algebra is often described as a set, making this definition circular. Our goal is to define specific set abstractions with restricted operations, which may be based on and assume a more general concept of sets

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
- No hidden representation
- Can be dynamically typed

It Is Possible to Do Object-Oriented Programming in Java

Kevlin Henney
kevlin@curbralan.com
@KevlinHenney

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

The Expression Problem

Philip Wadler, 12 November 1998

The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we take expressions as the data type, begin with one case (constants) and one function (evaluators), then add one more construct (plus) and one more function (conversion to a string).

	eval	print	check
Add			
Mul			
Sub			
…			

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.

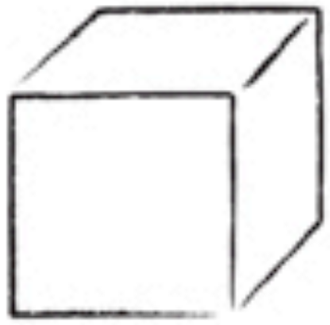
George Bernard Shaw diagnosed the case several years ago; in more recent times popular protest may have already become a social movement. Shaw averred that "every profession is a conspiracy against the laity." The contemporary publics are responding as though they have made the same discovery.

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
- ...

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