Meilir Page-Jones

Comparing Techniques by Means of Encapsulation and Connascence

Today the object-oriented approach to software development is at the height of fashion. As such, it threatens to replace the structured approach which was the staple development approach of the 1970s and 1980s.

Both approaches have their genesis in programming. Structured programming began in the late 1960s and is usually traced to workers such as Dijkstra [4]. Although object orientation is a river with many tributaries, the source of object-oriented programming is often traced—inter alia—to the late 1960s and the Simula language or to the Smalltalk work of Alan Kay and others at Xerox PARC, *circa* 1970 [5, 6].

Therefore, the structured approach rapidly grew into design and analysis techniques. Indeed, almost contemporaneously with the inception of structured programming, Larry Constantine and others were establishing the foundations of structured design [2]. Throughout the 1970s, workers such as Doug Ross [11] and Tom DeMarco [3] were developing a technique that came to be known as structured analysis.

Structured programming saw its initial rise in popularity in the early 1970s; structured design grew in popularity in the mid-1970s and structured analysis gained many adherents in the late 1970s. By contrast, object-oriented programming was a relative curiosity until the early 1980s. Object-oriented design and object-oriented analysis made their first significant appearances in the late 1980s [1].

During most of the two decades of the coexistence of the structured and the object-oriented approaches, there has been little interaction between researchers of the two approaches. Consequently, the two fields tended to develop different vocabularies and different foci of utilization. These apparent differences have tended to obscure some of the similarities between the structured approach and the object-oriented approach.

On the other hand, the two approaches are not—as some people would imply—simply isomorphs with disparate terminology. There are some deep structural distinctions between them.

In this article, I discuss the similarities and differences between the structured and object-oriented approaches to software design by means of two principles: encapsulation and connascence. I conclude by suggesting experiments involving connascence, the use of tools to mediate its effects and a set of steps necessary to define any new design paradigm.

Encapsulation and Connascence

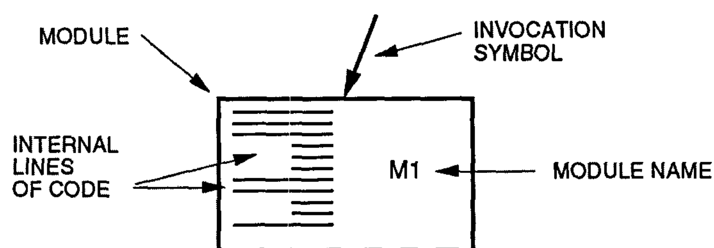
Both the structured and object-oriented approach rely on encapsulation—the technique by which a set of software components is aggregated into a structure that can be treated as a unit from an external point of view.

Structured design encapsulates lines of code into procedural units that Constantine called *modules*. Examples in traditional languages include procedures, subroutines and functions. I will term this type of encapsulation level-1 or "L1" encapsulation as seen in Figure 1.

Object-oriented design encapsulates units called *classes*.[1] The structure of a class is more complex than that of the structured-design module, since a class is a template or descriptor representing a set of modules (more often called *methods*) packaged around an internal structure that will hold the state of each object belonging to the class. This structure is the set of *internal variables*. I will term this type of encapsulation "L2." As Figure 2 indicates, L2 encapsulation includes L1 within it, because the methods of a class are structurally similar to the modules of structured design. (Incidentally, L0 encapsulation would be *null encapsulation*, in which the highest-level construct would be the humble line of procedural code or elementary variable.)

Structured design offers two criteria for evaluating the quality of encapsulation resulting from a given partitioning into modules. These criteria are: coupling, which measures the binding between code elements that are found in distinct modules; and cohesion, which measures the binding between code elements that are found in the same module. The ideal of structured design is to minimize the coupling of a system's design and to maxi-

[1]Objects are the structurally equivalent constructs that are spawned dynamically during system execution. In some implementations the class and the object are degenerately identical.



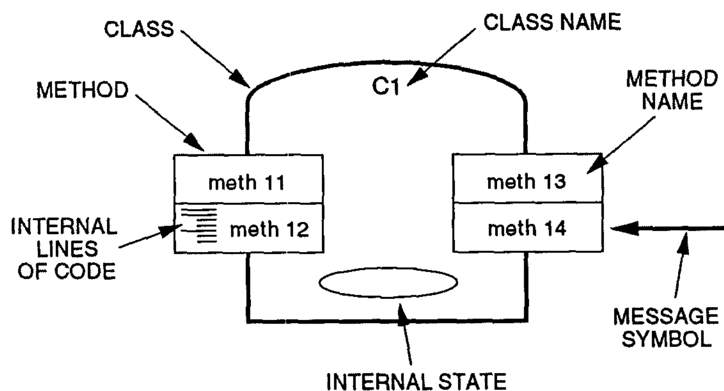


Figure 1. L1 Encapsulation

Figure 2. L2 Encapsulation

mize the cohesion of the design.

As we saw indicated earlier, the structures of object-oriented design are more complex than those of structured design. Consequently, there are potentially more criteria for evaluating object-oriented design quality. For example, in addition to the cohesion of a method in terms of the lines of code it contains, we could talk about the cohesion of a class in terms of the methods that are defined on it. Furthermore, we could talk about coupling between classes, or between methods of the same class or between methods of different classes.

In order to discuss the quality of an object-oriented design, we could invent new terms to augment those of coupling and cohesion, which would give labels to such criteria. However, to do so would be to overlook a chance to generalize coupling and cohesion into the single measure of *connascence*.[2]

I say that two elements of software are connascent if they are "born together" in the sense that they somehow share the same destiny. More explicitly, I define two software elements *A* and *B* to be connascent if there is at least one change that could be made to *A* that would necessitate a change to *B* in order to preserve overall correctness.

As a very simple example, let us take *A* to be the line declaring:

int i;

and *B* to be the assignment:

i := 7;

There are at least two examples of connascence between *A* and *B*. For instance, in the (unlikely) event that *A* were changed to **char** i; then *B* would certainly have to be changed, too. This is *connascence of type*. Also, if *A* were changed to **int** j; then *B* should be changed to j := 7;. This is **connascence of name.**

[2]I chose the word "connascence" from the Latin roots meaning "born together." It is etymologically close to the French "connaissance" meaning knowledge, awareness or consciousness. Although I'm blamed for coining the word, it can be found in Chambers Twentieth Century Dictionary (W&R Chambers Ltd., London).

The ideals of low coupling and high cohesion now generalize beyond structured design and yield a statement that is applicable to object-oriented design (or indeed to any future design paradigm with partitioning, encapsulation and visibility rules): *Eliminate any unnecessary connascence and then minimize connascence across encapsulation boundaries by maximizing connascence within encapsulation boundaries.*

One prediction of connascence, as applied specifically to object-oriented design, relates to the use of inheritance. Some designers allow a class to make use of both externally visible and internally visible components of a superclass. (The latter in Smalltalk, for example, might be the explicit use in a subclass of certain instance variables of a superclass.) This creates a great deal of connascence across major (class) encapsulation boundaries.

This use of inheritance implies that the maintainer of classes (C11, C12, etc.) that inherit from C1 would need to stay aware of any changes to the internal design of C1. Therefore, C1's maintainer could create disastrous results in descendant classes even through an innocuous name-change to an internal variable of C1.

Therefore, connascence tells us that inheritance by a subclass from a superclass should be restricted to only those features of the superclass that are already externally visible. Better yet, the notion of inheriting abstract behavior should be divorced from the notion of inheriting the internal implementation of such behavior. (See [10] for a further discussion of this point.)

Another object-oriented design prediction of connascence recommends foregoing the use of the C++ "friend" construct—a construct that blatantly promotes connascence across encapsulation boundaries. (One might say: "With friends like that, who needs enemies!") The argument for this design recommendation is similar to this argument.

Here we see two forms of connascence: name and type. Other forms include the following:

• Value, which indicates that two software elements must contain the same absolute values.
• *Position,* which refers to the specific location of two software elements in the overall code structure. Examples might be: the positional correspondence of actual and formal parameters; and the colocation of procedural lines that are meant to execute in sequence.
• *Algorithm,* which means that two software elements must agree on some common algorithm for their correct execution.
• *Meaning,* which implies that two software elements must be defined to have identical semantic (rather than merely syntactic) structure.

These forms of connascence are generic, in the sense that they could be defined in almost any design paradigm. It is possible, however, to define other forms of connascence for particular paradigms.

For example, in the data dictionary of classic structured analysis (as defined in [3], for example) we have a variation on the connascence of type, even though there is no explicit mention of type. Say there are two entries: *current-customer* and *potential-customer.* Assume that a user change, such as adding a field for a fax machine number, applies to data-dictionary entries for all kinds of customers. Thus there is a connascence between each of *current-customer* and *potential-customer* and the "phantom type" CUSTOMER.

This problem has always made changing a classic SA data dictionary a precarious activity, resulting in many inconsistent entries. It remains a problem even in current SA CASE tools that lack an explicit notion of type. (Incidentally, the reason that I introduced two kinds of customer in this example was that if there were only one kind—e.g., *current-customer*—that one would serve as a surrogate for the type CUSTOMER. The connascence between *current-customer,* and the type CUSTOMER would no longer exist. Or, to be precise, it would be localized at the single point *current-customer* and would cease to be a practical problem.)

In object orientation we might have the following:

• *Class membership,* which is effectively analogous to type, since a class represents an implementation of a type in object-oriented design.
• *Class variables,* that is the variables shared by all objects instantiated from the same class.

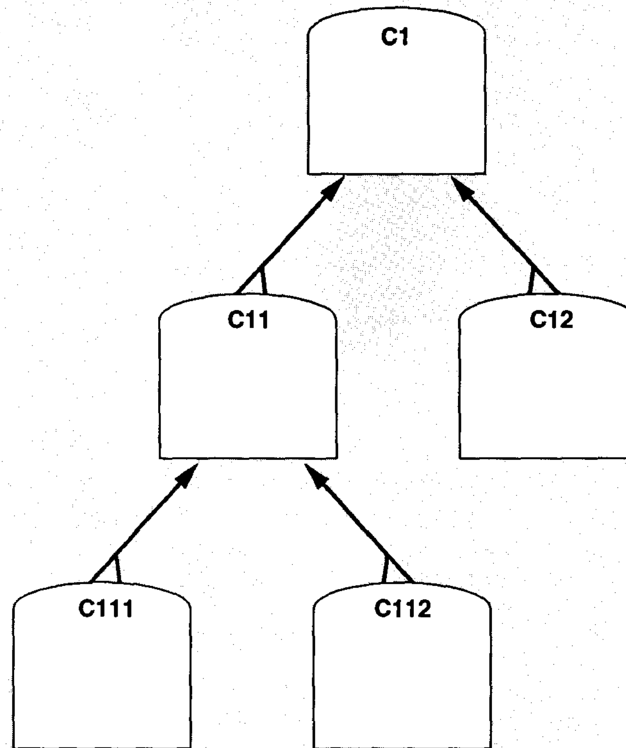


Figure 3. The class inheritance hierarchy under C1

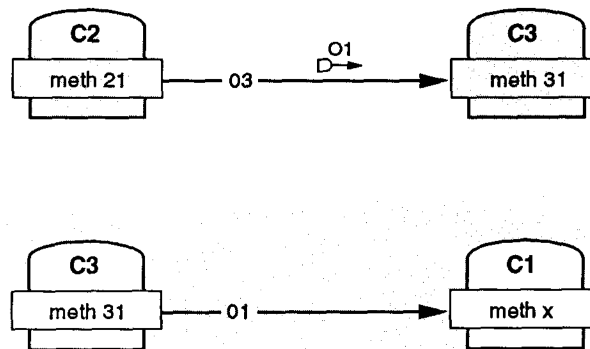


Figure 4. The message 03.meth31 (01) shown in UON

Figure 5. The message 01.methx(...,...,...)

• An odd kind of connascence derived from polymorphism, which can be illustrated by the following example: Assume a hierarchy of classes and subclasses, C1, C11, C12, C111, C112, ..., as shown in Figure 3.

Assume further that a method meth21 of an object o2, of class C2, sends a message to an object o3, of class C3. This message passes as one of its parameters an object o1, of class C1. So, within meth21 of o2, we might see the following statement shown graphically in Uniform Object Notation (UON) in Figure 4 [9]:

o3.meth31 (o1); —call this line 1

Let us further assume that within the method meth31 there is a message to the object o1—the object that was just passed to the method as shown graphically in Figure 5. (This message invokes the method, methx, of the object o1. In this discussion we do not care what the message's parameters are—if any.)

o1.methx (...,...,...); —call this line 2

The issue of connascence is this: In line 1 we could have legally passed an object o11, of class C11, or an object o112, of class C112, or indeed an object of any of the subclasses of C1. (See [8, p225] for a discussion of this principle.) If (see line 2) methx is defined on the class C1, then there will be no problem; line 2 will work correctly no matter which (legal) object was passed.

However, if methx is defined not on C1 but only on *some* classes of the class hierarchy beneath C1 then there will be a problem. Effectively, what we have done here is to ensure that the subclass hierarchy beneath C1 is not a true subtype hierarchy. This can be achieved either by defining methx on only some of C1's subclasses or by defining it on C1 and then overriding it in some subclasses. (See [7] for a further discussion.)

In this new situation, line 2 can be made to fail (if, for instance, the object passed to meth31 is of class C112, which lacks a definition of methx). There is thus a *connascence of polymorphism* between meth31 and the class hierarchy under C1. The nature of the problem could be modified—if not exactly solved—in the following ways:

1. Return to the situation wherein methx is defined on *all* classes under C1. While this is a good design practice—it aligns the subclass hierarchy more closely to a subtype hierarchy—given the existence of line 2, it is not always practical.

As a simple example, take C1 to be BIRD and C112 to be OSTRICH. Methx might be *fly*, which is defined on almost the entire hierarchy under C1—except C112. Formally speaking, an OSTRICH is not a subtype of BIRD; it does not exhibit a strict superset of the behavior of BIRD. Nevertheless, it seems pedantic in the extreme not to have the class OSTRICH inherit from the class BIRD . . . until some object-oriented programmer tries to get an ostrich to fly, that is.

2. Have meth21, the sender of the message in line 1, guard against the possibility that methx may not be defined on the class of the object sent as the parameter of that message. Unfortunately, this creates an additional connascence—this time between meth21 and meth31—because meth21 needs to know what meth31 needs to be guarded against. Neither does this solution remove the original connascence; it merely shifts it to being between meth21 and the class hierarchy under C1.

Observe that these examples of connascence exhibit various degrees of "explicitness." By this, I mean that some forms of connascence (e.g., connascence, by name) are readily apparent from the code itself (or could be revealed by a simple text editor).

Other cases (e.g., connascence by polymorphism) are usually obscure and require human skill to be revealed. It seems likely this latter, implicit form of connascence will probably incur the greatest development and maintenance costs. However, to be fair to the concept of polymorphism—and object orientation in general—it is true that polymorphism eliminates other forms of connascence that lead to many code modifications when, say, a new subclass is introduced.

Conclusion

Further research is needed into the forms of connascence that apply to modern software-design paradigms, especially to the paradigm of object-oriented design, which is now becoming more and more popular. From that basis we need to elicit by experiment the degrees of ill that are caused by the various forms of connascence. These experiments would measure the effects of connascence (both within and across encapsulation boundaries) on presumed dependent variables such as debugging cost and maintenance cost.

Connascence represents a set of interdependencies in software, some of which are not readily or immediately apparent from the code itself. Therefore, it would be very valuable if future CASE tools were to assist the human in making instances of implicit connascence more explicit, especially in structures that exhibit L2—or higher—encapsulation.

Finally, since it is unlikely that object orientation (or any other L2 encapsulation) will be the last word in design paradigms, let me suggest six steps through which to base a definition of any future design paradigm:

1. State the intended purpose and scope of applicability of the paradigm.
2. State the paradigm's encapsulation structure. State what the paradigm's components are and which components are contained within which.
3. In terms of the encapsulation structure, state the default visibility rules of the paradigm. This will prescribe the allowed connections among components and state the "boundaries of privacy" established

by the encapsulation structure.
4. List the possible forms of connascence that obtain within this paradigm. There will be explicit connascence, which follows the lines of the stated encapsulation structure and visibility, and implicit connascence, which will be more difficult to perceive because it subtly transcends the "official" paradigm structure.
5. Classify as much as possible the pernicious effects of each form of connascence in various contexts.
6. Suggest heuristics for deriving or modifying software designed under this paradigm in order to minimize the pernicious effects. ◼

References

1. Booch, G. *Object-Oriented Design with Applications.* Benjamin/Cummings, Redwood City, Calif., 1990.
2. Constantine, L. and Yourdon, E. *Structured Design.* Prentice Hall, Englewood Cliffs, NJ, 1979.
3. DeMarco, T. *Structured Analysis and System Specification.* Prentice Hall, Englewood Cliffs, NJ, 1978.
4. Dijkstra, E. *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, New York, NY, 1982.
5. Goldberg, A. and Robson, D. *Smalltalk-80: The Language.* Addison-Wesley, Reading, Mass., 1989.
6. Kay, A. *The Reactive Engine.* The University of Utah, Department of Computer Science, Aug. 1969.
7. LaLonde, W. and Pugh, J. Subclassing not= subtyping not= is-a. *J. Object-Oriented Prog. 3* 5 (1991) 57–62.
8. Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, 1988.
9. Page-Jones, M., Constantine, L. and Weiss, S. The uniform object notation. *Comput. Lang.* (Oct. 1990).
10. Porter III, H.H. Separating the subtype hierarchy from the inheritance of implementation. *J. Object-Oriented Prog. 4,* 9 (1992) 20–29.
11. Ross, D.T. and Schoman, K.E. Structured analysis for requirements definition, *IEEE Trans. Soft. Eng.* (Jan. 1977).

CR Categories and Subject Descriptors: D.2.1 [**Software**]: Software Engineering — *requirements/specifications;* D.2.10 [**Software**]: Software Engineering—*design;* D.3.3 [**Software**]: Program Languages—*language constructs and features;* I.6.0 [**Computing Methodologies**]: Simulation and Modeling—*general;* I.6.3 [**Computing Methodologies**]: Simulation and Modeling—*applications;* K.6.3 [**Computing Milieux**]: Management of Computing and Information Systems—*software management;* K.6.4 [**Computing Milieux**]: Management of Computing and Information Systems—*system management*

General Terms: Design, Methodology

Additional Key Words and Phrases: Analysis, connascence, coupling, modeling, visibility

About the Author:

MEILIR PAGE-JONES is a senior consultant at Wayland Systems Inc. He is currently conducting research into methods for applying object-oriented structures to the problem domains found in business and real-time applications. **Author's Present Address:** Wayland Systems Inc., Suite 14, 12819 SE 38 St., Bellevue, WA 98006, 76334.1247@compuserve.com.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/92/0900-147 $1.50

Run-Time Errors

Purify, a new software testing tool, detects run-time errors and identifies their causes:
- Reading uninitialized memory
- Reading or writing freed memory
- Array bounds violations
- Memory Leaks

Instruction-level checking
- Checks all code, including third party libraries
- Fast and comprehensive

Pure Software currently supports:
- C and C++
- SunOS 4.1.x

Purify can help you create more reliable software.
To find out more call or send email to *info @ pure.com*
(415) 903-5100

PURE SOFTWARE
MAKING SOFTWARE MORE RELIABLE

Circle #86 on Reader Service Card