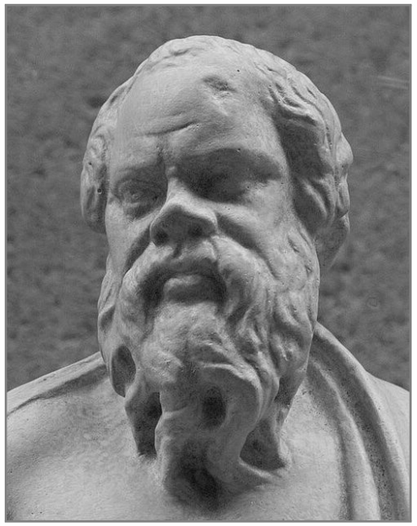
TCC and LCC

Yegor Bugayenko

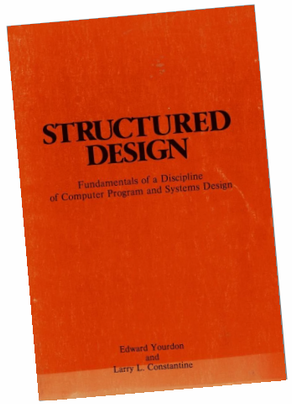
Lecture #8 out of 24
80 minutes

The slidedeck was presented by the author in this YouTube Video



“**Socrates**: I am a lover of these processes of division and bringing together, as aids to speech and thought; and if I think any other man is able to see things that can naturally be collected into one and divided into many, him I follow after and walk in his footsteps as if he were a god.”

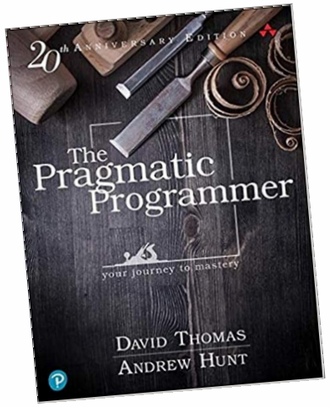
— Plato. Phaedrus (Dialogue), 370 B.C.



Edward Yourdon

"Module cohesion may be conceptualized as the cement that holds the processing elements of a module together. In a sense, a high degree of module cohesion is an indication of close approximation of inherent problem structure."

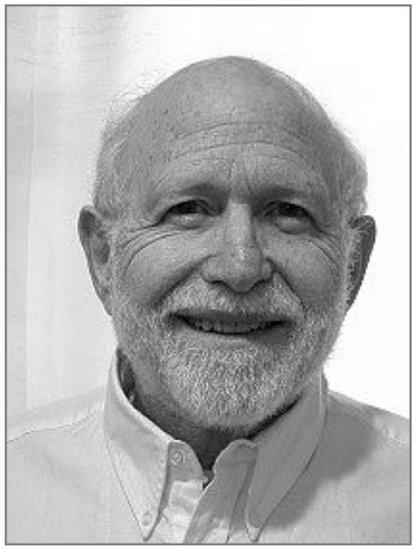
— Edward Yourdon and Larry Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice Hall, 1979. doi:10.5555/578522



Andrew Hunt

"We want to design components that are self-contained: independent, and with a single, well-defined purpose."

— Andrew Hunt and Dave Thomas. *The Pragmatic Programmer: From Journeyman to Master.* Pearson Education, 1999. doi:10.5555/320326



James M. Bieman

"We define two measures of class cohesion based on the direct and indirect connections of method pairs: TCC and LCC."

— James M. Bieman and Byung-Kyoo Kang. Cohesion and Reuse in an Object-Oriented System. *SIGSOFT Software Engineering Notes*, 20(51):259–262, 1995. doi:10.1145/223427.211856

Connectivity Between Methods in a Class

3.1 Connectivity between methods

The direct connectivity between methods is determined from the class abstraction. If there exists one or more common instance variables between two method abstractions then the two corresponding methods are *directly connected*.

Two methods that are connected through other directly connected methods are *indirectly connected*. The indirect connection relation is the transitive closure of direct connection relation. Thus, a method M_1 is indirectly connected with a method M_n if there is a sequence of methods $M_2, M_3, \ldots, M_{n-1}$ such that

$$M_1 \, \delta \, M_2, \cdots, M_{n-1} \, \delta \, M_n$$

where $M_i \, \delta \, M_j$ represents a direct connection.

"An instance variable is directly used by a method M if the instance variable appears as a data token in the method M. The instance variable may be defined in the same class as M or in an ancestor class of the class. $DU(M)$ is a set of instantce variables directly used by a method M."

Source: James M. Bieman and Byung-Kyoo Kang. Cohesion and Reuse in an Object-Oriented System. *SIGSOFT Software Engineering Notes*, 20(51):259–262, 1995. doi:10.1145/223427.211856

Tight and Loose Class Cohesion (TCC+LCC)

```
1  class Rectangle
2    int x, y, w, h;
3    int area()
4      return w * h;
5    int move(int dx, dy)
6      x += dx; y += dy;
7    int resize(int dx, dy)
8      w += dx; h += dy;
9    bool fit()
10     return w < 100
11       && x < 100;
```

Max possible connections (NP):
$N \times (N-1)/2 = 4 \times 3/2 = 6$

Directly connected (NDC $= 4$):
area+fit, area+resize, move+fit, resize+fit

Indirectly connected (NIC $= 2$):
area+move, move+resize

$\text{TCC} = \text{NDC}/\text{NP} = 4/6 = 0.66$
$\text{LCC} = (\text{NDC}+\text{NIC})/\text{NP} = 6/6 = 1.00$

Cohesion and Reuse in an Object-Oriented System*

James M. Bieman and Byung-Kyoo Kang
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
(303) 491-7096, Fax: (303) 491-2466
bieman@cs.colostate.edu, kang@cs.colostate.edu

Abstract

We define and apply two new measures of object-oriented class cohesion to a reasonably large C++ system. We find that most of the classes are quite cohesive, but that the classes that are reused more frequently via inheritance exhibit clearly lower cohesion.

1 Introduction

Software developers aim for systems with high cohesion and low coupling. The value of these goals has not been validated empirically [6]. Rather, they have been justified on the basis of intuition. The amount of reuse — the number of times that a component is reused — is an indicator of reusability. Of course, other factors such as the usefulness of a component are also components of reusability.

Cohesion refers to the "relatedness" of module components. A highly cohesive component is one with one basic function. It should be difficult to split a cohesive component. Cohesion can be classified using an ordinal scale that ranges from the least desirable category—*coincidental cohesion*—to the most desirable—*functional cohesion* [7]. To apply this cohesion model to classes in object-oriented systems, we need to add a new classification, *data cohesion* [5].

Bieman and Ott developed a set of functional cohesion measures based on program slices [2]. Those measures apply only to individual functions; their application to entire classes is not obvious. Chidamber and Kemerer developed a Lack of Cohesion in Methods (LCOM) measure for object-oriented software [4]. LCOM is effective at identifying the most non-cohesive classes, but it is not effective at distinguishing between partially cohesive classes. LCOM indicates lack of cohesion only when, compared pairwise, fewer than half of the paired methods use the same instance variables.

*Research partially supported by NASA Langley Research Center grant NAG1-1061.

0-89791-729-1/95/0004...$3.50

Cohesion measures that are sensitive to small changes are needed in order to evaluate the relationship between cohesion and reuse. In this paper, we develop sensitive class cohesion measures and apply them to a reasonably large C++ system. We evaluate the relationship between class cohesion and private reuse in this system.

2 Class Cohesion

The components of a class are the instance variables and methods defined in the class plus those that are inherited. A method and an instance variable are related by the way that an instance variable is used by the method. Two methods are related (connected) through instance variable(s) if both methods use the instance variable(s). Class cohesion is defined in terms of the relative number of connected methods in the class.

2.1 Relations between Class Components

Individual methods are tied together via two mechanisms. One mechanism, *MIV relations*, involves communication between methods through shared instance variables. The other mechanism, *call relations*, involves the sending of messages directly (or indirectly) from one method to another.

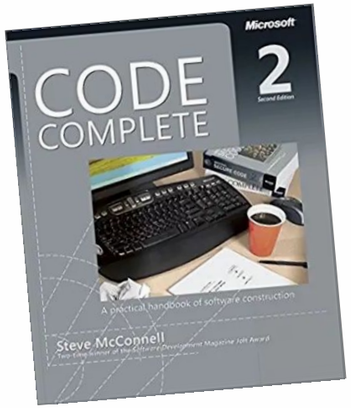
An *MIV relation* is created when two or more class methods read or write to the same class instance variable. We treat shared instance variables as glue that binds the class methods together.

Instance variables used by the server may also used indirectly by the client when one method invokes another through message passing. Thus, a *call relation* can be reflected by the MIV relation; two methods with a call-relation are also connected through the instance variables used by both methods. One method uses the instance variable(s) directly and the other uses the instance variable(s) indirectly through the call relation. There is no MIV relation when a server method neither writes nor reads instance variables. Call relations are usually determined statically due to static dynamic binding in object-oriented software. However, we have observed very few cases where dynamic binding affects class cohesion.

Figure 1 shows a C++ class *Stack* and Figure 2(a) shows the MIV relations among class components of *Stack* in Figure 1. A link between a rectangle and an oval indicates that the method corresponding to the

"If a class is designed in ad hoc manner and unrelated components are included in the class, the class represents more than one concept and does not model an entity. The cohesion value of such a class is likely to be less than 0.5."

— James M. Bieman and Byung-Kyoo Kang. Cohesion and Reuse in an Object-Oriented System. *SIGSOFT Software Engineering Notes*, 20(51):259–262, 1995. doi:10.1145/223427.211856

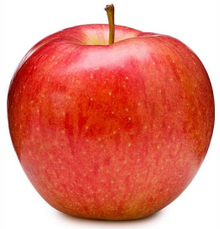


Steve McConnell

“Cohesion refers to how closely all the routines in a class or all the code in a routine support a central purpose—how focused the class is. The ideas of abstraction and cohesion are closely related—a class interface that presents a good abstraction usually has strong cohesion.”

— Steve McConnell. *Code Complete.* Pearson Education, 2004. doi:10.5555/1096143

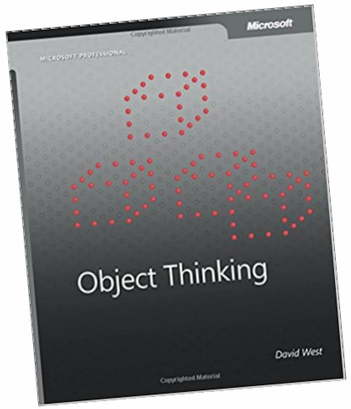
Abstraction

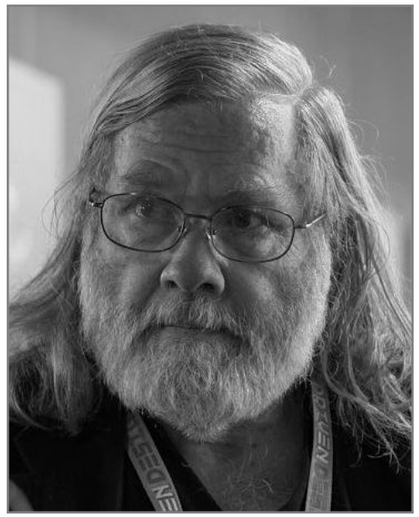


- Color: red
- Weight: 120g
- Price: $0.99

```
var file = {
  path: '/tmp/data.txt',
  read: function() { ... },
  write: function(txt) { ... }
}
```

The slide is taken from the "Pain of OOP" (2023) course.





"Isomorphism of the modules (objects) in problem and solution space is a desirable, in fact essential, quality for software."

— David West. *Object Thinking*. Pearson Education, 2004. doi:10.5555/984130

Inheritance vs. Cohesion

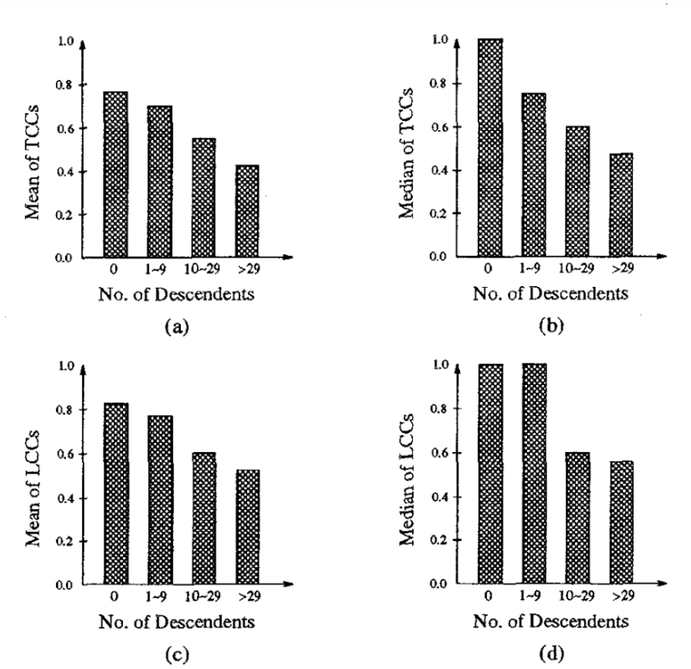


Figure 3: Number of descendents and Class Cohesion

"Our results show that the classes that are heavily reused via inheritance exhibit lower cohesion. We expected to find that the most reused classes would be the most cohesive ones."

Source: James M. Bieman and Byung-Kyoo Kang. Cohesion and Reuse in an Object-Oriented System. *SIGSOFT Software Engineering Notes*, 20(51):259–262, 1995. doi:10.1145/223427.211856

Inheritance is Code Reuse

```
1  class Manuscript {
2    protected String body;
3    void print(Console console) {
4      console.println(this.body);
5    }
6  }
7  class Article
8    extends Manuscript {
9    void submit(Conference cnf) {
10     cnf.send(this.body);
11   }
12 }
```

"The `Article` copies method `print()` and attribute body from the `Manuscript`, as if it's not a living organism, but rather a dead one from which we inherit its parts."

"Implementation inheritance was created as a mechanism for code reuse. It doesn't fit into OOP at all."

Source: Yegor Bugayenko. Inheritance Is a Procedural Technique for Code Reuse. https://www.yegor256.com/160913.html, 9 2016. [Online; accessed 22-09-2024]

Composition over Inheritance

```
class Manuscript
  protected String body;
  void print(Console console)
    console.println(this.body);

class Article
  extends Manuscript
  void submit(Conference cnf)
    cnf.send(this.body);
```

```
class Manuscript
  protected String body;
  void print(Console console)
    console.println(this.body);

class Article
  Manuscript manuscript;
  Article(Manuscript m)
    this.manuscript = m;
  void submit(Conference cnf)
    cnf.send(this.body);
```

Wikipedia: https://en.wikipedia.org/wiki/Composition_over_inheritance

TCC+LCC can be calculated by a few tools:

- jPeek for Java
- C++ — don't know
- Python — don't know
- JavaScript — don't know
- C# — don't know

Bibliography

James M. Bieman and Byung-Kyoo Kang. Cohesion and Reuse in an Object-Oriented System. *SIGSOFT Software Engineering Notes*, 20(51):259–262, 1995. doi:10.1145/223427.211856.

Yegor Bugayenko. Inheritance Is a Procedural Technique for Code Reuse. `https://www.yegor256.com/160913.html`, 9 2016. [Online; accessed 22-09-2024].

Andrew Hunt and Dave Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Pearson Education, 1999. doi:10.5555/320326.

Steve McConnell. *Code Complete*. Pearson Education, 2004. doi:10.5555/1096143.

Plato. Phaedrus (Dialogue), 370 B.C.

David West. *Object Thinking*. Pearson Education, 2004. doi:10.5555/984130.

Edward Yourdon and Larry Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1979. doi:10.5555/578522.