Source-code quality

Part 1. Software Metrics

Andy Kellens

“Not everything that can be counted counts,
and not everything that counts can be counted.”

-- Albert Einstein

Source-code quality

Do you want to write high-quality software?

What is high-quality software?

What is Software Quality ?

Conformance to user requirements

[Cro79]

Achieving excellent levels of fitness for use

[Hum89]

Total customer satisfaction

[IBM]

Degree to which a set of inherent characteristics fulfills requirements

[ISO9001-00]

What is Software Quality ?

- **External Quality**
  - Conformance to specification

    Does it do what it is supposed to do?
  - Correctness & Stability

    Are there many bugs?
- **Internal Quality**
  - Quality characteristics of the source code

    Is the software sustainable?
  - Documentation

    Is the code adequately documented?

What is Software Quality ?

- **External Quality**
  - Conformance to specification
    Does it do what it is supposed to do?
  - Correctness & Stability
    Are there many bugs?
- **Internal Quality**
  - Quality characteristics of the source code
    Is the software sustainable?
  - Documentation
    Is the code adequately documented?

Internal quality impacts external quality + cost

Software erosion

Time

"Over time, the perceived quality of the system will decline"

Time pressure **Lack of knowledge/documentation**

Poor original design **Lack of tool support**

Limitations of technology

Cost of Software Evolution

On average, an evolving software product is rewritten from scratch every 6.8 years [Tamai et al.] (1992)

The legacy crisis [Seacord et al.] (2003)

Proportion of software maintenance costs	Reference & Year
> 90%	[Erlikh] (2000)
75%	[Eastwood] (1993)
> 90%	[Moad] (1990)
60-70%	[Huff] (1990)
60-70%	[Port] (1988)
65-75%	[McKee] (1984)
> 50%	[Lientz & Swanson] ((1981)
67%	[Zelkowitz et al.] (1979)

Cost of Software Evolution

On average, an evolving software product is rewritten from scratch every 6.8 years [Tamai et al.] (1992)

The legacy crisis [Seacord et al.] (2003)

Proportion of software maintenance costs	Reference & Year
> 90%	[Erlikh] (2000)
75%	[Eastwood] (1993)
> 90%	[Moad] (1990)
60-70%	[Huff] (1990)
60-70%	[Port] (1988)
65-75%	[McKee] (1984)
> 50%	[Lientz & Swanson] ((1981)
67%	[Zelkowitz et al.] (1979)

How do we know if our software is sustainable?

How do we keep it sustainable?

Protect quality attributes

Need for tools

- **Manual effort**
  - Review meetings
  - Find quality issues
  - 2 seconds per line of code

    250 KLOC * 2

    500 000 seconds

    140 hours

    18 days

Need for tools that help in assessing quality attributes

Need for tools

- **Manual effort**
  - Review meetings
  - Find quality issues
  - 2 seconds per line of code

    250 KLOC * 2

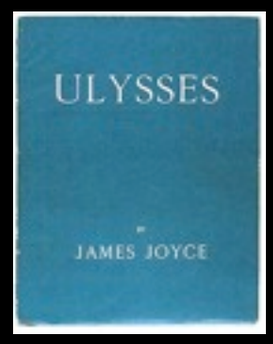
    500 000 seconds

    140 hours

    18 days

+

+



Need for tools that help in assessing quality attributes

Tools for finding quality issues

- **Metrics**
  - Measure various properties of the source code
  - Characterize modularity, complexity, cohesion, coupling, ...
- **Visualizations**
  - Visual representation of the software system
  - Comprehensive view to identify issues
  - Next lecture

Software metrics

Software metrics

- **Measure characteristics of software**
- **Not restricted to source code:**
  - Process metrics
  - Requirements metrics
  - Design metrics
  - ...
- **Our focus: source-code quality metrics**

Know what you measure

- **Precise knowledge of the metrics**
- **Measuring abstract concepts**
- **What does it mean to measure:**
  - Coupling
  - Complexity
  - Size
- **Human interpretation necessary**
- **E.g., when are two classes coupled too tightly?**
- **E.g., is a system with a million lines of code big?**

Measuring code

Source-code metrics

- **Only measure source-code artefacts**
- **Relation with source-code quality**
- **Simple example: SLOC**
  - Source lines of code
  - Simply count each line of code
  - Related with complexity, maintainability, ...

Operating System	SLOC (Million)
Debian 2.2	55-59[3][4]
Debian 3.0	104[4]
Debian 3.1	215[4]
Debian 4.0	283[4]
Debian 5.0	324[4]
OpenSolaris	9.7
FreeBSD	8.8
Mac OS X 10.4	86[5]
Linux kernel 2.6.0	5.2
Linux kernel 2.6.29	11.0
Linux kernel 2.6.32	12.6[6]
Linux kernel 2.6.35	13.5[7]

Source: Wikipedia

Object-oriented code metrics

- Measure properties of OO systems
- Seminal paper: Chidamber and Kemerer (1994)
- Known as CK metrics

476 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 20, NO. 6, JUNE 1994

A Metrics Suite for Object Oriented Design

Shyam R. Chidamber and Chris F. Kemerer

Abstract—Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). In addition, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed. This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Metrics developed in previous research, while contributing to the field's understanding of software development processes, have generally been subject to serious criticisms, including the lack of a theoretical base. Following Wand and Weber, the theoretical base chosen for the metrics was the ontology of Bunge. Six design metrics are developed, and then analytically evaluated against Weyuker's proposed set of measurement principles. An automated data collection tool was then developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and suggest ways in which managers may use these metrics for process improvement.

Index Terms—CR categories and subject descriptors: D.2.8 [software engineering]: metrics; D.2.9 [software engineering]: management; F.2.3 [analysis of algorithms and problem complexity]: tradeoffs among complexity measures; K.6.3 [management of computing and information systems]: software management. General terms: Class, complexity, design, management, measurement, metrics, object orientation, performance.

I. Introduction

IT has been widely recognized that an important component of process improvement is the ability to measure the process. Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This emphasis has had two effects. The first is that this demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). Second, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed.

This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Previous research on software metrics, while contributing to the field's understanding of software development processes, have generally been subject to one or more types of criticisms. These include: lacking a theoretical basis [41], lacking in desirable measurement properties [47], being insufficiently generalized or too implementation technology dependent [45], and being too labor-intensive to collect [22].

Following Wand and Weber, the theoretical base chosen for the OO design metrics was the ontology of Bunge [5], [6], [43]. Six design metrics were developed, and analytically evaluated against a previously proposed set of measurement principles. An automated data collection tool was then developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and to suggest ways in which managers may use these metrics for process improvement.

The key contributions of this paper are the development and empirical validation of a set of theoretically-grounded metrics of OO design. The rest of this paper is organized as follows. The next section presents a brief summary of the research problem, followed by a section describing the theory underlying the approach taken. Then Weyuker's list of software metric evaluation criteria is presented, along with a brief description of the empirical data collection sites. The Results section presents the metrics, their analytical evaluation, the empirical data and a managerial interpretation of the data for each metric. Some concluding remarks are presented in the final section.

II. Research Problem

There are two general types of criticisms that can be applied to current software metrics. The first category are those theoretical criticisms that are leveled at conventional software metrics as they are applied to traditional, non-OO software design and development. Kearney, et al. criticized software complexity metrics as being without solid theoretical bases and lacking appropriate properties [21]. Vessey and Weber also commented on the general lack of theoretical rigor in the structured programming literature [41]. Both Prather and Weyuker proposed that traditional software complexity metrics do not possess appropriate mathematical properties, and consequently fail to display what might be termed normal predictable behavior [34], [47]. This suggests that software

Manuscript received February 17, 1993; revised January, 1994. recommended by S. Zweben. This research was supported in part by the M.I.T. Center for Information Systems Research (CISR), and the cooperation of two industrial organizations who supplied the data.

The authors are with the Massachusetts Institute of Technology, E53-315, 30 Wadsworth Street, Cambridge, MA 02139 USA; e-mail: shyam@athena.mit.edu or ckemerer@sloan.mit.edu.

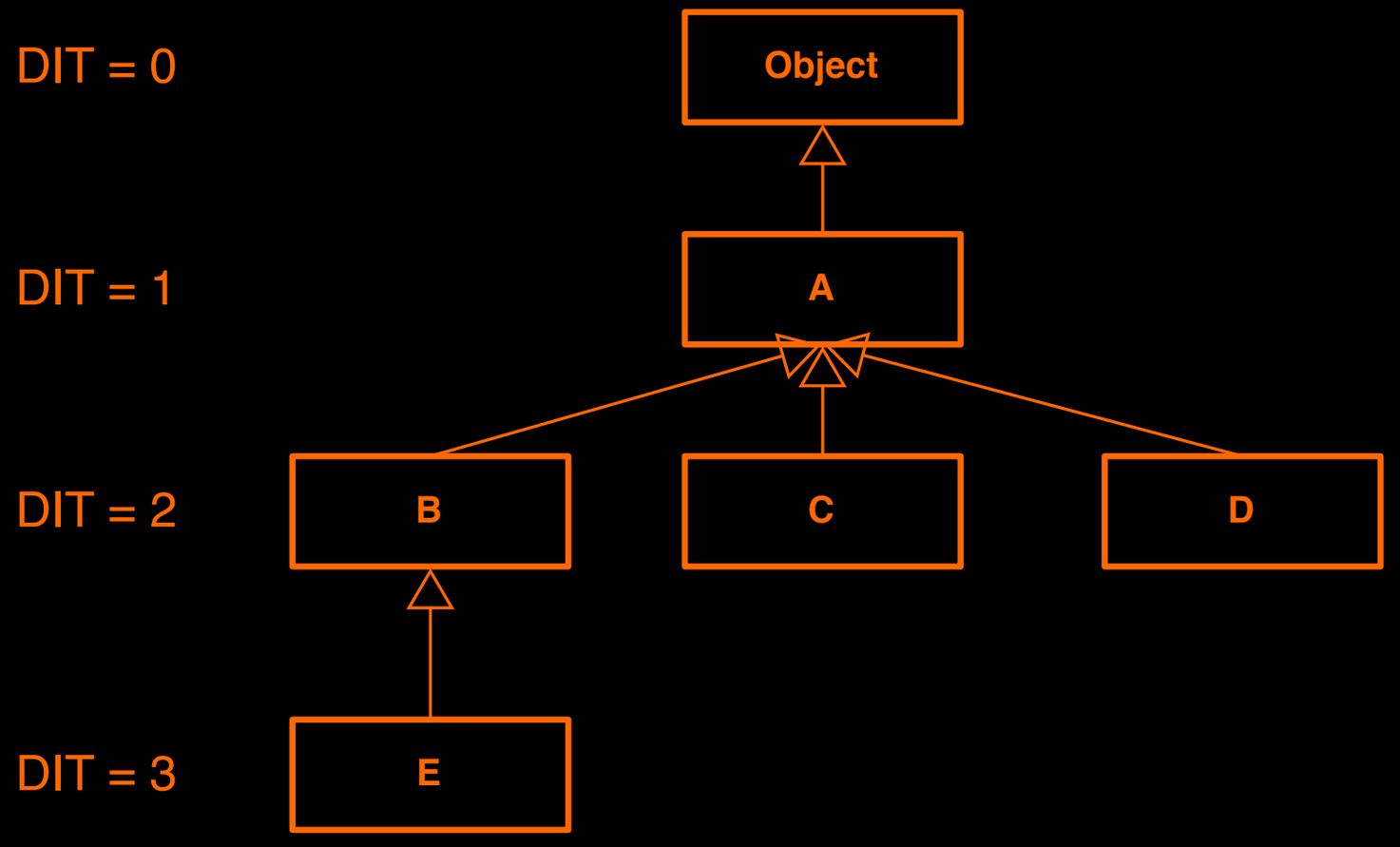
IEEE Log Number 9401707.

WMC: Weighted methods per class

- **Sum of complexity of all methods in a class**
- **How measure complexity:**
  - Use McCabe (see later)
  - Assign value of 1 per method
- **Low WMC might indicate data classes**
- **High WMC might indicate god classes**

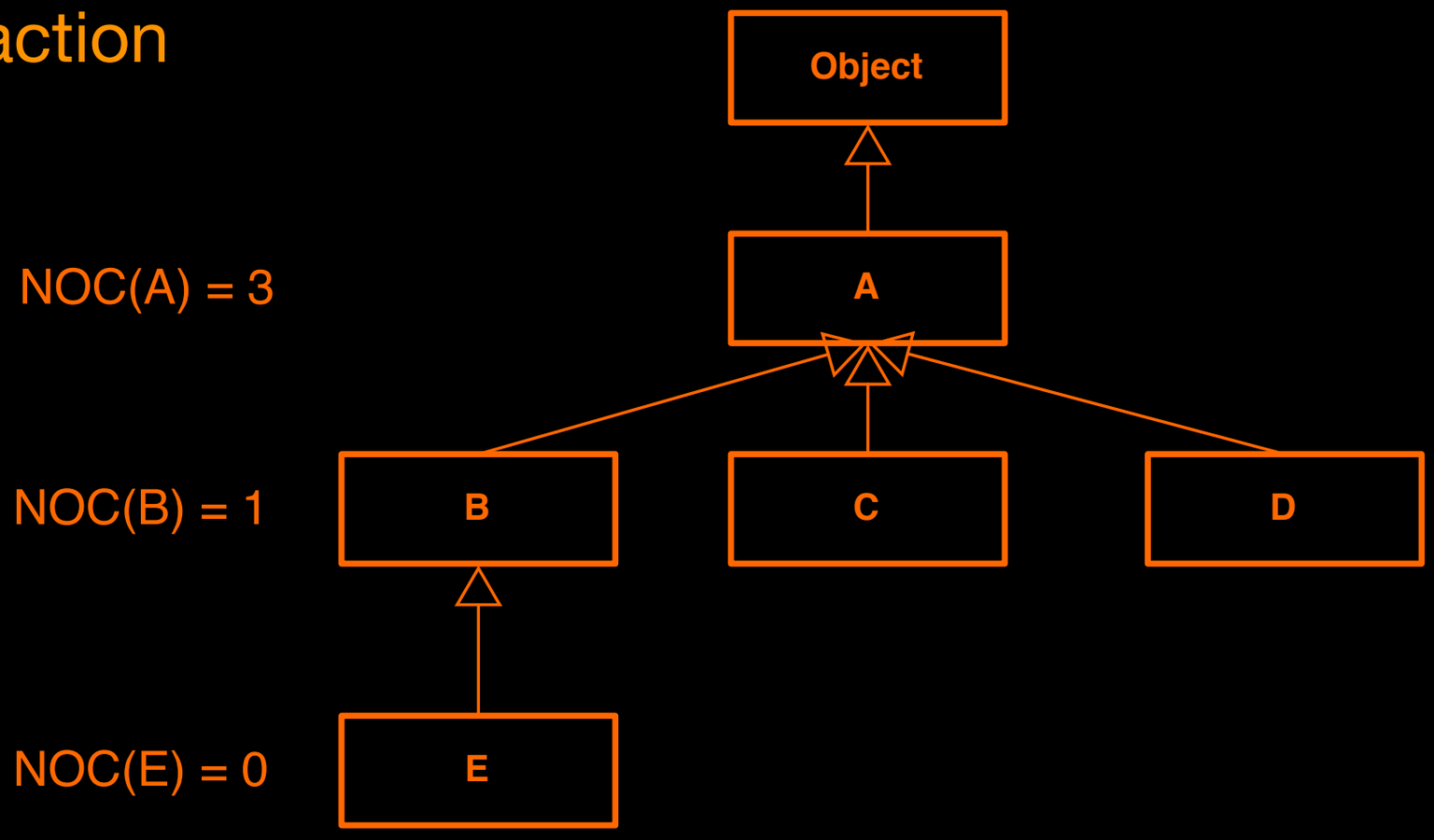
DIT: Depth of Inheritance Tree

- Inheritance levels from the object hierarchy top
- In Java, Smalltalk at least 1
- Low value: poor OO?
- High value: overly complex hierarchy?



NOC: Number of Children

- **Count the direct subclasses of a class**
- **High NOC:**
  - Lots of reuse
  - Improper use of subclassing
  - Improper abstraction



RFC: Reponse for a Class

- Number of different methods that can be invoked when an instance of a class receives a message
- In other words, sum of the number of methods in the class and the number of distinct, remote methods called from within those methods
- RFC is *not* transitive (RFC' is transitive variant)
- Measures potential communication between classes
- High RFC might indicate higher chance of faults

Measuring coupling and cohesion

Coupling

- **Dependency between entities (classes, modules, ...)**
- **Strive for loosely coupled entities**
- **Example metric: CBO (CK metric)**
  - Coupling between Object Classes
  - # of ***unique*** classes to which a class is coupled
  - Use of fields and methods of other classes
  - In the case of a polymorphic call: count all candidates
  - Two-directional?
  - Inheritance?

Example

```
class Address
    { //...
    }

class Person
{
    String name;
    Integer age;
    Address address;

    public String getName() {return name;}
    public void setName(String n) {name = n;}

    public Integer getAge() {return age;}
    public void setAge(Integer a) {age = a;}

    public Integer getAddress() {return address;}
    public void setAddress(Address a) {address = a;}}


class ClientCode
{
    Person person;
    void doSomething()
        {
            person = new Person();
            person.setAge(22);
            person.setName("Test");
        }
    void shipPackage() {
        Address shippingAddress = person.getAddress();
        Address fromAddress = new Address();
        //...
}}
```

CBO(Address) = 0

CBO(Person) = 1

CBO(ClientCode) = 2

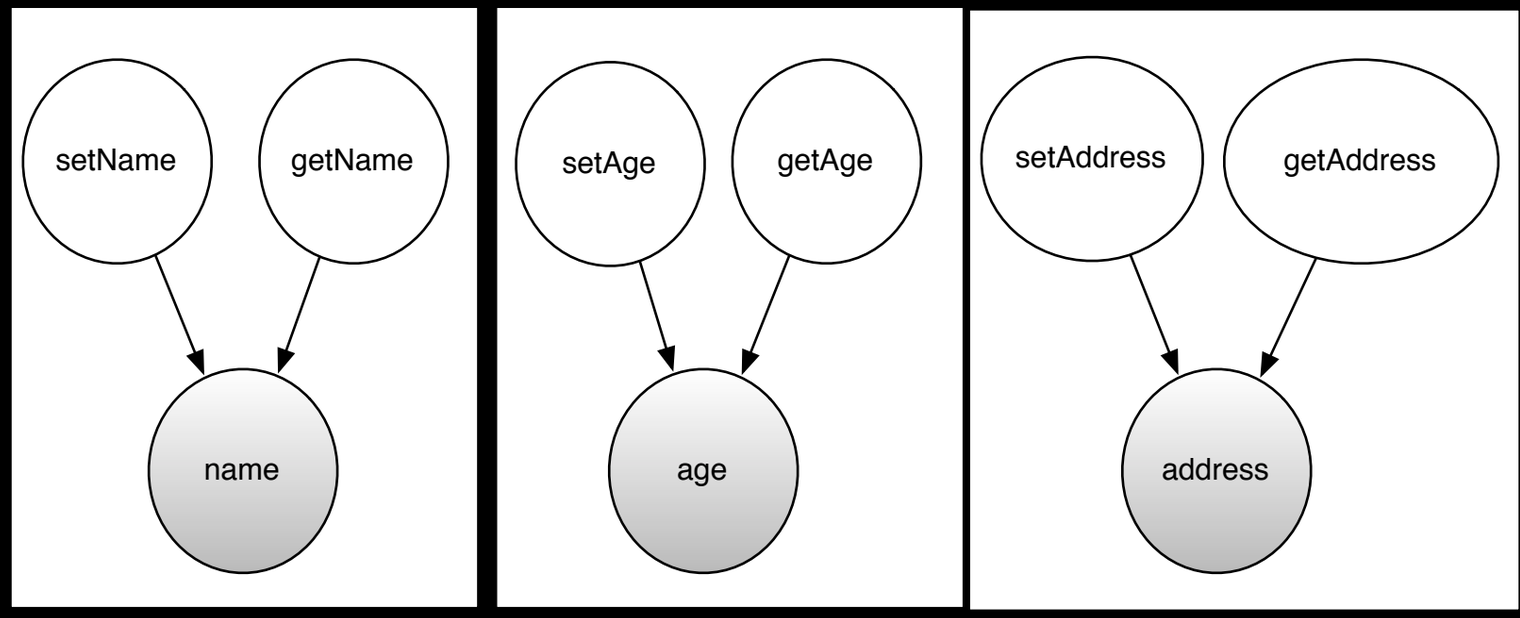
Cohesion

- **Within one module (class, package, ...) have high cohesion**
- **Example metric: LCOM4 (Hitz & Montazeri)**
  - Lack of Cohesion of Methods
  - Two methods of the same class are related if:
    - They call each other
    - They access the same field
  - For all methods in a class, create a graph showing the connections between methods
  - LCOM = number of connected sub-graphs
  - Ideal case: LCOM = 1

Example

```
class Person
{
    String name;
    Integer age;
    Address address;

    public String getName() {return name;}
    public void setName(String n) {name = n;}

    public Integer getAge() {return age;}
    public void setAge(Integer a) {age = a;}

    public Integer getAddress() {return address;}
    public void setAddress(Address a) {address = a;}}
```

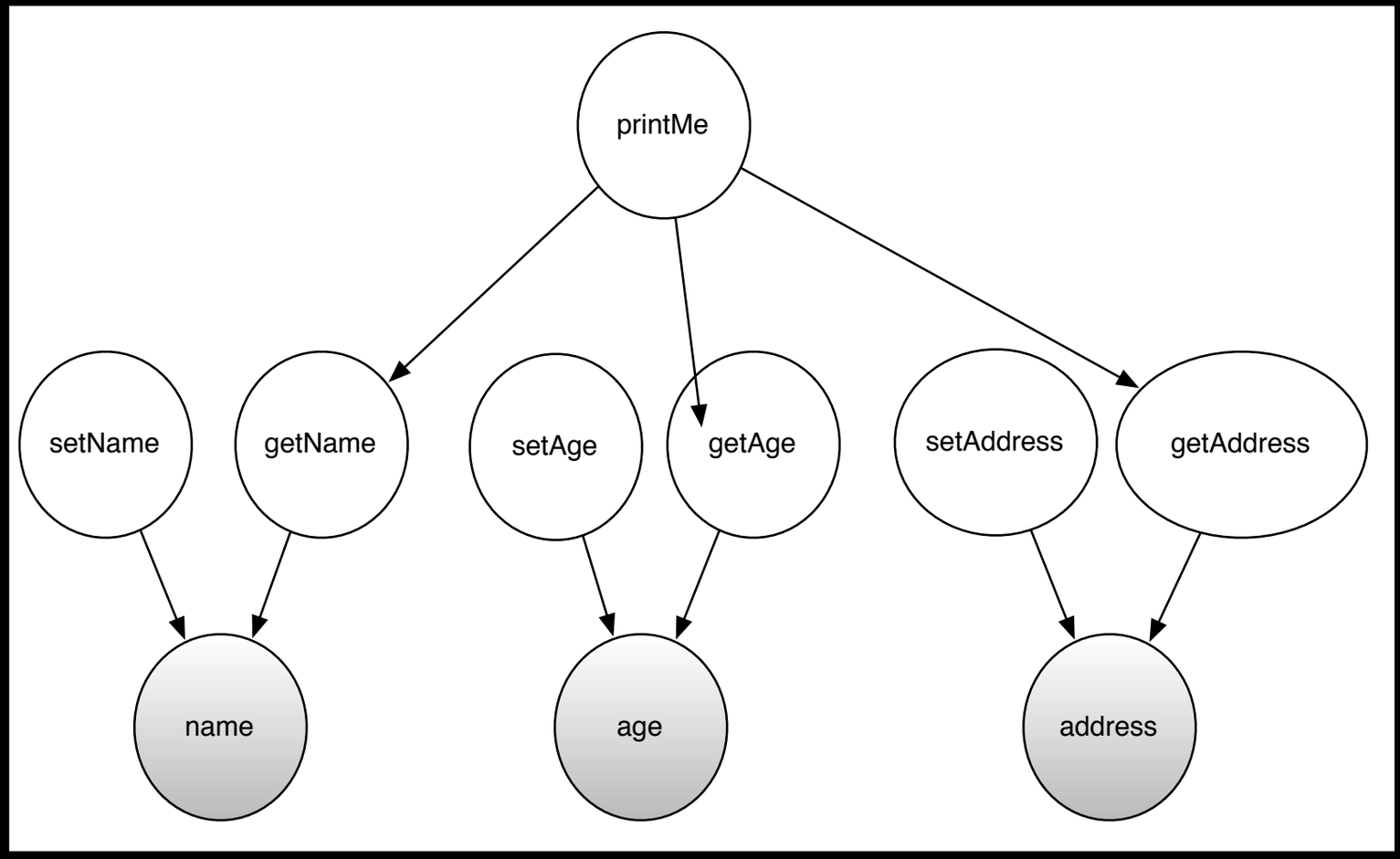
LCOM4(Person) = 3



Example 2

```
class Person
{
   String name;
   Integer age;
   Address address;

   public String getName() {return name;}
   public void setName(String n) {name = n;}

   public Integer getAge() {return age;}
   public void setAge(Integer a) {age = a;}

   public Integer getAddress() {return address;}
   public void setAddress(Address a) {address = a;}

   public void printMe()
   {
      System.out.println(this.getName());
      System.out.println(this.getAge());
      System.out.println(this.getAddress());
}}
```

LCOM4(Person) = 1



Measuring complexity

Complexity

Degree of incidental complexity of a program, by construction.

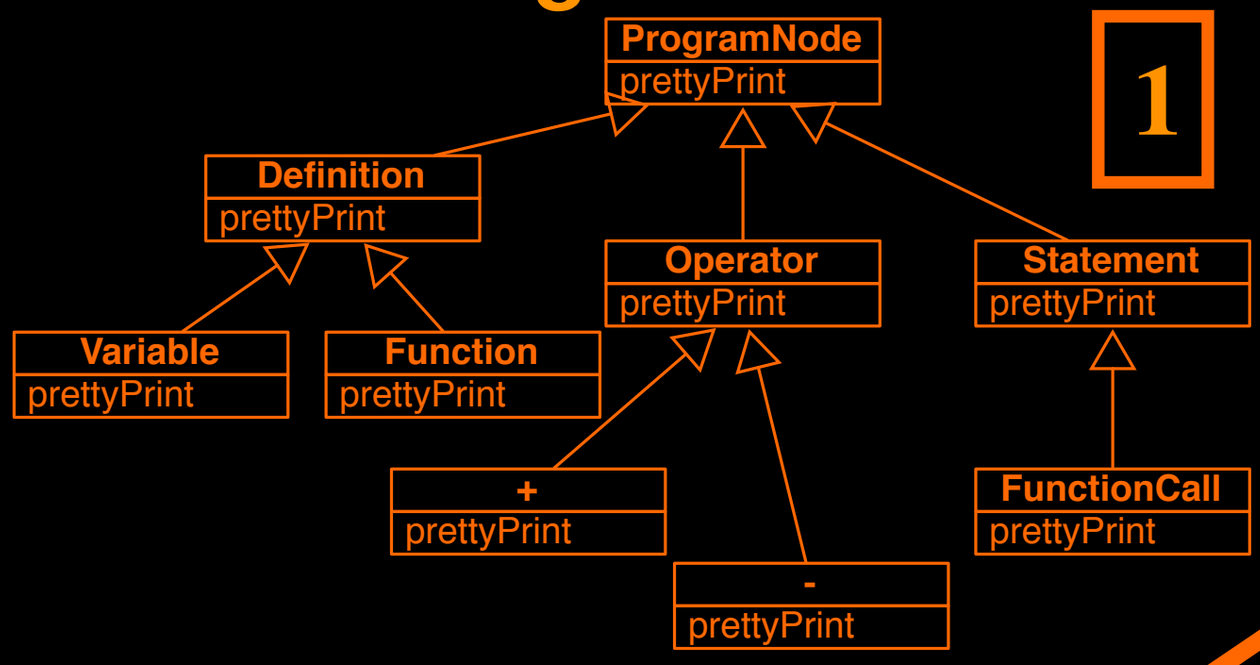
- **Incidental vs. Inherent (Accidental vs. Essential)**
  - Software that deals with a complex domain is inherently complex
  - Incidental complexity is due to the design and implementation

"No Silver Bullet -- Fred Brooks" (1987)

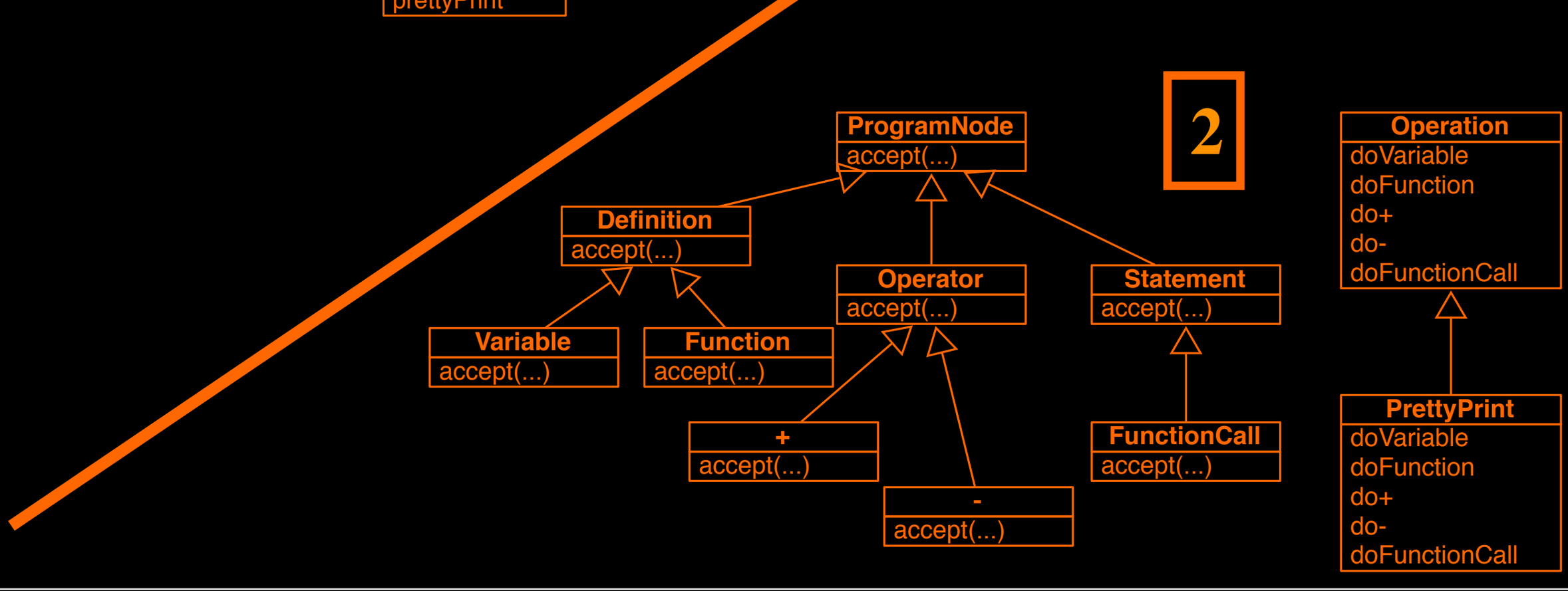
Incidental complexity is to be avoided.
Complexity measures often show contrasts in the code: i.e. problem areas.

Problem

- **Which design is best?**



1



2

Over-design

- **Introduction of accidental complexity in the design**
- **KISS principle**
  - Keep It Simple Stupid
  - Choose a simple solution that suffices
  - Refactoring into something more complex when **necessary**
- More accidental complexity
  - more difficult to reuse, extend, maintain, ...

Measuring complexity (at a low level)

- **Measure:**
  - complexity of individual entities
  - aggregation of the complexity of modules/files/classes/...
- **High complexity not necessary a problem**
- **However, is the complexity where you expect it to be?**

McCabe cyclomatic complexity

- Thomas McCabe (in 1976)
- Idea: count the number of linearly independent path's through a piece of source code
- How to compute: based on a control flow graph

Complexity M = E - N + 2P

with:

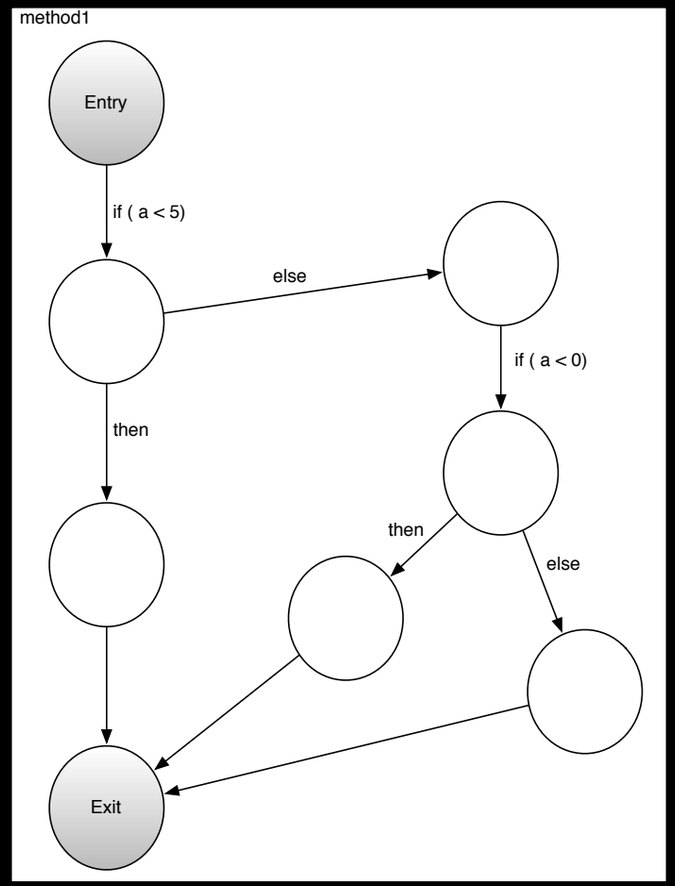
E = number of edges

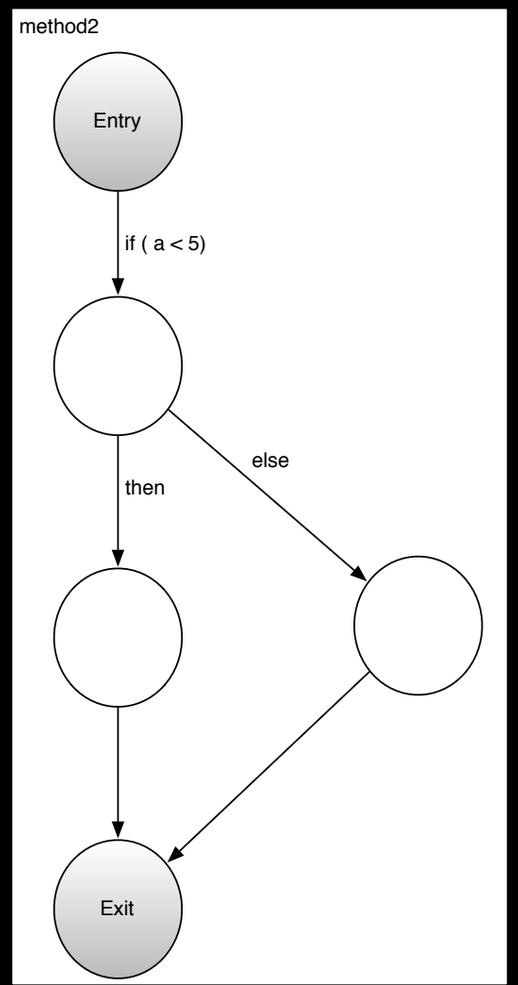
N = number of nodes

P = number of connected components

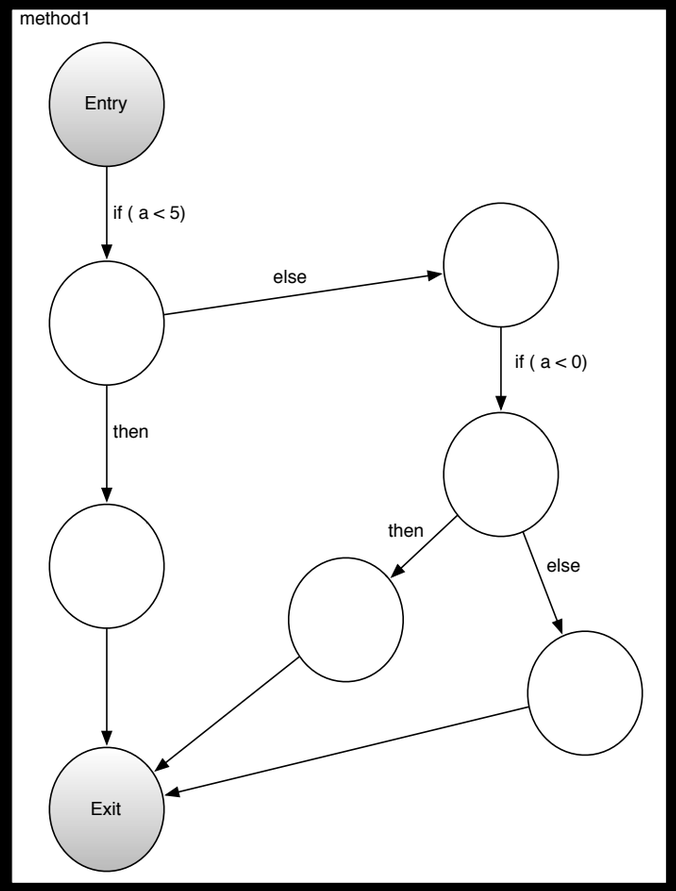
Example

```
public void method1(Integer a) {
    if (a < 5)
        {...}
        else {
            if (a < 0)
             {...}
            else{...};
            };
    System.out.println("Done");
}

public void method2(Integer a) {
    if (a < 5)
        {...}
        else {...};
    System.out.println("Done too");
}
```

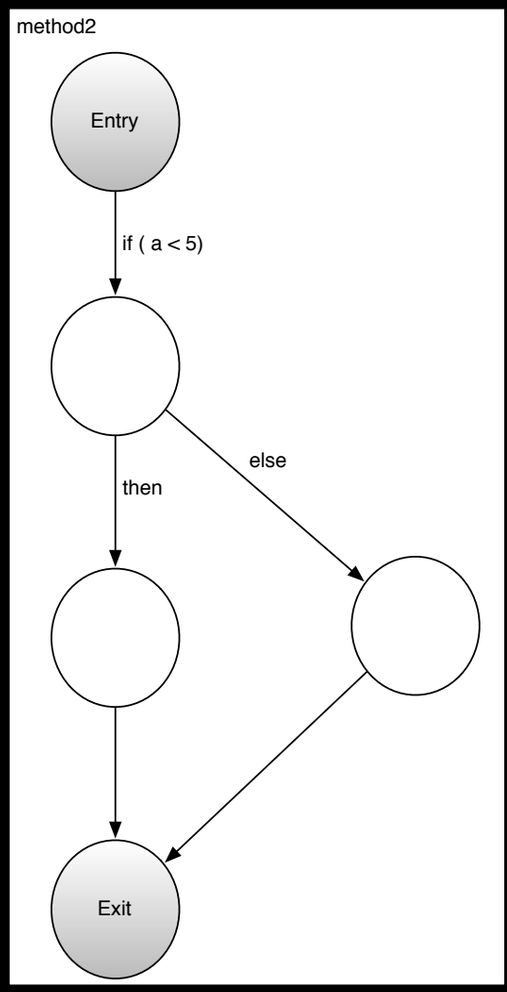




Example



$$M = E - N + 2P$$



$$M(method1) = 9 - 6 + 2 * 1$$
$$= 5$$

$$M(method2) = 5 - 3 + 2 * 1$$
$$= 4$$

Interpreting metrics

Interpreting metrics

- **Metrics can be useful:**
  - Assessing quality
  - Finding places to improve
- **However:**
  - How well do they reflect reality?
  - Operations on metrics
  - Knowledge needed about the system

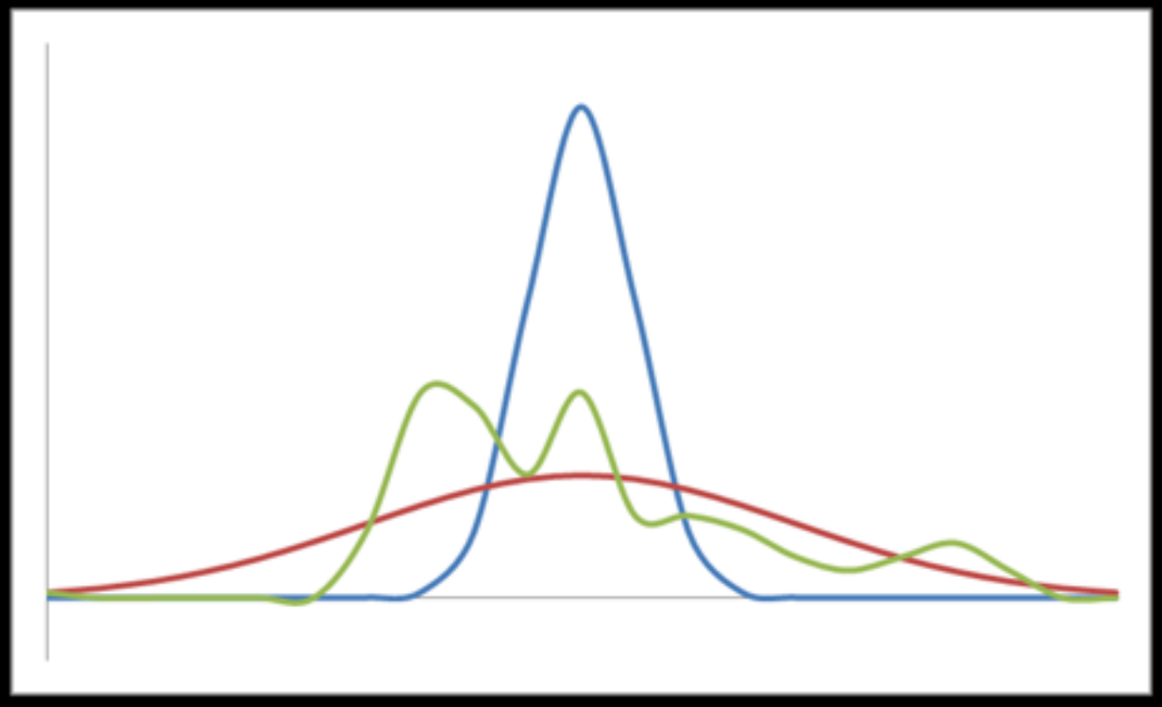
How well do metrics reflect reality?

- **Quality attributes are rather abstract concepts**
- **E.g., complexity of a method**
  - McCabe cyclomatic complexity
  - Only one aspect of complexity
  - Are methods with a high cyclomatic complexity really complex to understand/maintain?
- **When using metrics**
  - Important to know **exactly** what the metric computes
  - Human interpretation is needed

Operations on metrics

- **Depending on scope not all kinds of calculations are allowed**
- **However, even when allowed be careful!**
- **For example:**

  - "The average McCabe complexity of methods in my system is 4"



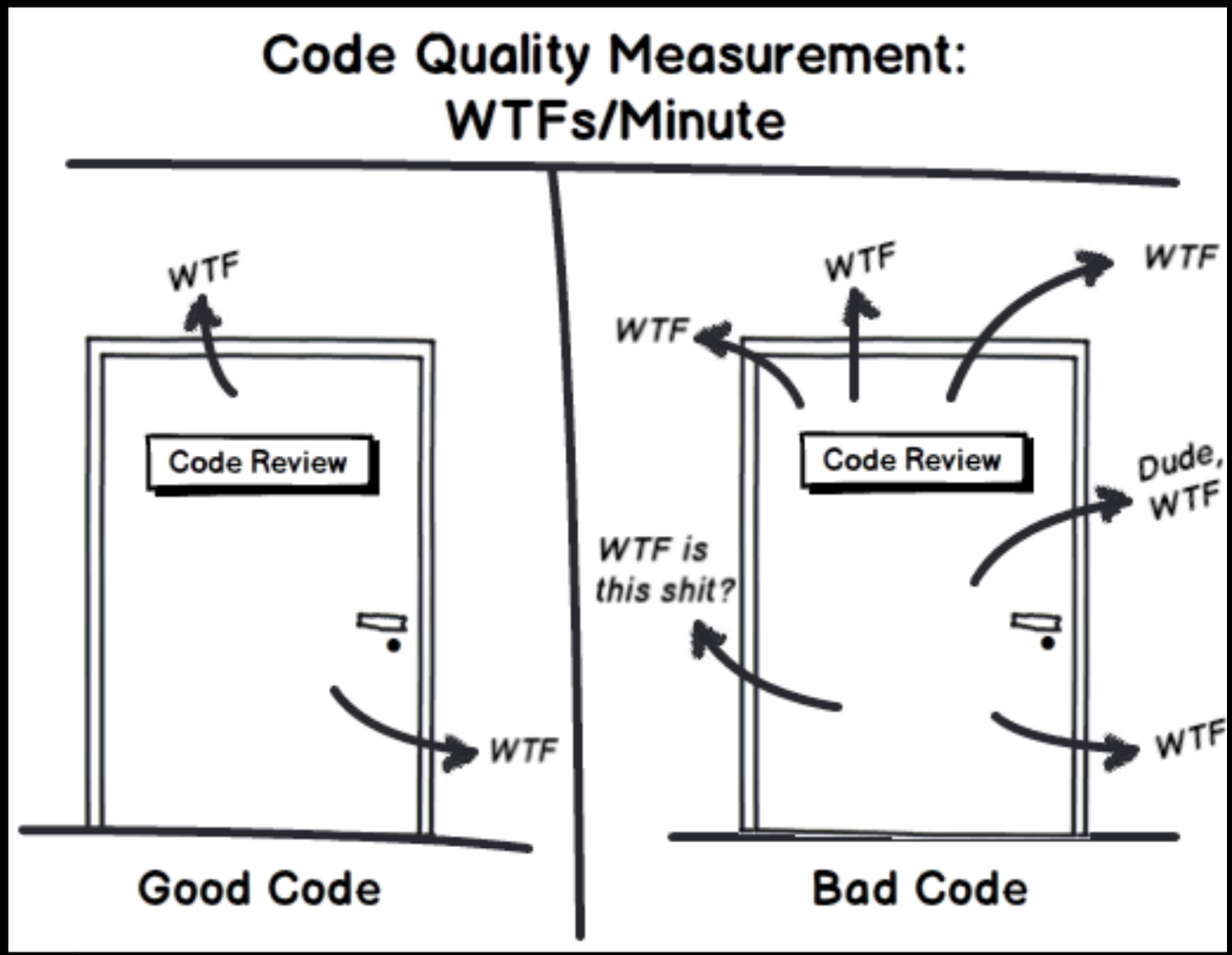
Knowledge needed about the system

- "Poor" metrics do not necessary indicate problems
- Depending on the system
- E.g., is the coupling where you expect it? Is the complexity of the system located where you expect it?

▼McCabe Cyclomatic Complexity (avg/max per method)		1.321	0.71
▼Node.java		2.143	0.99
▼Node		2.143	0.99
delete	4		
optimize	3		
search	2		
insert	2		
replace	2		
Node	1		
list	1		
▶Leaf.java		1.143	0.35
▶Tree.java		1	0
▶EditorExample.java		1	0
▶Handler.java		1	0
▶Empty.java		1	0

Conclusions

- **Metrics to measure attributes of software**
- **Assess quality**
- **However:**
  - Know what is measured
  - Know how to interpret
  - Expert knowledge about the system is needed


Code Quality Measurement:
WTFs/Minute
WTF
Code Review
WTF
Good Code
WTF
WTF
WTF
Code Review
Dude,
WTF
WTF is
this shit?
WTF
Bad Code