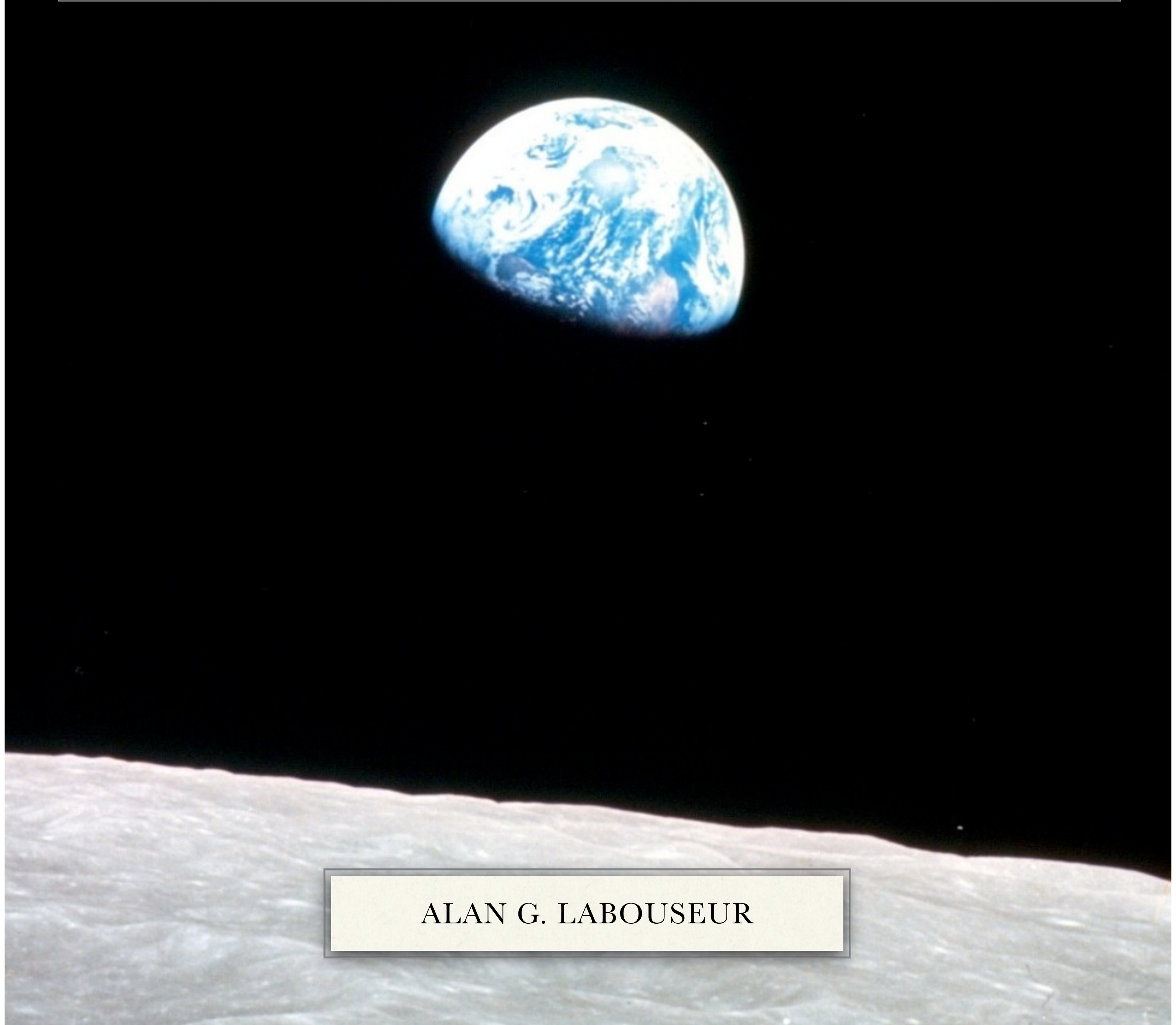
More than complexity. More than feel.

CODE RECKON



ALAN G. LABOUSEUR

Think Different

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. - *C.A.R. Hoare, From his 1980 Turing Award Lecture*

Every truth passes through three stages before it is recognized. In the first, it is ridiculed, in the second it is opposed, in the third it is regarded as self-evident. - *Arthur Schopenhauer*

It don't mean a thing if it ain't got that swing. - *Duke Ellington*

Elegant Simplicity

Thousands of years ago Lao Tzu wrote, in the Tao Te Ching, "The people themselves need to learn simplicity." In fact, elegant simplicity the core of Lao Tzu's teachings and of Taoism itself.

Simple means not complex.

Years later, the great Fred Brooks wrote about essential and accidental complexity in software, basing his ideas of the essential and the accidental on Aristotle's "Categories". Complexity can be pruned to its bare essentials for easy comprehension.

Elegant simplicity is not-complex and easy-comprehension in balance, as yin and yang.

Complexity in software development is well understood and there are many means to measure it. Comprehension, the second component of elegant simplicity, is not very well understood and there are few means of measure here. This is where we'll spend our time in this report.

Chapter 1

BACKGROUND

ALAN G. LABOUSEUR

- *BS, Computer Science, Marist College - 1990*
- *MS, Computer Science, Pace University - 1995*
- *PhD in progress, Computer Science, SUNY Albany*

- Entrepreneur 1990 - 2006, 2013 - ?
- Adjunct faculty in Computer Science and Information Technology 1995-2006
- Full-time faculty in Computer Science and Information Technology 2003-present

SECTION 1

History

I have been a professional software developer since 1990. In that time I have owned three consulting companies, all service businesses built around custom database and software development.

I have written software for . . .

- large companies such as Scholastic, GE Capital, IBM, Pearson, East New York Savings Bank, Pricewaterhouse Coopers, Lincoln Medical and Mental Health Center, and Viacom;
- specialized firms like Keymark Financial Services, Dick Clark Corporate Productions and the Co-Ed Uniform Company;
- and startups like Classroom Authors, GotSchool, and Direct Response Marketing.

In the public sector I've consulted to the New York State Office of Mental Health, New York University, New York Institute of Finance, and the Department of Child Welfare.

Internationally I helped the Global Association of Risk Professionals deal with information issues in the US and Europe and provided strategic counseling and implementation services to BioPharma Greenhouse, a biotech firm operating in the US and the People's Republic of China

These projects were spread over many industries, including but not limited to the following:

- Finance and Banking
- Risk Management
- Biotech and Pharmaceutical
- Commercial Real Estate
- International Trade
- Event Management
- Healthcare Administration
- Publishing
- Higher Education
- Marketing

In all this time I've personally designed, developed, debugged, and documented a lot of source code in a lot of systems for a lot of companies. Some, though certainly not all, of the details are shown in table 1.

I've also seen and modified a lot of other people's code on my journeyman's path... code of all types: the good, the bad, and the ugly. I've developed a good "feel" for what quality code looks and smells like.

I've spent the past nine years teaching software development in addition to my consulting. In that time I've noticed that I can look at my students' source code and make a fast intuitive judgement about its quality (whatever that is) that turns out to be right more often than not. My "feel" for how comprehendible source code looks is quite good. But "feel" doesn't scale, and it's too subjective to be scientific.

Company	*Application*	*Approx. Lines of Code*	*Approx. Dates in use*
IBM	MMS - the Meeting Management System	10 K	1990 - 1994+
New York Institute of Finance	SMS - the Student Management System	24 K	1993 - 2001
Simon & Schuster	ISBN Generator and Sales Catalog database publisher	4 K	1993 - 1995+
Viacom	AMPS - the Advanced Manuscript Processing System	4 K	1994 - 1997
GE Capital	Quick Quote	5 K	1997 - 2000
PriceWaterhouse Coopers	Risk Perceptions	6.5 K	1999 - 2000
NYS Office of Mental Health	PSMS	5.2 K	1996 - 2000
Co-Ed Uniform Company	OpMan and WebMan	11.5 K	1994 - present
Scholastic	Teacher's Store E-commerce database	2 K	2006 - 2009+
Approximate total designed, developed, debugged, and documented lines of code:		~ 72.2 K	

Table 1

SECTION 2

Goals

More Than Complexity

My view of source code quality is more than (and also *other* than) complexity alone. Source code complexity is well-understood and not terribly interesting to me in this context. There's more to it than that. My experience tells me that code *feel* and *comprehendability* -- the amount of effort required to understand a piece of code -- also relate to measures of code quality; the yang to complexity's yin.

More Than Feel

"Feel" doesn't scale. My goal is to enhance my own understanding of *why* I feel certain ways about source code. I hope to note some objective, measurable, and detectable traits of source code quality and "comprehendability" along the way.

Measurable and Detectable Quality Traits

To this end, I'll look at some important (and directed) readings in software development and computer science with an eye towards identifying measurable and detectable traits of source code quality outside of complexity.

In addition to identifying these measurable and detectable traits, I'll describe some possible ways one might implement these measures and detections in a compiler, profiler, or some supplemental analysis system.

Finally, I'll note other ideas that come to me along this journey. These may point towards additional or other directions for future investigation.

Chapter 2

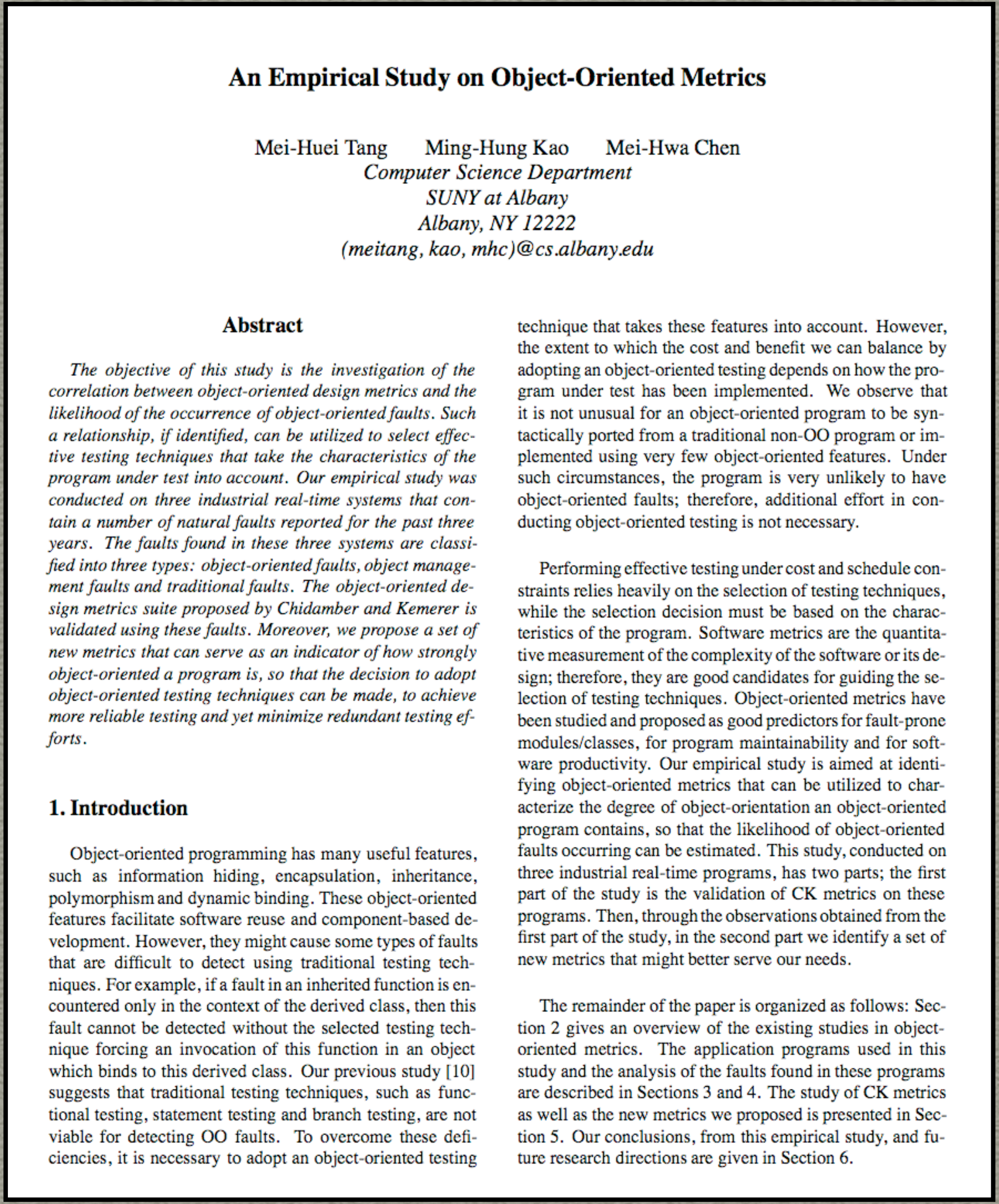
AN EMPIRICAL STUDY ON OBJECT-ORIENTED METRICS

An Empirical Study on Object-oriented Metrics

by Mei-Huei Tang, Ming-Hung Kao, Mei-Hwa Chen
[meitang, kao, mhc]@cs.albany.edu
Computer Science Department, SUNY at Albany, Albany, NY 12222

This study, conducted on three industrial real-time systems that contained a number of natural faults, identified object-oriented faults, object management faults, and traditional faults.

After validating CK metrics with those faults, the authors propose a set of new metrics to measure how strongly object-oriented a program is.



An Empirical Study on Object-Oriented Metrics

Mei-Huei Tang Ming-Hung Kao Mei-Hwa Chen
Computer Science Department
SUNY at Albany
Albany, NY 12222
(meitang, kao, mhc)@cs.albany.edu

Abstract

The objective of this study is the investigation of the correlation between object-oriented design metrics and the likelihood of the occurrence of object-oriented faults. Such a relationship, if identified, can be utilized to select effective testing techniques that take the characteristics of the program under test into account. Our empirical study was conducted on three industrial real-time systems that contain a number of natural faults reported for the past three years. The faults found in these three systems are classified into three types: object-oriented faults, object management faults and traditional faults. The object-oriented design metrics suite proposed by Chidamber and Kemerer is validated using these faults. Moreover, we propose a set of new metrics that can serve as an indicator of how strongly object-oriented a program is, so that the decision to adopt object-oriented testing techniques can be made, to achieve more reliable testing and yet minimize redundant testing efforts.

1. Introduction

Object-oriented programming has many useful features, such as information hiding, encapsulation, inheritance, polymorphism and dynamic binding. These object-oriented features facilitate software reuse and component-based development. However, they might cause some types of faults that are difficult to detect using traditional testing techniques. For example, if a fault in an inherited function is encountered only in the context of the derived class, then this fault cannot be detected without the selected testing technique forcing an invocation of this function in an object which binds to this derived class. Our previous study [10] suggests that traditional testing techniques, such as functional testing, statement testing and branch testing, are not viable for detecting OO faults. To overcome these deficiencies, it is necessary to adopt an object-oriented testing technique that takes these features into account. However, the extent to which the cost and benefit we can balance by adopting an object-oriented testing depends on how the program under test has been implemented. We observe that it is not unusual for an object-oriented program to be syntactically ported from a traditional non-OO program or implemented using very few object-oriented features. Under such circumstances, the program is very unlikely to have object-oriented faults; therefore, additional effort in conducting object-oriented testing is not necessary.

Performing effective testing under cost and schedule constraints relies heavily on the selection of testing techniques, while the selection decision must be based on the characteristics of the program. Software metrics are the quantitative measurement of the complexity of the software or its design; therefore, they are good candidates for guiding the selection of testing techniques. Object-oriented metrics have been studied and proposed as good predictors for fault-prone modules/classes, for program maintainability and for software productivity. Our empirical study is aimed at identifying object-oriented metrics that can be utilized to characterize the degree of object-orientation an object-oriented program contains, so that the likelihood of object-oriented faults occurring can be estimated. This study, conducted on three industrial real-time programs, has two parts; the first part of the study is the validation of CK metrics on these programs. Then, through the observations obtained from the first part of the study, in the second part we identify a set of new metrics that might better serve our needs.

The remainder of the paper is organized as follows: Section 2 gives an overview of the existing studies in object-oriented metrics. The application programs used in this study and the analysis of the faults found in these programs are described in Sections 3 and 4. The study of CK metrics as well as the new metrics we proposed is presented in Section 5. Our conclusions, from this empirical study, and future research directions are given in Section 6.

EXISTING METRICS

Background

- "Software metrics are the quantitative measurement of the complexity of the software or its design." Actually, there are many other varieties of software metrics beyond complexity. That's my main point in these readings and this report.

- Thomas McCabe's 1976 cyclomatic complexity metric [2] is a popular measure of complexity based on program structure. (A similar set of textual measures created by Halsted in 1977 are popular as well.) Cyclomatic complexity is based on analysis of a program's CFG (control flow graph, not context-free grammar). Note my suggestions for other uses of the control flow graph in calculating non-complexity measures of source code quality.

CK Metrics

- Chidamber and Kemerer's complexity CK metrics [3] have been validated in this and other works (Basili, Briand, and Melo; Li and Henry; others).

- CK metrics, in summary:
  - Weighted Methods per Class (WMC)
  - Class Inheritance Tree Depth
  - Number of Child Classes
  - Coupling Between Objects (this should read "among" objects since it could involve three or more.)
  - Response for a class (RFC)

Experiment

- The authors analyzed three years worth of trouble reports to classify the faults into object-oriented and non-object-oriented faults. (Of the object-oriented faults, the second of the two categories they noted was memory-related. Most of these memory-related faults disappear in a garbage-collected language/environment.)

- They conclude that among the CK metrics noted here, only WMC and RFC turn out to be significant.

❖ *IDEA:*

I wonder what the outcome would have been if the WMC metric had excluded private classes by assigning them a weight of 0 instead of counting them (with weight of 1) along with everything else.

NEW METRICS

Background

- The authors invent some new metrics to try out on the same software and measure against CK metrics.
  - Inheritance Coupling: the number of parent classes to which a given subclass is coupled via variables or methods.
  - Coupling Between Methods: the number of functional dependancies among inherited methods and new or redefined methods. (This seems a lot like Coupling Between Objects, since an object is data plus some methods defined on it.)
  - Number of Object / Memory Allocations: a count of the statements that allocate new objects in a class.
  - Average Method Complexity: the average method size for each class (as a count of source lines, sum of bytecode size, character count, what?).

Results

- Average Method Complexity is shown to be a significant predictor of fault-prone classes. This makes a lot of sense because the simplest source code measure, the size of the code (in number of functional lines) is widely accepted as a fault indicator. (We'll see this elsewhere in this report.)

❖ *Traits We Can Measure or Detect, and How*

- All of the metrics noted in this paper can be calculated by various kinds of source code analysis. Most can be done in the front-end of a compiler (lex, parse, semantic analysis). Others can be done in the back-end via data flow analysis of the control-flow graph. Many IDEs do a lot of these already. Eclipse, Visual Studio, IntelliJ, NetBeans, and others.

Chapter 3

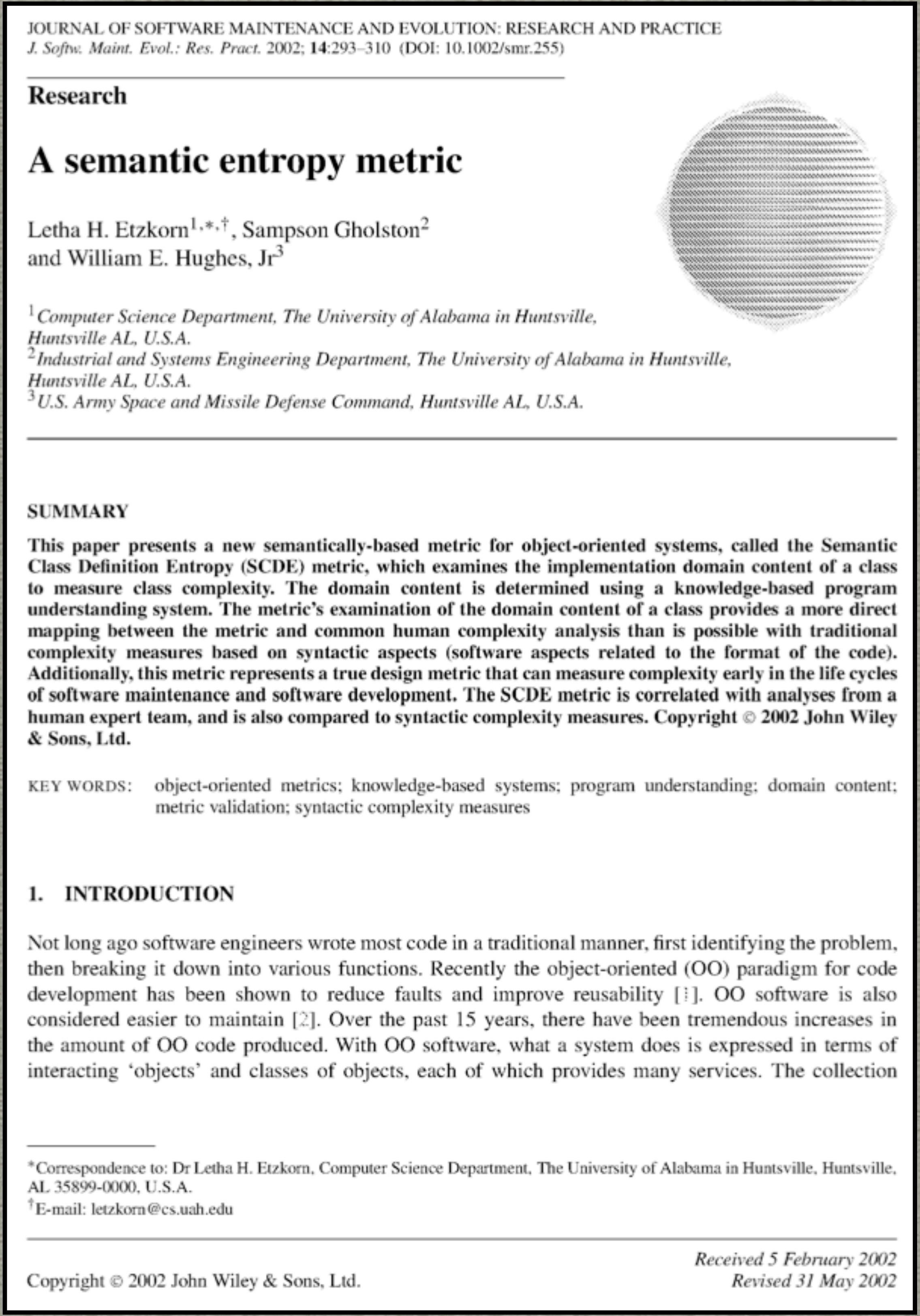
A SEMANTIC ENTROPY METRIC

A Semantic Entropy Metric

by Letha H. Etzkorn, Sampson Gholston, and William Hughes, Jr.

University of Alamaba in Hunstville, AL

This 2002 paper introduces a semantically-based metric for Object-oriented systems called the Semantic Class Definition Entropy (SCDE) Metric. It seeks to go beyond traditional complexity metrics that look only at the structure and/or text of the code (e.g., the work of McCabe, Halsted and that crew) and incorporate domain content through considering internal-documentation-related information from comments and identifiers in the source code. The idea is to get at the human-level complexity ("comprehendability") of the task the program is performing as described by the programs own internal documentation (comments and identifiers).



JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2002; **14**:293–310 (DOI: 10.1002/smr.255)

Research

A semantic entropy metric

Letha H. Etzkorn[1,*,†], Sampson Gholston[2] and William E. Hughes, Jr[3]

[1] *Computer Science Department, The University of Alabama in Huntsville, Huntsville AL, U.S.A.*
[2] *Industrial and Systems Engineering Department, The University of Alabama in Huntsville, Huntsville AL, U.S.A.*
[3] *U.S. Army Space and Missile Defense Command, Huntsville AL, U.S.A.*

SUMMARY

This paper presents a new semantically-based metric for object-oriented systems, called the Semantic Class Definition Entropy (SCDE) metric, which examines the implementation domain content of a class to measure class complexity. The domain content is determined using a knowledge-based program understanding system. The metric's examination of the domain content of a class provides a more direct mapping between the metric and common human complexity analysis than is possible with traditional complexity measures based on syntactic aspects (software aspects related to the format of the code). Additionally, this metric represents a true design metric that can measure complexity early in the life cycles of software maintenance and software development. The SCDE metric is correlated with analyses from a human expert team, and is also compared to syntactic complexity measures. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: object-oriented metrics; knowledge-based systems; program understanding; domain content; metric validation; syntactic complexity measures

1. INTRODUCTION

Not long ago software engineers wrote most code in a traditional manner, first identifying the problem, then breaking it down into various functions. Recently the object-oriented (OO) paradigm for code development has been shown to reduce faults and improve reusability [1]. OO software is also considered easier to maintain [2]. Over the past 15 years, there have been tremendous increases in the amount of OO code produced. With OO software, what a system does is expressed in terms of interacting 'objects' and classes of objects, each of which provides many services. The collection

*Correspondence to: Dr Letha H. Etzkorn, Computer Science Department, The University of Alabama in Huntsville, Huntsville, AL 35899-0000, U.S.A.
†E-mail: letzkorn@cs.uah.edu

Received 5 February 2002
Revised 31 May 2002

Problems with Syntactic Metrics

- Herraiz and Hassan [8] note that . . .

  > Syntactic complexity metrics cannot capture the whole picture of software complexity. Complexity metrics that are exclusively based on the structure of the program or the properties of the text (for example, redundancy, as Halstad's metrics do), do not provide information about the amount of effort that is needed to understand a piece of code.

  This idea of **measuring the amount of effort needed to understand a piece of code** is what fascinates me, and what drives this entire report, actually.

- Lines of Code - This sounds promising because common sense states that the more lines of code there are there greater the opportunity for mistakes. This is indeed true in some sense. But LOC is notoriously difficult to measure. Depending on how they are counted, a program could vary in complexity based on whether or not comments, includes, blank lines, and macro definitions are included or not. Worse still is the vagaries of programmer whims regarding putting multiple statements per line, or even including "functional" yet "empty" lines (like only a semi-colon) for various formatting or readability reasons.

- Lines of Code (again) - There are many (myself included) that argue that the best programmers write less code. I bet there is correlation between code compactness and quality programming, at least to a point. The best programmers also write clear code, and that defies over-compacting. (There is difference between compact code and minified code, after all.)

- Coupling Between Objects, Coupling Between Methods, Lack of Cohesion in Methods - These measure, in one form or another, the extent to which the parts of a class perform the same or related tasks. The idea is that methods that access the same member variable are performing related tasks. Li and Henry [5], CK metrics [3], and TKC (Tang, Kao, an Chen - see prior chapter) metrics are similar.

  These metrics can fail to correctly reflect actual class cohesion when the constructor and destructor are used in the method analysis. If member variables are initialized in the constructor (a well-known best practice; see my "Code Complete" chapter) then every other method in the class will be considered to be cohesive with the constructor. (This is clearly not the case, but you can see how the analysis can make that mistake.) It gets worse. Since the constructor is is then cohesive every method in the class, all classes are therefore cohesive to each other. This means the class will have a maximum cohesion rating regardless of whether any of the methods have anything at all to do with each other. Ugh.

- Different structural aspects of two sets of source code can result in different metric values even if both programs perform the same task. Clearly syntactic measures do not provide us with a picture of overall quality. We need more, for balance.

Code-related and Domain-related Internal Documentation

- Code-related documentation involves looking only at the functional code. This is what we do in syntactic analysis.

- Domain-related documentation involves looking at semantics via comments and identifiers in the source code. Biggerstaff [6] discusses the nature of determining not only what the code does, but why it does it by looking not only at the code but also at the comments and identifiers.

- By using syntactic and semantic measures in concert, we can determine both programming information (code complexity) and domain information (human complexity, or "comprehendability"), and thus glean a more complete picture of source code quality than we could otherwise.

- Syntactic Analysis + Semantic Analysis = Overall Quality.

THE SEMANTIC CLASS DEFINITION METRIC

Because they represent the basic building blocks of any Object-oriented system, the authors use the class as their basic unit of measurement. If we accept that the complexity of a given design is influenced by the complexity of its component classes then this makes sense. (If we do not accept that -- a consideration not taken into account by the authors -- there there's trouble in River City; and that starts with "T" which rhymes with "C" and that stands for complexity!)

Now that we've decided that we're looking at classes we need to figure out how to measure semantic entropy. Prior authors used "name strings" (identifiers, basically). Our authors use something more grand: "the domain related concepts and key words that are identified as belonging to the class". That's cool, but how in the world do they identify these? Two tools: DESIRE (DESign Information Recovery Environment) [6] and PATRicia (Program Analysis Tool for Reuse) [7]. The authors use PATRicia (since they wrote it) in this work.

Using PATRicia the authors develop a system to identify domain-related concepts and keywords. These are extracted from the comments and identifiers the classes comprising the system under observation. The authors posit that the amount of "information" (they are very unclear on exactly what they mean by that, and whether or not it is different than data (which is most certainly is!)) conveyed by each of the identified domain-related concepts or keywords is inversely related to its probability of occurring. They then develop a formula for the average amount if information contributed by each domain-related concept or keyword (again, from the code and comments). The details are in the paper.

FINDINGS

The authors conduct three experiments comparing their semantic entropy metric to analyses from human experts and syntactic complexity measures. In the frist case, the semantic entropy metric correlates nicely with syntactic predictors of complexity. When measured against human experts the semantic entropy metric correlates strongly, permitting the authors to assert that their metric is like having a group of human experts analyze the code. (They also point out that their system can only be as good as the PATRicia knowledge base and that that may be a limiting factor in domains with with which PATRicia is not all that familiar.)

Overall, the authors state that their semantic entropy metric performed as well or better than syntactic complexity metrics and human experts. Very impressive.

❖ *Traits We Can Measure or Detect, and How*

- The semantic entropy metrics described in this paper are highly reliant on the knowledge base used to identify domain-related concepts and keywords and extract them from the comments and identifiers the classes comprising the software system. I don't think it's practical to assume any sort of wide dissemination of this/these is likely. From that point of view, stand-alone analyses of this sort are extremely difficult to implement.

- But that opens up an opportunity for a web-based service. If some enterprising Ph.D. student with entrepreneurial experience were to start a business around this idea it might be very successful. There would be a ton of issues to work out, not the least of which would be those around privacy and security.

- A service that measures the amount of effort needed to understand a piece of code would be invaluable for software developers the world over.

- It's something to think about . . .

Chapter 4

CODE COMPLETE

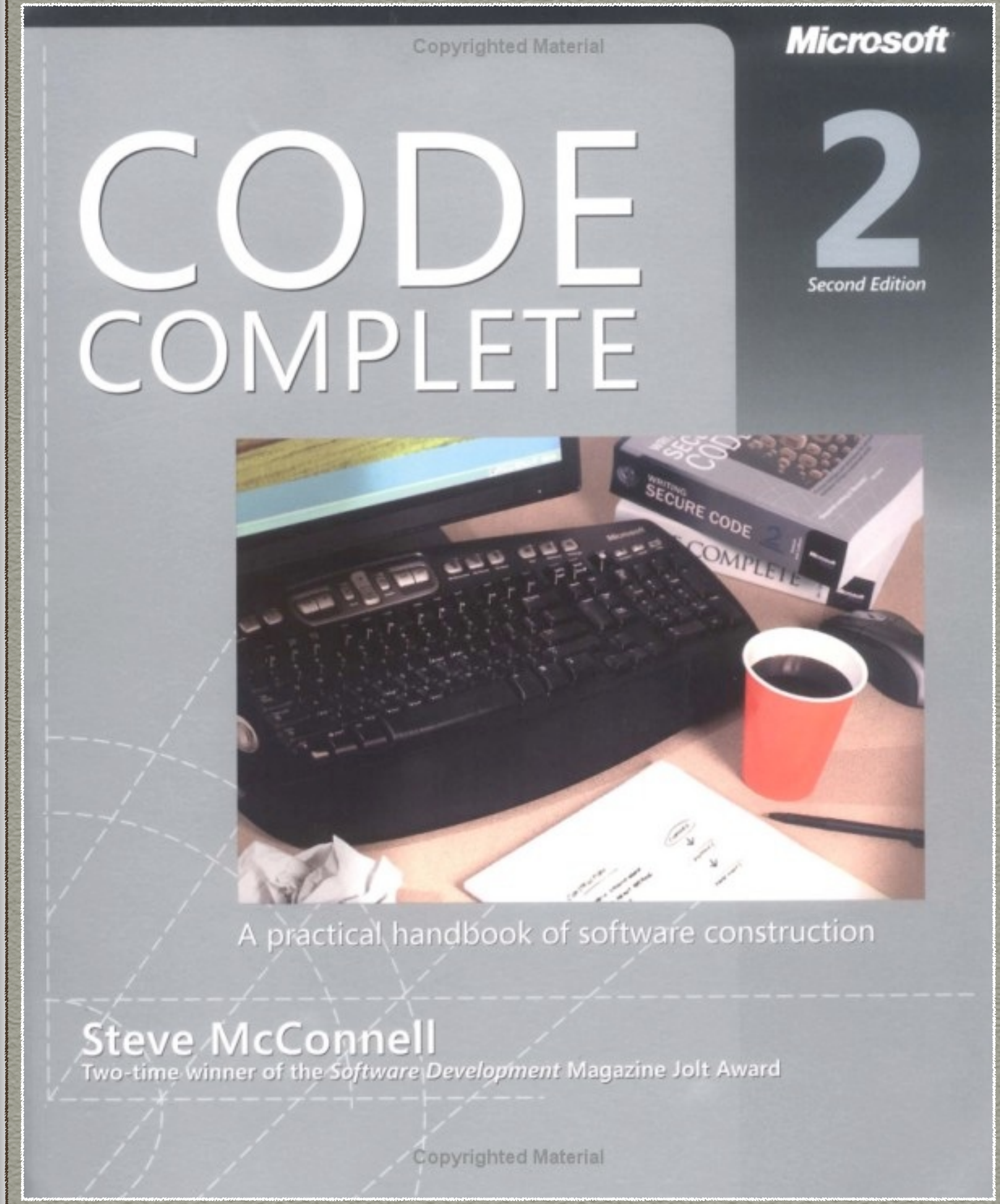
Code Complete, 2nd Edition

by Steve McConnell

Microsoft Press, 2004. 960 pages.

ISBN: 0735619670.

This 2004 edition, revised and updated from the 1995 original, is one of the best books on software development ever written. I have used this book as the text for programming classes I've taught. It's full of great advice and wisdom on the art and science of software development. It's certainly a good place to begin looking for objective measures of source code quality.


Copyrighted Material
Microsoft
CODE
COMPLETE
2
Second Edition
A practical handbook of software construction
Steve McConnell
Two-time winner of the Software Development Magazine Jolt Award
Copyrighted Material

HIGH-QUALITY ROUTINES

Reasons to Create a Routine

- Isolate and therefore reduce complexity
- Hide implementation details
- Limit effect of changes
- Hide data, sequences, and pointers operations
- Introduce understandable abstractions
- Avoid duplicate code
- Improve portability
- Simplify complicated boolean tests
- Make central points of control
- Facilitate code reuse

Small Routines a Waste?

Just because something is small does not mean it shouldn't be a routine. Routines make code more readable, even in small doses.

Design Goal:

"The goal is to have each routine do one thing well and not do anything else."

Also known as *functional cohesion*, this leads to higher reliability.

Good Routine Names . . .

- describe everything it does.
- are typically composed of verbs and direct objects.
- avoid meaningless / vague/ wishy-washy verbs.
- don't differentiate by number alone.
- are as long as necessary.
- describe the return value, if any.
- use opposites precisely.

Parameters . . .

- should be listed in input-modify-output order and have any status /error variables last
- are never used working variables.
- document interface assumptions.

- use naming conventions.

❖ *Traits We Can Measure or Detect, and How*

- Duplicate code - compiler backend during code optimization. (Compare basic blocks.)
- The complexity of boolean tests - compiler front end in parse. (Count the number of "boolean operator" productions for each expression.)
- The semantic structure of identifier names - compiler front end in parse. (Use a string tokenizer to split the identifiers into component "words" to send to a grammar classifier.)
- Identifiers differentiated only by number or letter - compiler front end in lex (using pattern matching).
- Parameters used as working variables - compiler front end in semantic analysis. (Check for parameters appearing on the left-hand-side of assignment-type statements during type-checking.)

NAMING PRACTICES

Declaration and Initialization

- If your language supports implicit declarations, turn them off!
- Always initialize variables when they are declared if possible. If that is not possible, then initialize them as close to their first use as possible.
- Use constant or readonly variables wherever you can.
- Initialize class member data in constructor.
- Initialize constants once (if possible).
- Initialize variables with an init() routine.
- Reset counters before their next use.

Minimizing Scope

- Keep variables as local as possible.
- Keep window of vulnerability as small as possible.
- Keep average span as small as possible.
- Use just in time assignment where you can.

- Favor the smallest scope for the variable.

Variable Use

- Use each variable for one purpose only.
- Ensure all declared variables are used.

Identifier Names

- Express what rather than how.
- Avoid names with hidden meanings.
- Use nouns rather than verbs for variables; leave the verbs for routine names.
- Make them as long as they need to be.
- Make names descriptive of their contents or purpose.
- Never use the name "temp". Ever.
- Use capitalization and underscores effectively to increase readability. Understanding a variable name should not be like solving a cryptogram.
- For Booleans, use positive variable names (e.g., *found* instead of *notFound* or even *notUnfound*) and use names that imply true or false (e.g., isFound)
- For constants, use all capital letters and represent abstract entity (pi) rather than constant value (three_point_one_four).
- Differentiate variable names and routine names by beginning variable names in lowercase and routine names in uppercase.
- Differentiate class names from instance names by beginning class names with uppercase and instance names with lowercase .
- Use standardized abbreviations consistently.
- Remove articles.
- Remove useless suffixes.
- Do not use phonetic abbreviations.
- Finally, use common sense. If an identifier name cannot be understood at first glance, don't use it.

Identifier Names to Avoid

- Misleading names
- Names with similar meanings
- Variables with different meanings but similar names

- Homonyms
- Misspelled words or even words commonly misspelled
- Multiple languages in the same project.
- Names that are totally unrelated to what the variable represents
- Names containing hard-to-read characters
- Reserved words

❖ *Traits We Can Measure or Detect, and How*

- Variables that could be constants - compiler back end during code optimization. (Live variable analysis, reaching definitions, and constant propagation all help here).
- Uninitialized variables - compiler front and (via symbol table) and back end (via code optimizations). Most compilers will do this for you already.
- Scattered variable initialization - compiler front end in semantic analysis. (As type checking is proceeding we can also look at the symbol table and scope data to see where the variable initializations are coming from.)
- Unnecessary scope - compiler front end in semantic analysis. (Analysis of the abstract syntax tree (AST) and the scope data contained in the symbol table (and possibly linked to the AST) might help us identify variables whose scope could be constricted.
- Variable names that do not include nouns. This is another case of semantic structure. Do it in the compiler front end in parse. (Use a string tokenizer to split the variable names into component "words" to send to a grammar classifier. A Natural Language Toolkit like http://www.nltk.org/ seems like it might be useful here.)
- The use of "temp" and other "stop words" as inadvisable variable names - compiler front end in parse. (Keep a list of "stop words" accessible to the parser component of the compiler and warn the programmer if any are detected.)

❖ *IDEA:*

I wonder what other information retrieval techniques (like "stop words") could be used in programming language analysis via compiler extensions or other analysis means.

- The use of commonly misspelled or misleading words, reserved words, curse words, whatever... - compiler front end in parse. ("Stop words" again.)
- The use of homonyms - We might use a pronunciation dictionary like the one at CMU (http://www.speech.cs.cmu.edu/cgi-bin/cmudict) to determine identifiers that are spelled differently but sound similar to the ear, and this muddle the meaning od the code. (E.g., "Ham and Eggs" vs. "Hammond Eggs".)

- Identifier names unrelated to their meaning - compiler front end in lex and parse by analysis of the comments (if any) and the variable names. (This is a little far out, but there's been some work in semantic distance here, which we'll look at later in this report.)

CONDITIONAL EXPRESSIONS

if statements

- Write the normal (expected) path through the code first, in the *if* block, then write the unusual cases in the *else* blocks.

- Put the most common cases first.

- You almost always need an else, so be careful when omitting it. James Elshoff [1] found that 50 to 80 percent of *if* statements alone should have been followed by an *else*.

case statements

- Put the normal case first.

- Order cases by frequency.

- If all cases are equally likely and equally important, chose another order like numeric or alphabetic or something so that the code is predictable.

- Use the *default* clause to detect errors.

- Avoid accidentally dropping through cases. Be careful with *break*.

❖ *Traits We Can Measure or Detect, and How*

- *if* and *case* statements not ordered by their likelihood - An analysis of the source code in concert with runtime profiling might be able to detect whether or not *if* and *case* statements are ordered by their likelihood. The compiler front-end could output the expected "bound abstract relative frequency" (or BARF; I just made that up) of the *if* and *case* clauses. The profiler could then observe actual runtime events, compute *if* and *case* clause actual runtime frequency, compare it to the BARF, and report back to the user.

- *if* clauses without an *else* - These can be easily detected in the parse phase of compilation. Warnings or hints could be generated pointing this out and asking the user to take special care with them. (A good compiler gives errors, warning, and hints. Errors stop compilation; hints do not; warnings might, depending on their severity or the mood of the compiler writer at the time s/he wrote it.)

- *case* statements without a break after each clause - These can be easily detected in the parse phase of compilation. Warnings or hints could be generated pointing this out and asking the user to take special care with them. (Microsoft's C# compiler does this today.)

Chapter 5

EVIDENCE-BASED FAILURE PREDICTION

Evidence-based Failure Prediction
by Nachi Nagappan and Thomas Ball
Microsoft Research - RiSE, Redmond, Wa.

The authors discuss six metrics for failure prediction they examined in a case study performed on the Windows Vista and Windows Server 2003 code. Windows is about 40 million lines of C, C++, and C# and is used (for better or worse) by virtually everybody with a computer all over the world.

Microsoft® Windows® .NET™ Server 2003

```
A problem has been detected and Windows has been shut down to prevent damage
to your computer.

 Ntfs.sys

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0xFFFFFBA0B0925AC0,0x0000000000000001,0xFFFFF88001CE9AF5,0
x0000000000000005)


***      Ntfs.sys - Address FFFFF88001CE9AF5 base at FFFFF88001C54000, DateStamp
 4a5bc14f
```

INTRODUCTION

When talking about software metrics it's useful to consider both internal and external metrics. Internal metrics are those that are derived from the code itself. (Perhaps "intrinsic" would be a better term?) Cyclomatic Complexity is one example of an intrinsic metric. External (extrinsic?) metrics are those derived from external assessments of the product. Bugs and failures are extrinsic metrics.

The best intrinsic metrics are those that relate to extrinsic ones. (Actually, what good at all is an internal metric that has no bearing on the external evaluation of your software product?) Thankfully, internal metrics have been shown to be useful as early indications of externally visible product quality.

Here are a the metrics the authors measured on the 40 million lines of Windows source code.

1. CODE COVERAGE

Code coverage refers to the extent to which all source code in a given project can be tested (presumably by automated unit testing or something similar). For example, if there are 100 lines of code and we have tests that verify the functionality of 84 of them (note we are talking about functionality, so we do not need 84 tests but far less in most cases), then we have 84% code coverage. But what about the other 16%? What might cause code to avoid test coverage?

Unreachable branches are one example of ways in which code may avoid test coverage. If there are blocks of code that are unreachable then it's clearly impossible to test them since they never execute. This the authors note that it can be argued that higher coverage should lead to the detection of more flaws in the code. (And, following that logic, higher quality code upon release assuming the flows that are discovered are fixed before release.) This line of thinking makes sense to me.

Doh! I guess this is why we read read research, because it turns out that this is not the case. Who'da thunk it? The authors point out (in [9]) several flaws the the above logic(?)

- Coverage is about lines of code and has nothing to do with whether or not they are the right lines or even the right code.
- The simple fact that a statement was executed does not at all imply that it has been tested with all possible data values within and without the intended domain.
- What about complexity? Achieving 84% coverage on a module with cyclomatic complexity of 1000 is far more difficult than achieving 84% coverage on a module with a cyclomatic complexity of 10. (But is it 100 times harder? I doubt it. I wonder what that scaling relation is.) In fact, a better measure could be constructed from looking at complexity values tied into code coverage to help describe the effectiveness of code coverage percentages.

The authors compared branch and block coverage values for Windows Vista with Vista field failure reports six months after release. They observed weak positive correlation between coverage and quality and low prediction precision. As a result of their findings they suggest that code coverage is not very useful all by itself but may be when combined with other factors as previously mentioned, like complexity and code churn.

2. CODE CHURN

Successful software is never finished: it evolves over time along with the enterprise it serves. Code churn measures the changes (adds/updates/deletes) made to a module/component over the course of time and quantifies the extent of this change.

Code churn data is usually extracted from the version control system used to manage the code. Differences are measured in terms of lines of code. (A utility like *Diff* (or *WinDiff*), common to virtually all version control systems, even bad ones like CVS) calculates the lines-of-code delta between versions of the same module/component.)

Since software projects vary in so vastly in size, and modules vary a lot in size within the same system, *relative churn* is a more accurate measure. Relative churn is calculated using delta-lines-of-code values normalized (divided into) by some other factor, usually total lines of code or source file count, or something similar.

The authors hypothesize that code/modules that change more from one release to the next are more likely to contain faults than code/modules that change less. To test this they examined relative code churn between the release of Windows Server 2003 and Windows Server 2003 Service Pack 1. [10]

Using a variety of metrics derived by normalizing absolute code churn with different factors (total lines of code, file count, files churned, others) the authors found that the correlation between actual and estimated defect density is strong, positive, and statistically significant. This means that code churn and code quality (as measured by post-release defect reporting) are highly and positively correlated. The more code churn, the more defects.

3. CODE COMPLEXITY

Source code complexity can be expressed using several different metrics, some of which we've already discussed in this report in chapters two and three. The metrics popular at Microsoft are . . .

1. Executable lines of code
2. Cyclomatic complexity
3. Fan-in - other functions calling a given module

4. Fan-out - other functions called from a given module

5. Total number of methods in a class (all of them)

6. Maximum inheritance depth for a class

7. Coupling to other classes

8. Number of subclasses directly inheriting from a parent

The authors experimented once again on Windows Server 2003 and found that those complexity measures were fair predictors code quality, though not as good as code churn.

In discussing related work the authors note that Basili et al. observed in 1996 that the CK metrics WMC and RFC correlate with defects while LCOM does not. It's interesting to note that these findings agree with those of Chen et al. (chapter 2).

4. CODE DEPENDANCIES

In any serious-sized software development effort many people or many teams work independently on many different parts of the system. A software dependency is a relationship between two pieces of code: either a data dependency wherein the two pieces of code share some data element(s) or a call dependency wherein one piece of code calls the other.

In this context we can see code churn in a new light. Suppose one piece of code (let's call it "Kirk") has many dependencies on another piece of code (called "Spock" for example (and giggles)). If there is a lot of churn in the Spock code then we would naturally expect a certain amount of churn in the Kirk code in order to keep up with it and stay in sync. Put another way, churn often propagates across dependencies. It's only logical to conclude, therefore, that a high degree of dependence coupled with churn will cause errors that will propagate through the system, reducing it's quality. ("Fascinating." as Spock might say.)

The authors studied the dependencies among the binaries in Windows Vista and used them to predict failures. When compared to actual reported failures they found the precision and recall to be as good as those for code churn. Very cool.

5. PEOPLE AND ORGANIZATIONAL METRICS

Software is written by people. Human people (in most cases... though I've had a few consulting experiences where I was in doubt of that). The great Fred Brooks wrote in the 1995 anniversary edition of his famous *Mythical Man Month* that software product quality is strongly affected by organizational structure. This seems self-evidently true to me. So it makes sense, then, to wonder if attributes of the software development organization, measured as metrics, might predict faults as well as attributes of the code itself when measured as metrics.

Our authors wondered the same thing and came up with eight (8) organizational metrics to test against the entire Windows Vista code base.

1. Number of unique engineers who touched the code and are still employed at the company as of the software release date. The more people who touch the code, the lower the quality.
2. Number of unique ex-engineers who touched the code but left the company before the software release date. A lot of team members leaving the company affects institutional knowledge about the project and thus quality.
3. Edit frequency - The total number of times the source code for a given module was edited. An edit here is defined as a transaction consisting of an engineer checking a module out of the version control system, altering it, and checking it back in. This has nothing to do with the number of lines altered during the edit transaction. More edits means more instability, which leads to lower quality.
4. Depth of Master Ownership (DMO) - This metric determines the organizational level of ownership of a module based on the number of edits done and who in the organization did them. The organizational level of the person who has engineers reporting to him or her who have done a certain percent of the edits is considered to be the DMO. The authors used 75% for their experiments. The lower the level of ownership, the better the quality.
5. Percentage of organization contributing to development - This is the ratio of the number of people reporting to the DMO over the overall organization size. More organizationally cohesive contributors make for higher quality.
6. Level of organizational code ownership - The percent of edits made by the organization that contains the module owner. If there is no owner, then it's the precent of edits made by the organization that made the majority of edits to that module. More edit-wise cohesive contributors make for higher quality.
7. Overall organization ownership - The ratio of the percent of people at the DMO level making edits to the module relative to the total number of engineers who ever touched the module. (High is good.) The more concentrated the contribution to a module, the higher the quality.
8. Organizational intersection factor - The number of different organizations contributing more than 10% of the edits to a given module. Fewer organizations contributing to the code mean higher quality.

The authors calculated all of these metrics on 50 random splits of the Windows Vista code and obtained precision and recall values better than those of code churn (which was the best so far, until now) with little variance. **Organizational metrics were much better indicators of code quality then attributes of the code itself!** I did not expect that. That's very cool indeed.

6. INTEGRATED APPROACH

This is more of a meta-metric wherein the authors take several of the metrics defined earlier in this chapter and organize them in the context of a social network defined by the relationships among the developers as defined by their code interactions.

A social network defined by code interactions? Very cool indeed. "See Putting it All Together:
Using Socio-Technical Networks to Predict Failures" by

Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu1 for a detailed explanation of this idea. It's available online at
http://cabird.com/papers/bird2009pat.pdf .

❖ *Traits We Can Measure or Detect, and How*

- Unreachable code can be detected by a decent compiler using control-flow analysis. There are even some automated tools for this in compiler compilers like ANTLR and others.

- Code churn is easily calculated from data available in source code version control systems. At least two popular open source version control systems, Git and Mercurial, both have extensions for calculating code churn available today. Many other version control systems do as well.

- It would be interesting to develop a source code health/ quality dashboard that integrates all of these metrics and presents a holistic picture of the system under consideration. Most of the pieces are likely already there (compilers that detect unreachable code, test-driven development and unit testing frameworks like jUnit and nUnit and others, code churn analysis in source code version control systems, and the others.) Eclipse and Visual Studio already have some of this in place, especially the code-centric metrics, but I don't know of any sort of software developers workbench that does it all. It's a big job, but quite do-able, I think.

❖ *Hey... Wait a Minute*

It's important to note that these finds by these authors, while fascinating and very cool indeed, have ground truth only at Microsoft. And within Microsoft, only for Window Vista and Windows 2003 Server. The results may or may not generalize to other organizations and other software systems

That said, other studies have been done on large, open-source code bases, specifically Eclipse and and PostgreSQL, and drawn similar or supporting conclusions. This is a fascinating area for further research.

Chapter 6

REFERENCES

[1] Elshoff, James L. 1976. "An Analysis of Some Commercial PL/I Programs." IEEE Transactions on Software Engineering SE-2 no 2 (June): 113-120

[2] McCabe, Thomas J. 1976. "A Complexity Measure." IEEE Transactions on Software Engineering SE-2(4): 308-320

[3] S. R. Chidamber and C. F. Kemerer. "A metrics suite for object oriented deign." IEEE Transactions on Software Engineering, 20(6) June 1994: 476–493,

[4] Oram, A. and Wilson G. editors. "Making Software: What Really Works, and Why We Believe It." O'Reilly 2011 ISBN 978-0-596-80832-7

[5] Li W., Henry A. "Object-oriented Metrics that Predict Maintainability". The Journal of Systems and Software 1993; 23(2): 111-122

[6] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. 1994. Program understanding and the concept assignment problem. Commun. ACM 37, 5 (May 1994), 72-82. DOI=10.1145/175290.175300 http://doi.acm.org/10.1145/175290.175300

[7] Etzkorn LH, Davis CG. 1997. Automatically Identifying Reusable Components in OO Systems. IEEE Computer vol 30 issue 10. IEEE DOI 10.1109/2.625311

[8] Herraiz I, HassanA. Beyond Lines of Code: Do We Need More Complexity Metrics? chapter 8 pp 125-141 in *Making Software: What Really Works and Why We Believe It* edited by Andy Oram and Greg Wilson, 2011.

[9] Nachiappan Nagappan and Thomas Ball. Evidence-based Failure Prediction, chapter 23 pp 415-434 in *Making Software: What Really Works and Why We Believe It* edited by Andy Oram and Greg Wilson, 2011.

[10] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In Proceedings of the 27th international conference on Software engineering (ICSE '05). ACM, New York, NY, USA, 284-292. DOI=10.1145/1062455.1062514 http://doi.acm.org/10.1145/1062455.1062514