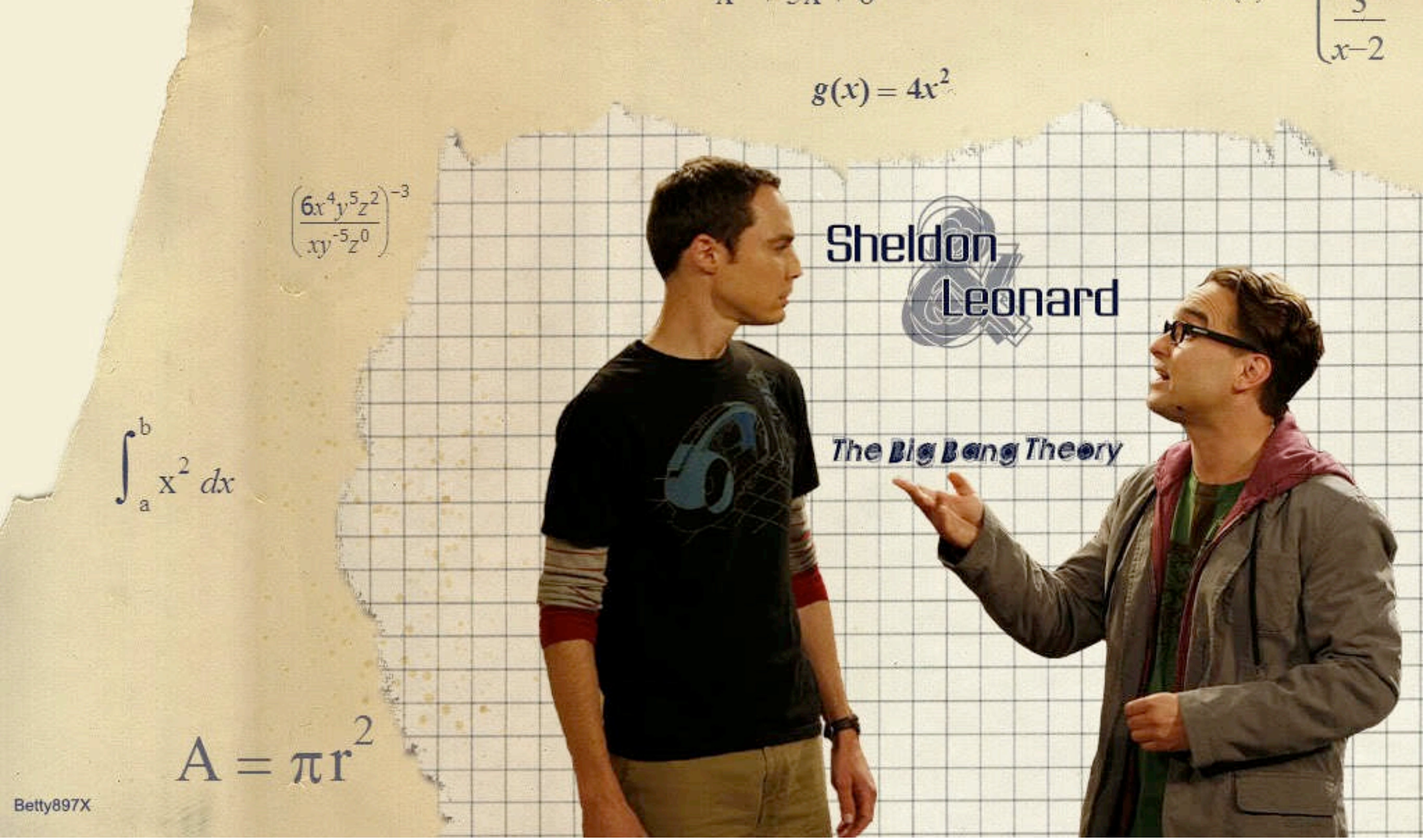
Empirical Software Engineering

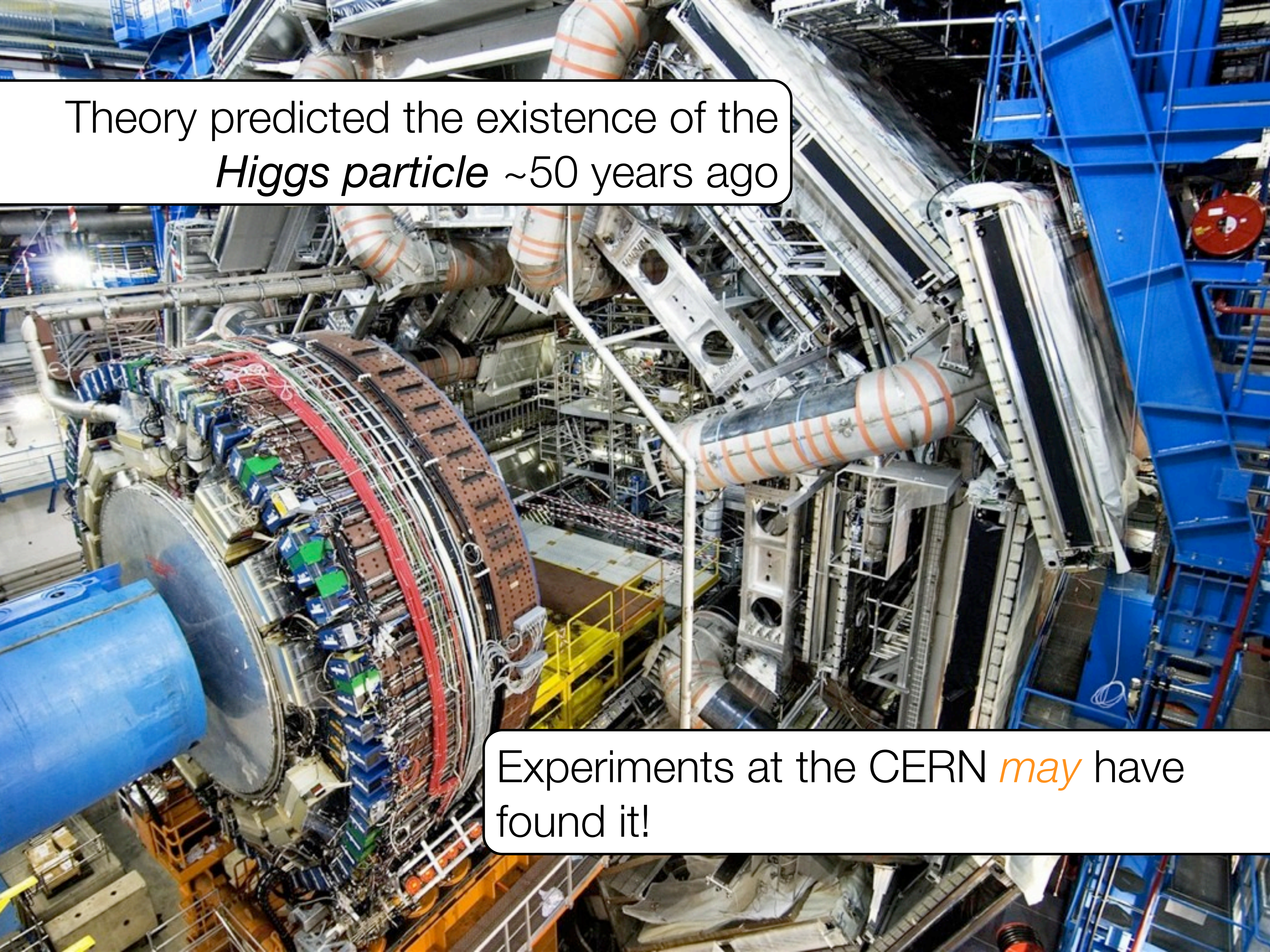
SW Maintenance & Evolution

What is it?

- Empiricism: *Observation*- and *experimental* driven
- Confirms (or rejects) theories through experiments
- Observations of phenomena can be the first step towards a theory
- A theory is more *general*
- A theory may concern things that have not been observed yet



Experimental Physics vs. Theoretical Physics



Theory predicted the existence of the *Higgs particle* ~50 years ago

Experiments at the CERN *may* have found it!

Theoretical models and abstractions of physics to rationalize and predict natural phenomena

Is there something like *theoretical* and *experimental* software engineering?

There is *empirical* software engineering!

Experiments and observations to explain natural and physical phenomena

Is anybody out there?

Confirm

verify

Falsify

The Role of Empiricism in Software Engineering

- SE is a (relatively) young field with *strong emphasis on the engineering aspect*
- Countless tools, languages, frameworks, ... are available
- Development paradigms: Agile, Test-Driven, X-treme, ...
- (Vague) ideas of how to develop good software

The Role of Empiricism in Software Engineering

- SE is a (relatively) young field with *strong emphasis on the engineering aspect*
- Countless tools, languages, frameworks, ... are available
- Development paradigms: Agile, Test-Driven, X-treme, ...
- (Vague) ideas of how to develop good software

In SE decisions are still based on *guts* and *personal experience*

Theoretical Aspect of in SE

- not really developed
- a few (theoretical) models
- a vague beliefs of laws and theories in SE
- cause - effect relations are unkown

Cocomo

Lehman's Laws

Mythical Man Month

Metrics

Theoretical Aspect of in SE

- not really developed
- a few (theoretical) models
- a vague beliefs of laws and theories in SE
- cause - effect relations are unkown

Cocomo

Lehman's Laws

Mythical Man Month

Metrics

People required (P) = Effort Applied / Development Time [count]

Theoretical Aspect of in SE

- not really developed
- a few (theoretical) models
- a vague beliefs of laws and theories in SE
- cause - effect relations are unkown

Cocomo

Lehman's Laws

Mythical Man Month

Metrics

Group intercommunication formula: $n(n - 1) / 2$

People required (P) = Effort Applied / Development Time [count]

Theoretical Aspect of in SE

- not really developed
- a few (theoretical) models
- a vague beliefs of laws and theories in SE
- cause - effect relations are unkown

Complexity = E − N + 2P

Cocomo

Lehman's Laws

Mythical Man Month

Metrics

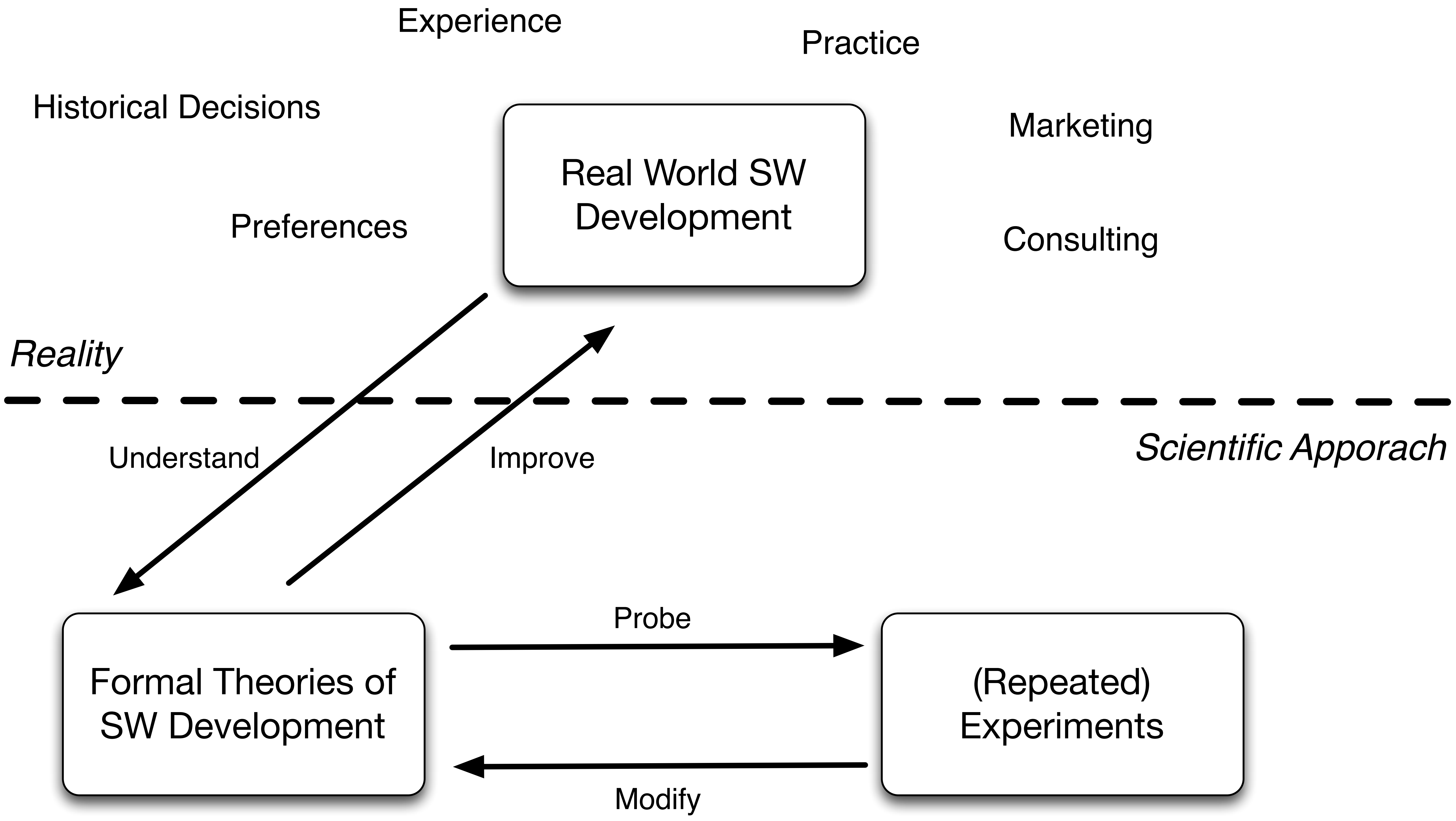
Group intercommunication formula: n(n − 1) / 2

Lines of code

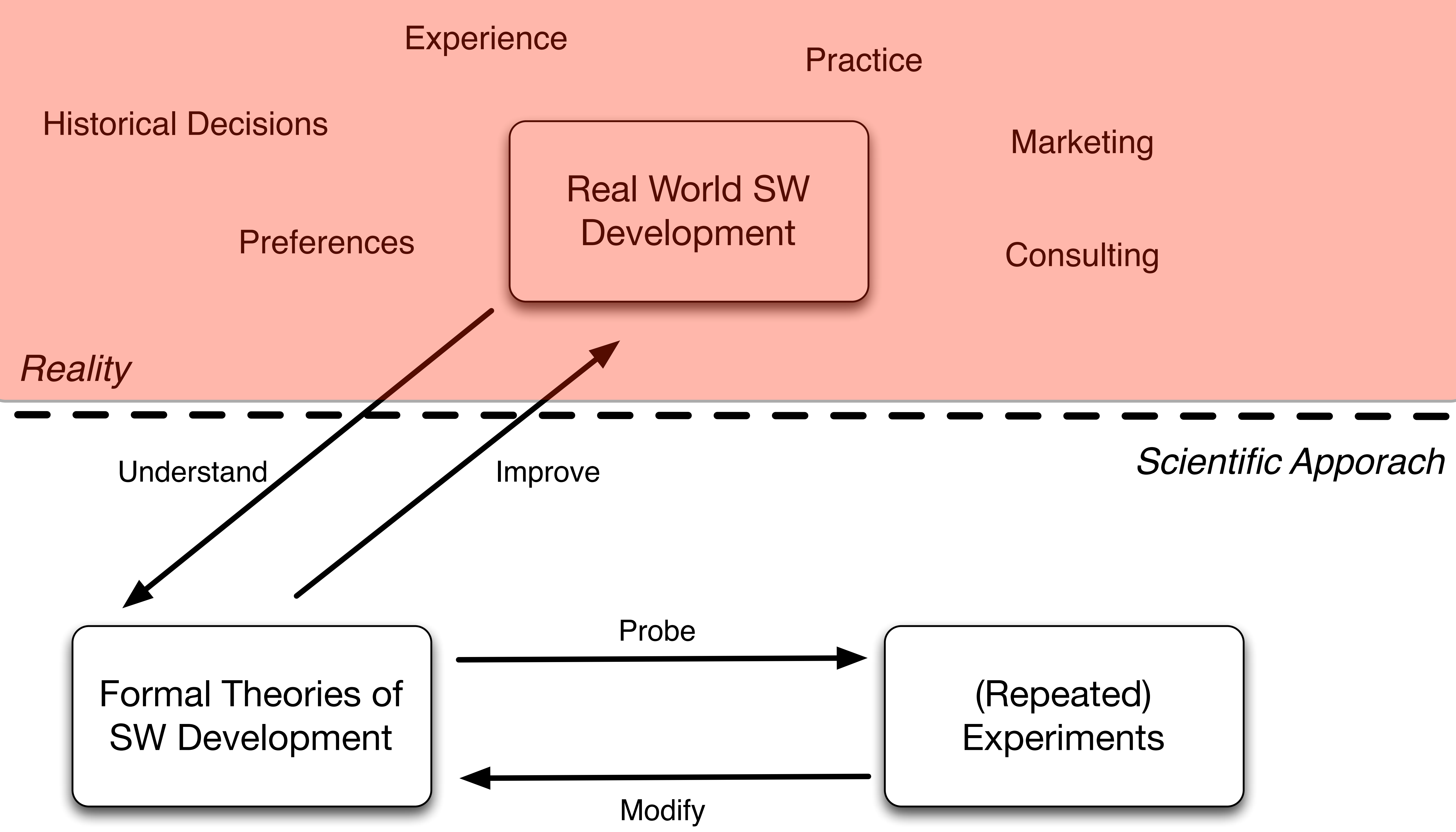
People required (P) = Effort Applied / Development Time [count]

Problems of Models and Theories in SE

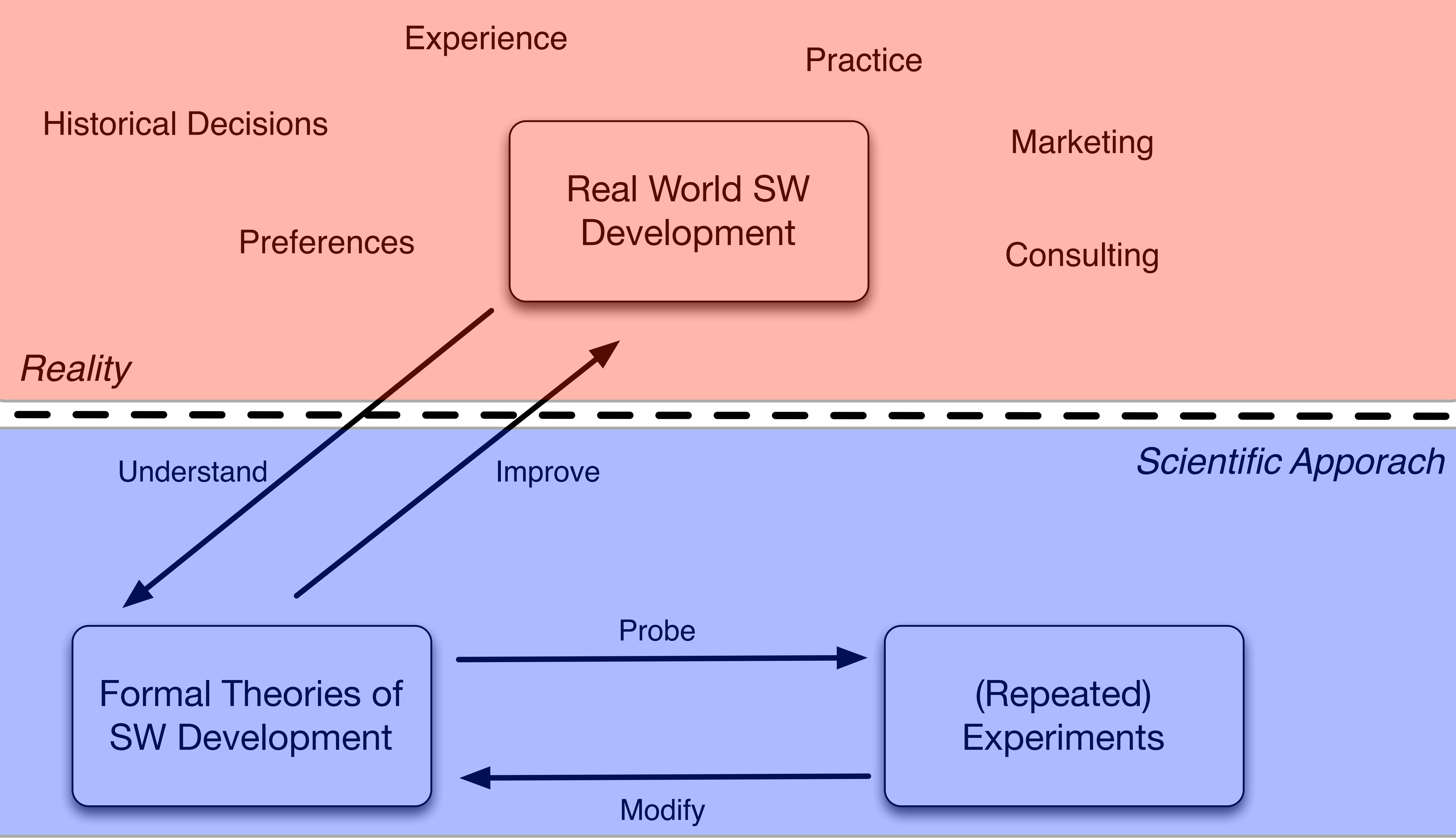
- Hard to generalize for all software projects
- Software projects differ widely in so many aspects
- Not always straightforward to apply in practice
- No standard measurement procedures and units
- Sometimes grounded on anecdotal evidence



Credits to *Dr. Victor Pankratius & Andreas Höfer, Karlsruhe Institute of Technology*



Credits to *Dr. Victor Pankratius & Andreas Höfer, Karlsruhe Institute of Technology*



Credits to *Dr. Victor Pankratius & Andreas Höfer, Karlsruhe Institute of Technology*

The Role of Empiricism in Software Engineering

- Investigates (and attempts to explain) phenomena in SW
- Quantifies a phenomena or fact of SW with numbers
- Quantifies implicit knowledge
- Knowledge drawn form empirical SE can help practitioners
- Supports decision making

Is there a relation or connection between things?

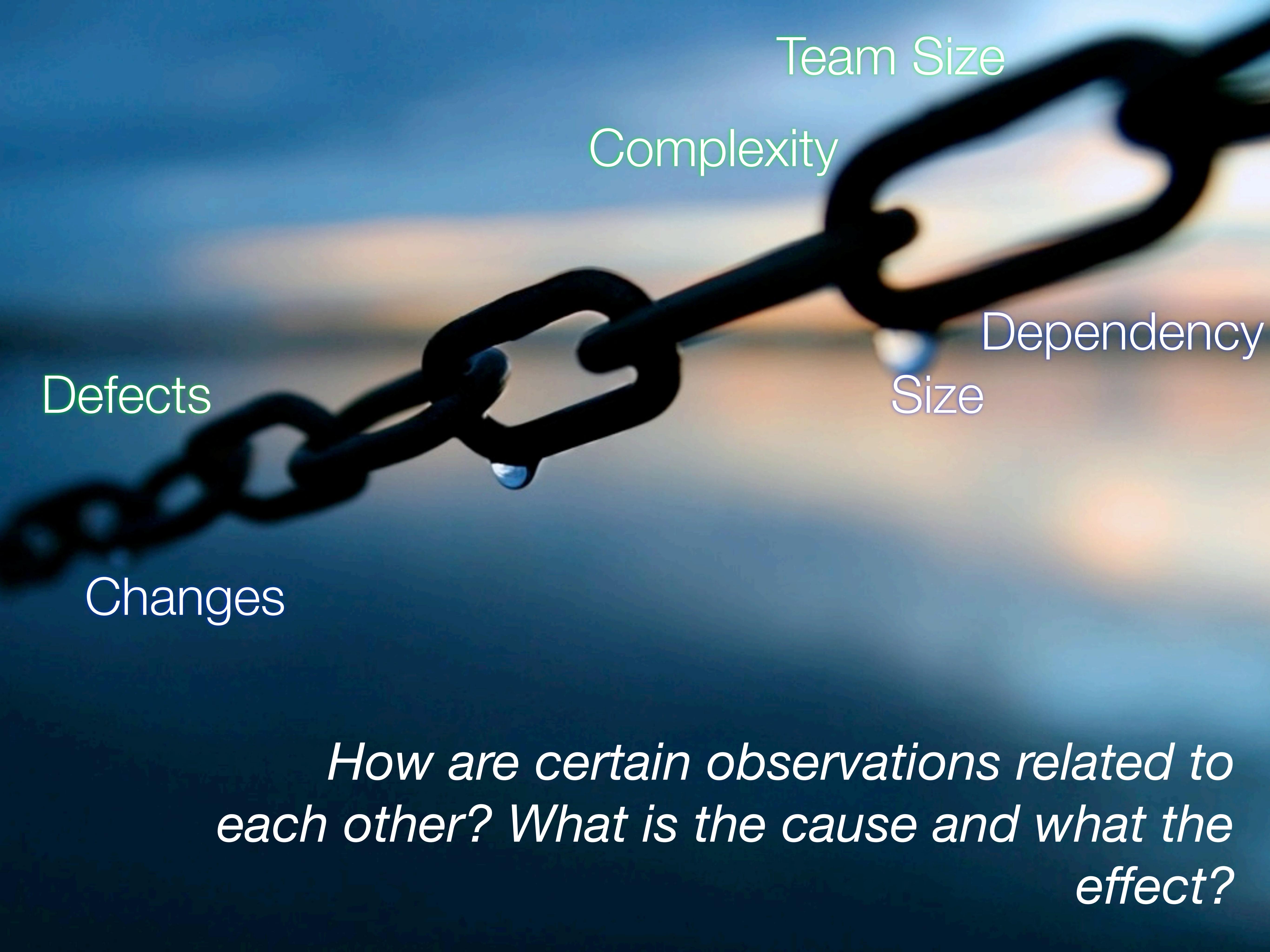
Is size related to defects?

Is there a difference?

Difference between a waterfall and a iterative development process?

Can we make forecasts?

Do we have to expect more defects in the future?



How are certain observations related to each other? What is the cause and what the effect?

Relation: Correlation Analysis

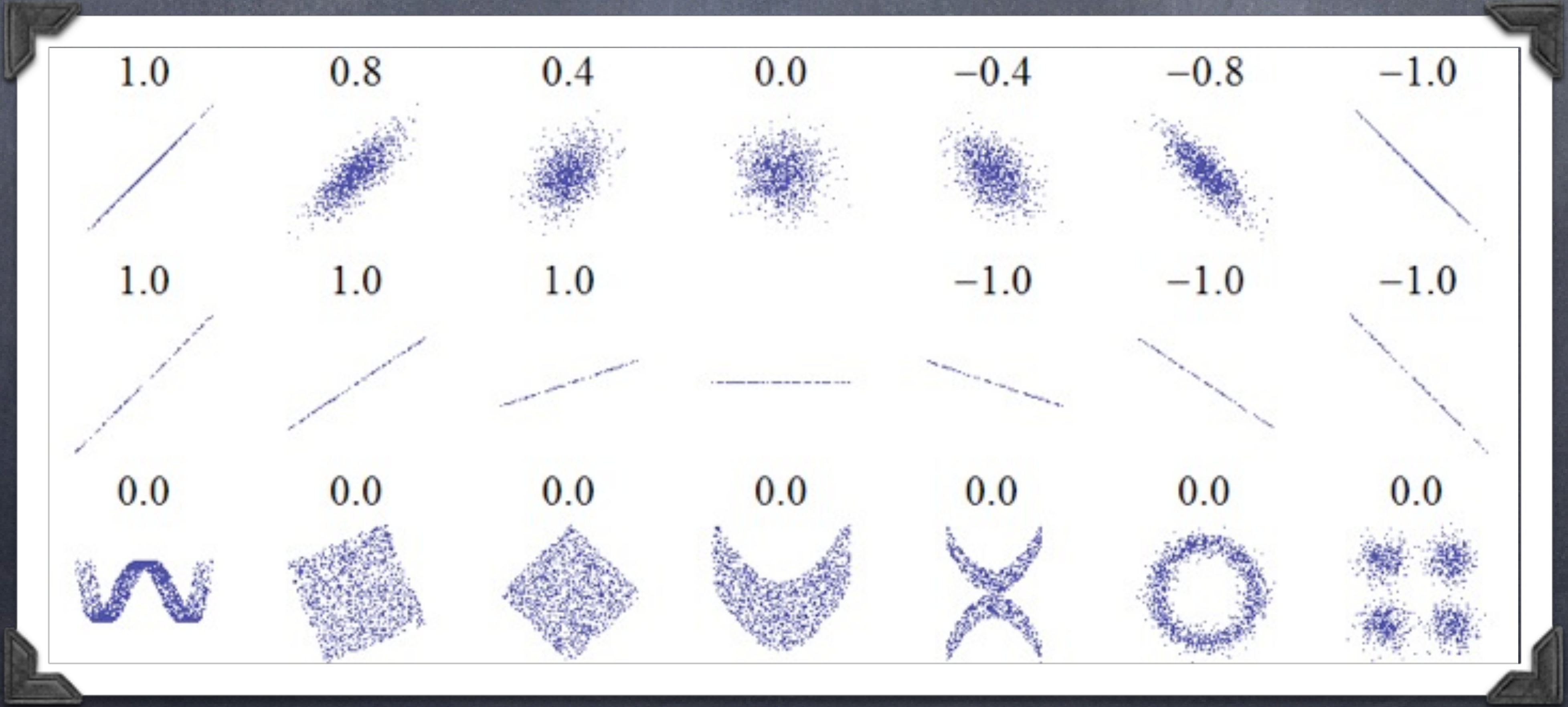
- You may remember your stats class
- Investigates the strength of the relation between two measured variables
- Investigates the direction of the relation: Positive or Negative
- Correlation coefficients are available in any statistics package
- Parametric vs. non-parametric correlation

$$Cov(X,y) = E((X - E(X))(Y - E(Y)))$$

$$r(\mathrm{X},\mathrm{Y}) = \frac{Cov(\mathrm{X},\mathrm{Y})}{\sigma(\mathrm{X})\sigma(y)}$$

Correlation Refresher

- Parametric correlation: Pearson correlation
- Linear relationship only!
- Significance tests require normal distribution



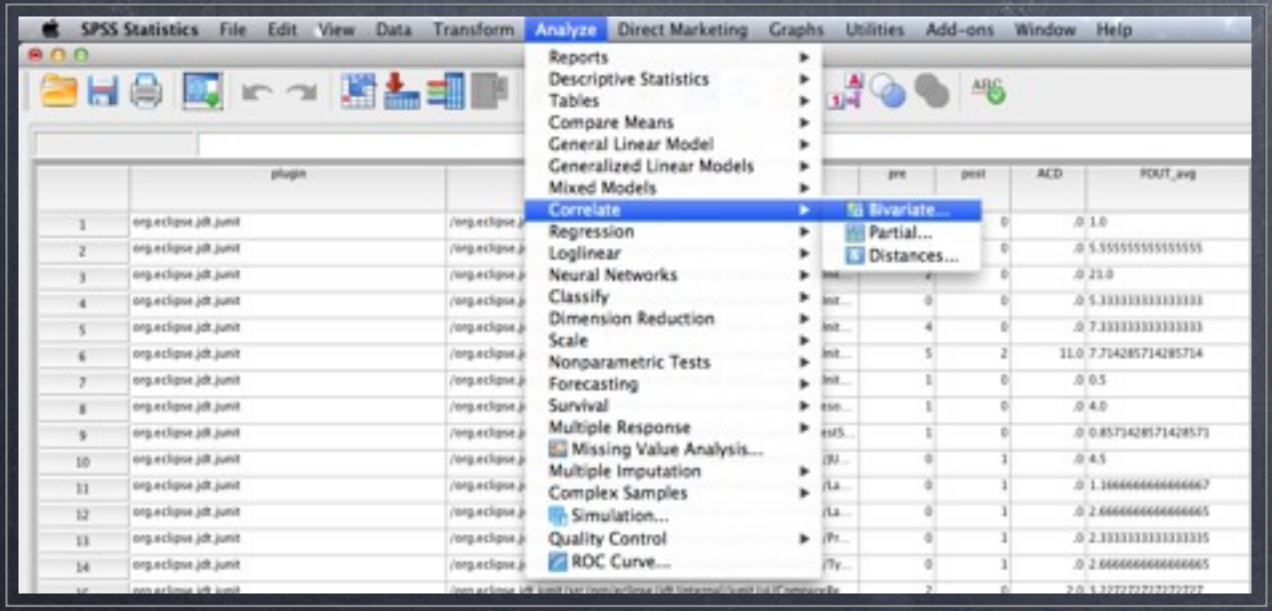
Wikipedia

Correlation Refresher

- Non-parametric correlation: Spearman rank correlation
- Linear relationship is not needed!
- Applies to non-normal distributions
- Available for ordinal data

Correlation Refresher

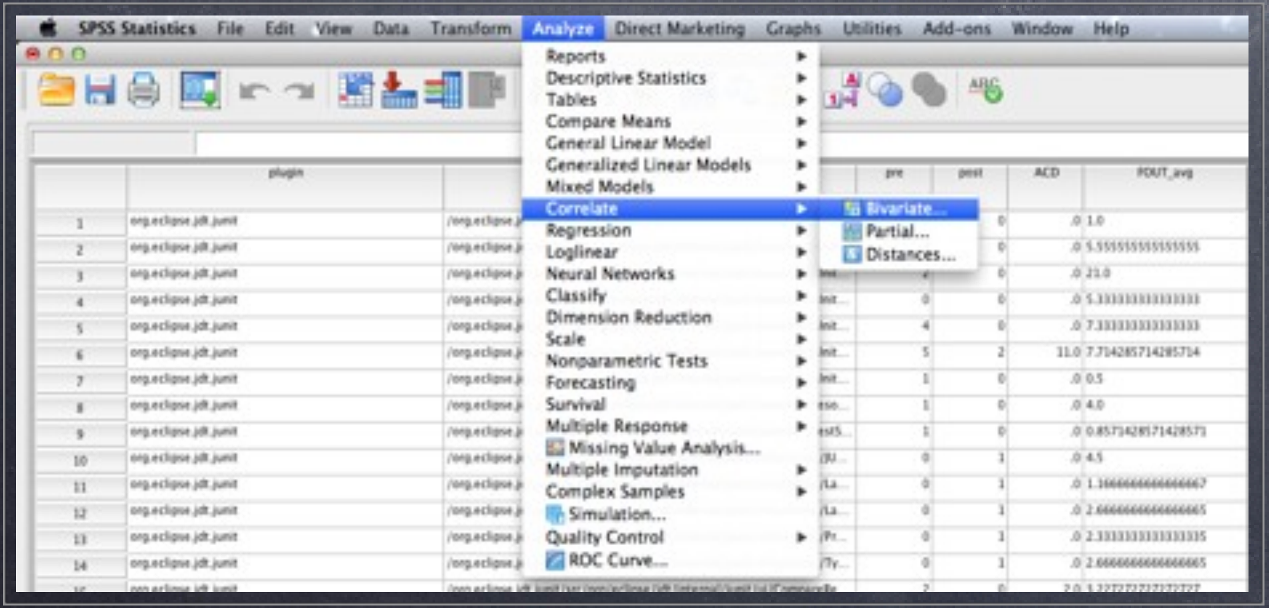
- Non-parametric correlation: Spearman rank correlation
- Linear relationship is not needed!
- Applies to non-normal distributions
- Available for ordinal data

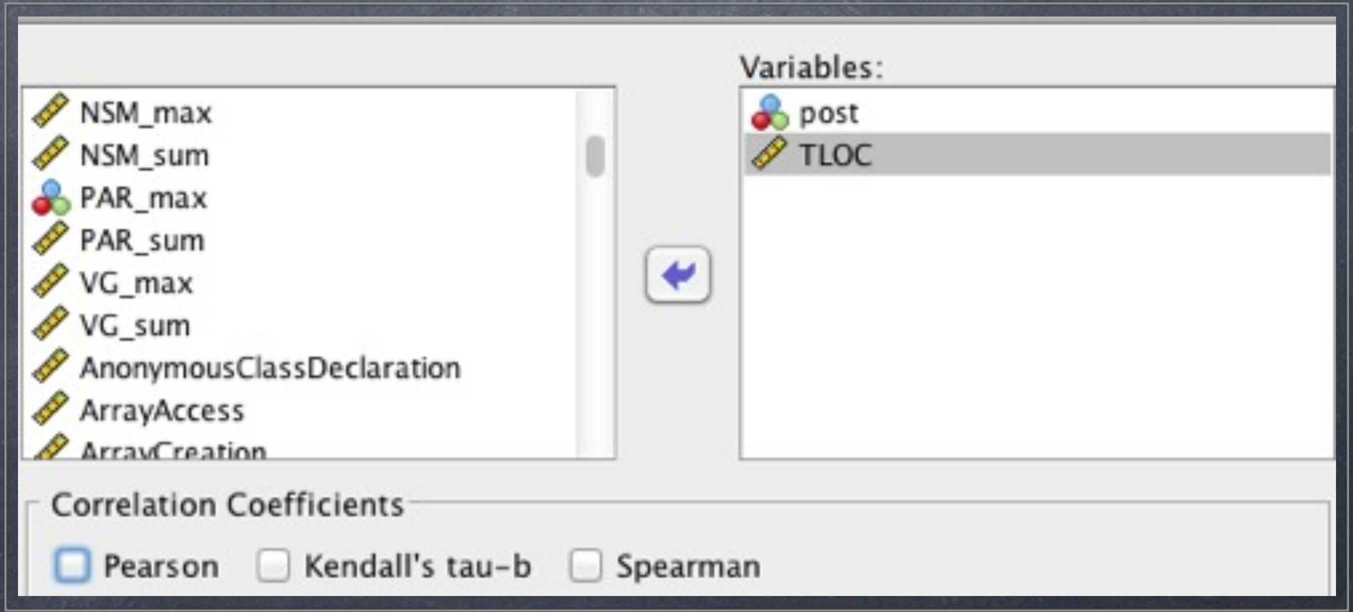


	plugin			pre	post	ACD	FOUT_avg
1	org.eclipse.jdt.junit	/org.eclipse.j			0	.0	1.0
2	org.eclipse.jdt.junit	/org.eclipse.j			0	.0	5.5555555555555555
3	org.eclipse.jdt.junit	/org.eclipse.j	Init...	2	0	.0	21.0
4	org.eclipse.jdt.junit	/org.eclipse.j	Init...	0	0	.0	5.333333333333333
5	org.eclipse.jdt.junit	/org.eclipse.j	Init...	4	0	.0	7.333333333333333
6	org.eclipse.jdt.junit	/org.eclipse.j	Init...	5	2	11.0	7.714285714285714
7	org.eclipse.jdt.junit	/org.eclipse.j	Init...	1	0	.0	0.5
8	org.eclipse.jdt.junit	/org.eclipse.j	eso...	1	0	.0	4.0
9	org.eclipse.jdt.junit	/org.eclipse.j	estS...	1	0	.0	0.8571428571428571
10	org.eclipse.jdt.junit	/org.eclipse.j	/JU...	0	1	.0	4.5
11	org.eclipse.jdt.junit	/org.eclipse.j	/La...	0	1	.0	1.1666666666666667
12	org.eclipse.jdt.junit	/org.eclipse.j	/La...	0	1	.0	2.6666666666666665
13	org.eclipse.jdt.junit	/org.eclipse.j	/Pr...	0	1	.0	2.3333333333333335
14	org.eclipse.jdt.junit	/org.eclipse.j	/Ty...	0	1	.0	2.6666666666666665

Correlation Refresher

- Non-parametric correlation: Spearman rank correlation
- Linear relationship is not needed!
- Applies to non-normal distributions
- Available for ordinal data





Hypotheses & Statistical Significance Testing

- Is there a difference between two observations?
- Is this difference due to chance?
- Statistical significance testing helps us
- It a method to test a given hypothesis with data
- Parametric vs. non-parametric tests

$$H_0 : \mu = \mu_0 \; v.s. \; H_1 : \mu \neq \mu_0$$

Statistical Testing: Parametric Tests

- Parametric tests make assumptions regarding the data distributions
- Often they require normal distributions, e.g., *t-test*
- In this case a hypothesis is about the parameters of a normal distributions
- A system can handle 1000 requests per second (request per seconds are normally distributed with standard deviation 5: Request~N(1000,5))
- During the last day on average 1100 request per second observed
- Is this increase just by chance or do we have to adjust the system?

Statistical Testing: Non-Parametric Tests

- Non-parametric test assume no particular distributions
- In other words, they do not a priori determine the kind and number of parameters
- This is very helpful if data is not normally distributed
- ... or if data is ordinal
- Have less power than parametric tests
- Often software *engineering data* is *heavily skewed*

Statistical Testing: 1 Sample vs. 2 Samples

- 1 sample test
- 1 measured, observed value
- Test against an hypothetical, assumed value
- Measured 1100 requests/s is tested against the $H_0 : u = 1000/s$
- 2 samples test
- 2 measured, observed values are compared to each other: Difference?
- $D = X_1 - X_2$ is then tested against the $H_0 : D = 0$

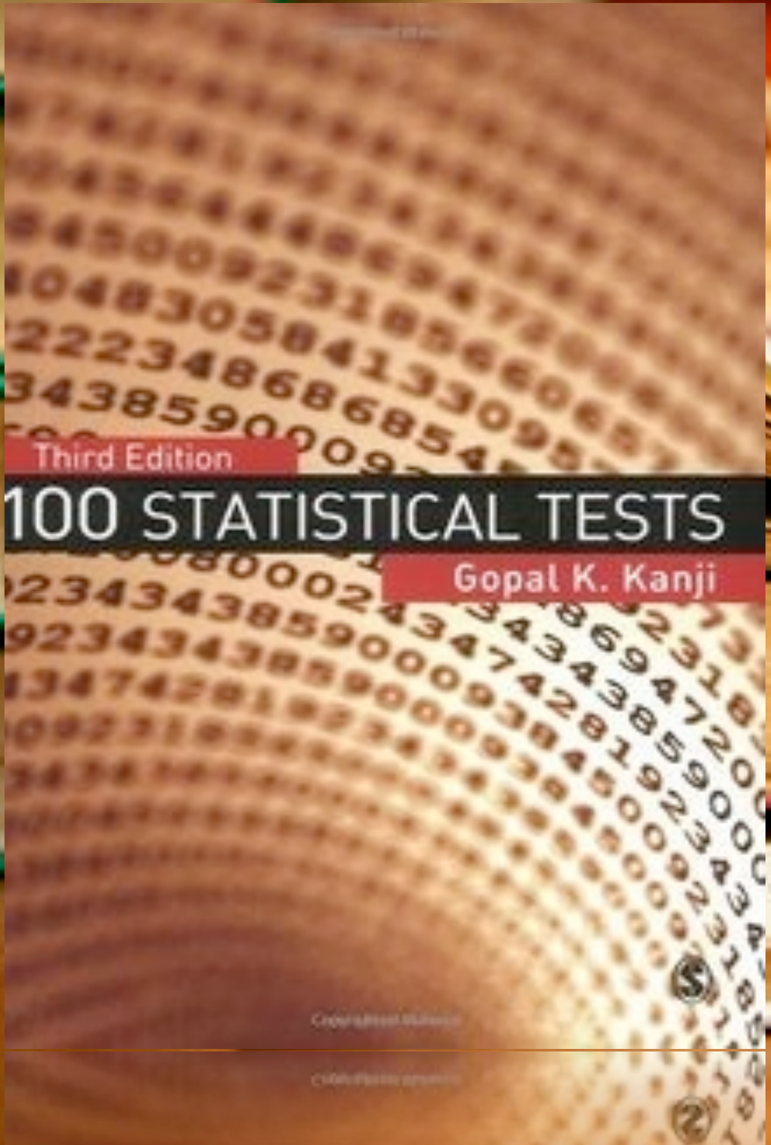
Statistical Testing: Paired vs. Non-Paired Tests

- Paired: The same group is measured twice (“before & after”)
- a.k.a related samples test
- more than 2 measurements are possible
- Question: Is there a difference after the treatment?
- “Does the number of defects increase over the releases?”
- “Are there now less defects in our system since we introduced test-driven development?”

Statistical Testing: Paired vs. Non-Paired Tests

- Non-paired: Two different groups are measured
- a.k.a independent samples test
- More than just 2 groups are possible
- Question: Is there a difference between to groups?
- “Do reviews find more defects than pair-programming?”
- “Which cost model is more accurate and delivers better forecasts?”
- “Is an iterative development process better than a linear one?”

Choosing the *correct* statistical test is *essential*!



Third Edition

100 STATISTICAL TESTS

Gopal K. Kanji

Lothar Papula

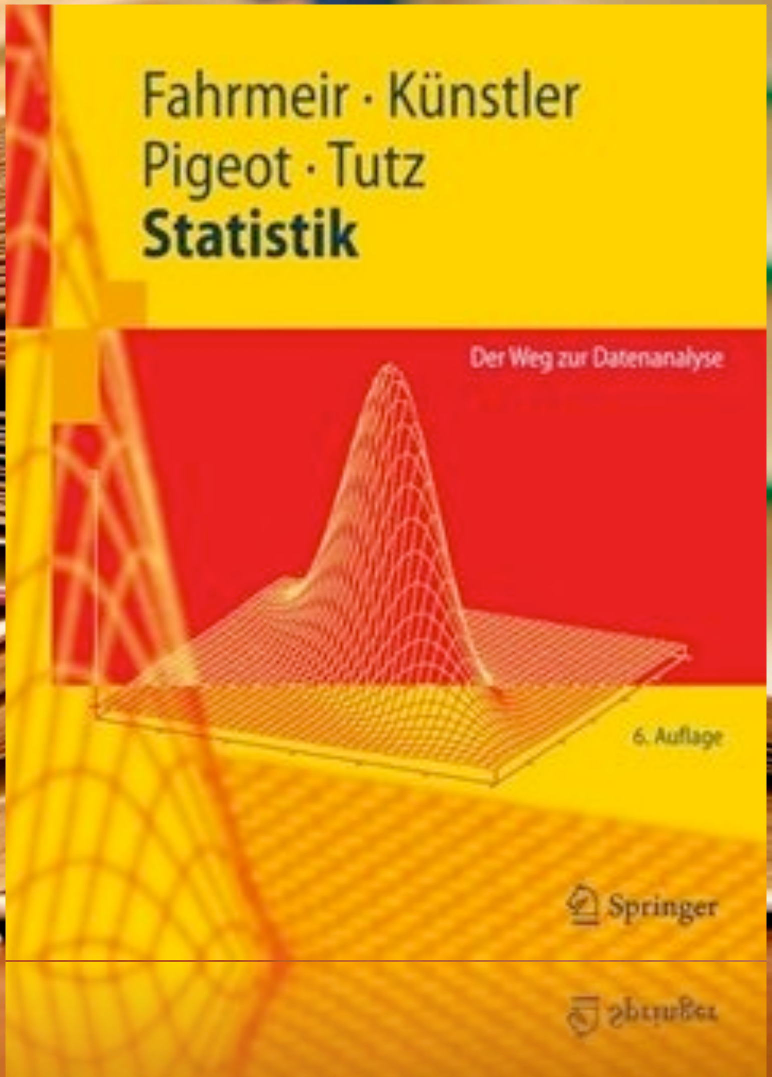
Mathematik für Ingenieure und Naturwissenschaftler Band 3

Vektoranalysis, Wahrscheinlichkeitsrechnung, Mathematische Statistik, Fehler- und Ausgleichsrechnung

5. Auflage

STUDIUM

VIEWEG+ TEUBNER



Fahrmeir · Künstler
Pigeot · Tutz
Statistik

Der Weg zur Datenanalyse

6. Auflage

Springer

I see defects in your future and code changes!

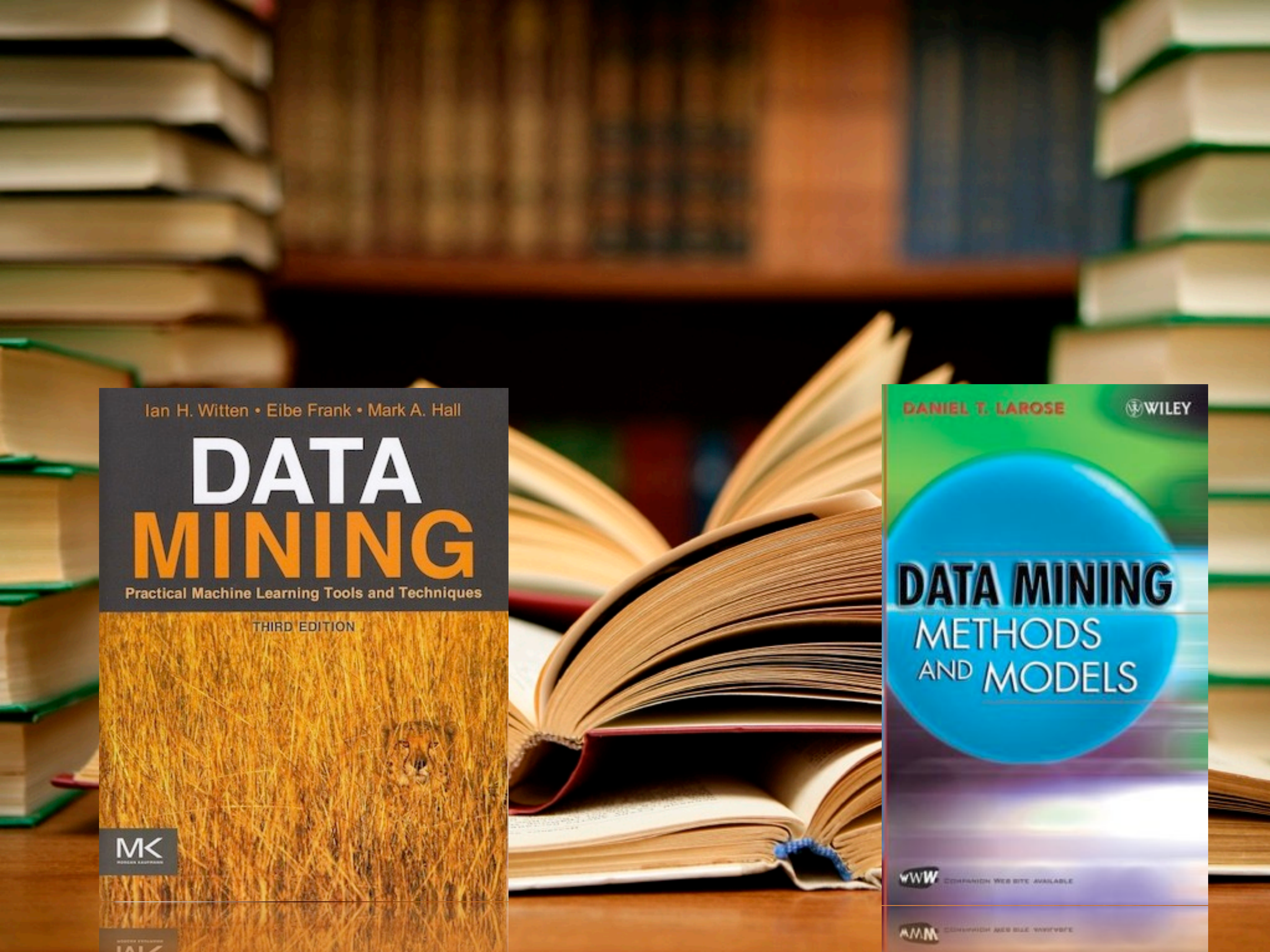
Regression Models

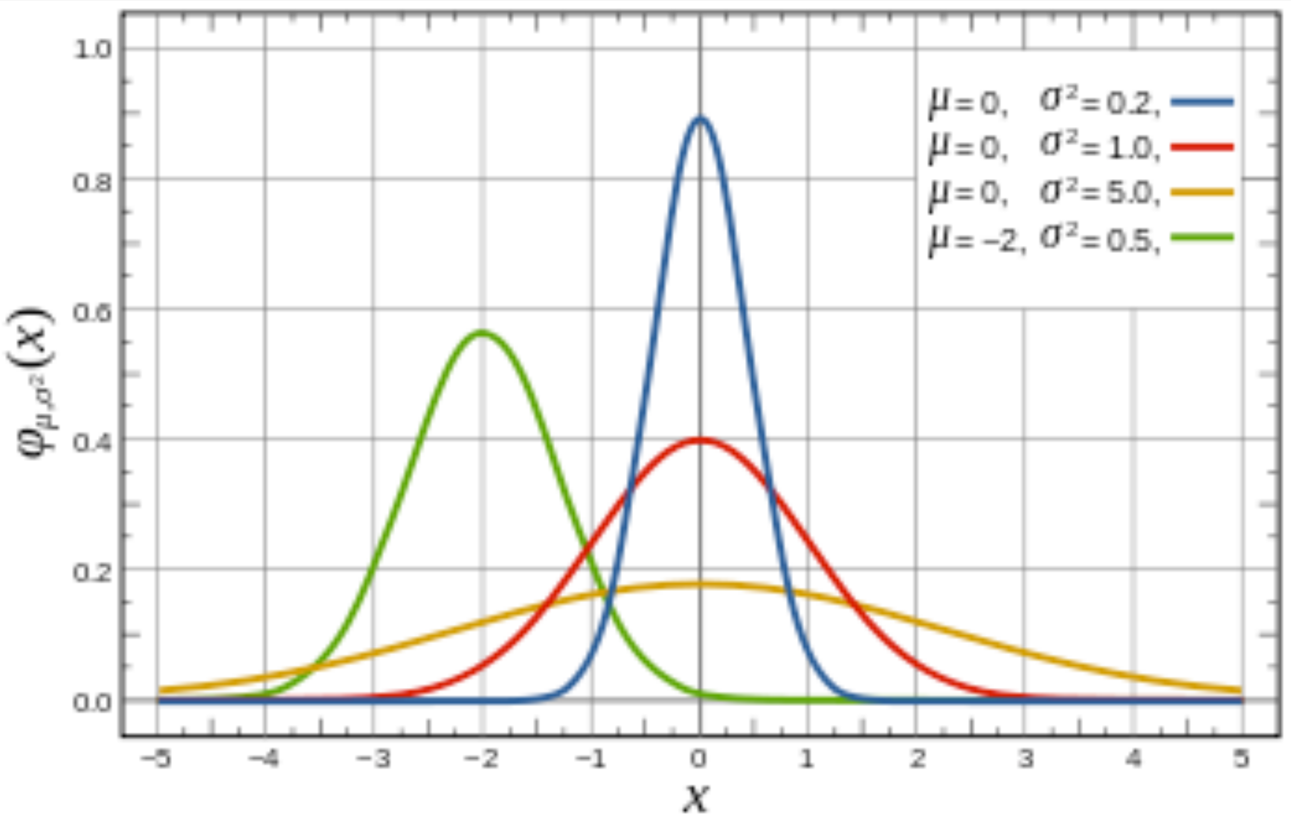
Data Mining Models

Time Series

Clustering

I see defects in your future and code changes!


Ian H. Witten • Eibe Frank • Mark A. Hall
DATA MINING
Practical Machine Learning Tools and Techniques
THIRD EDITION
MK
DANIEL T. LAROSE
WILEY
DATA MINING
METHODS AND MODELS
WWW COMPANION WEB SITE AVAILABLE



Empirical SE is statistics (researcher's view)

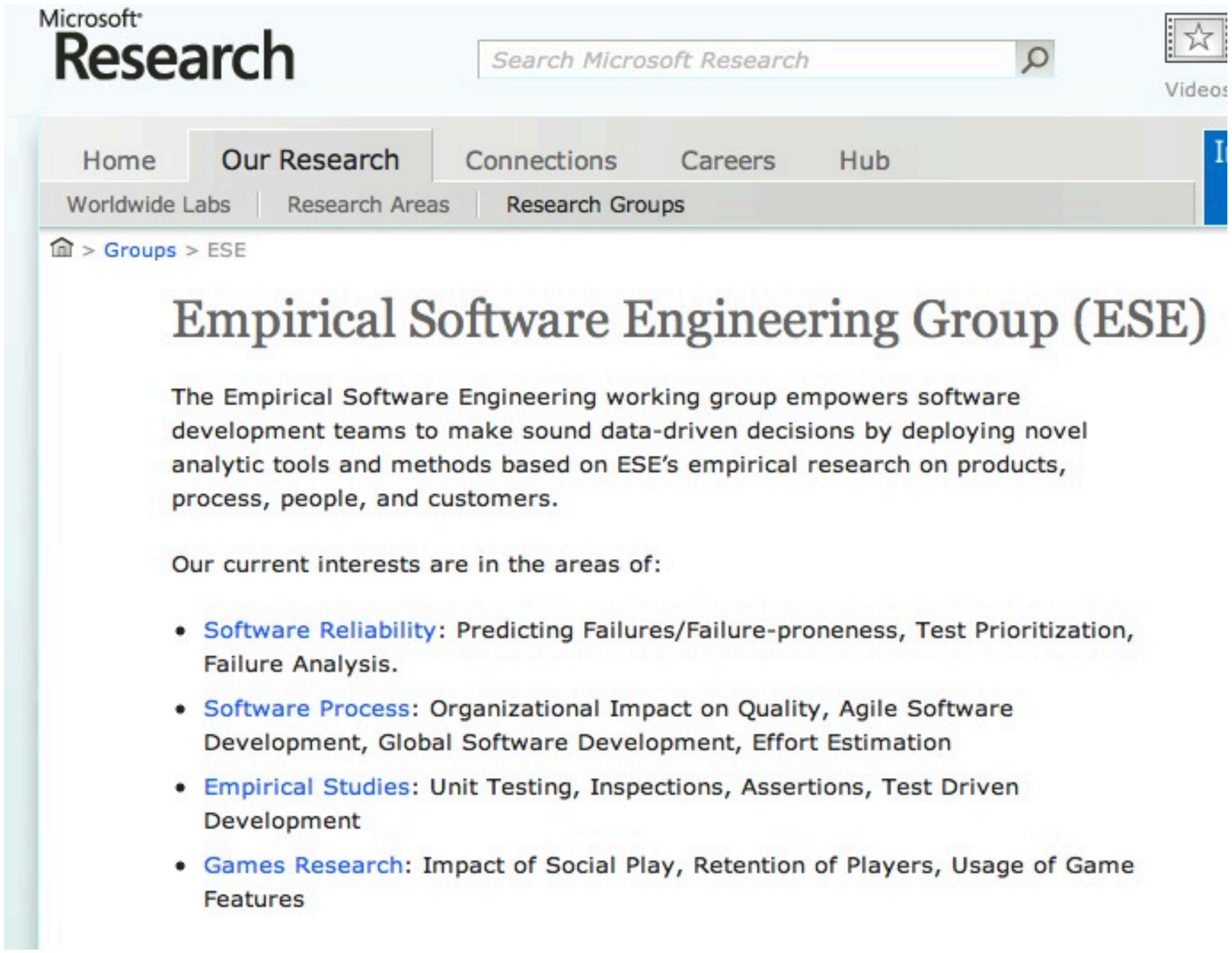
Empirical SE is business analytics (practitioner's view)

Quality Control in SE

Capability Maturity Model

ISO/IEC 15504 (SPICE)

IBM Cleanroom process



Empirical Software Engineering Group (ESE)

The Empirical Software Engineering working group empowers software development teams to make sound data-driven decisions by deploying novel analytic tools and methods based on ESE's empirical research on products, process, people, and customers.

Our current interests are in the areas of:

- Software Reliability: Predicting Failures/Failure-proneness, Test Prioritization, Failure Analysis.
- Software Process: Organizational Impact on Quality, Agile Software Development, Global Software Development, Effort Estimation
- Empirical Studies: Unit Testing, Inspections, Assertions, Test Driven Development
- Games Research: Impact of Social Play, Retention of Players, Usage of Game Features

http://research.microsoft.com/en-us/groups/ese/

Let's talk about data!

Where does it come from?

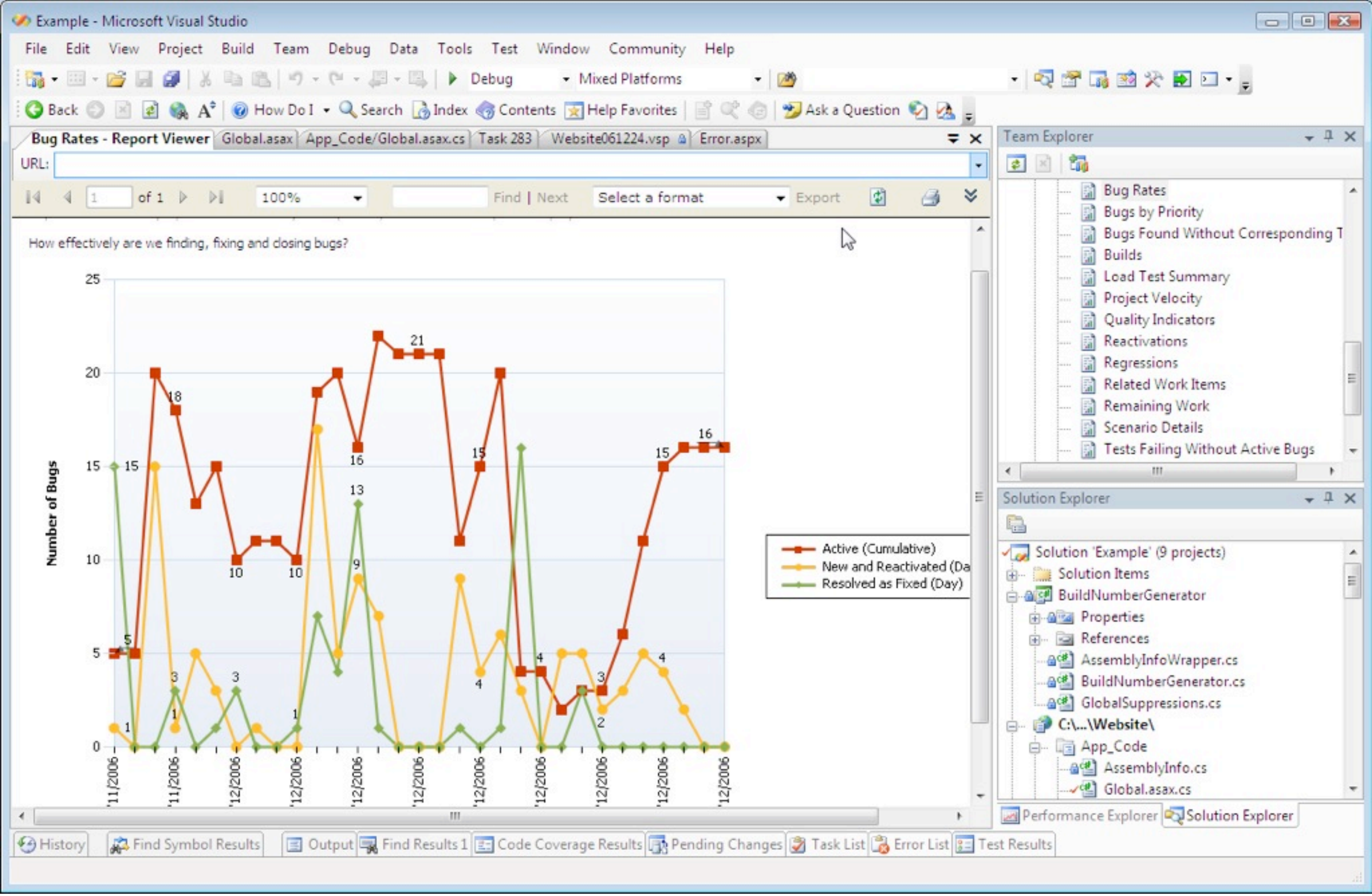
Software Repositories

- Versioning Systems
    - Git, Mercurial, Svn, Cvs, Team Foundation Server
- Issue Tracker
    - Bugzilla, Jiira, Redmine
- Feature Tracker
- Mailing Lists
- Discussion Boards

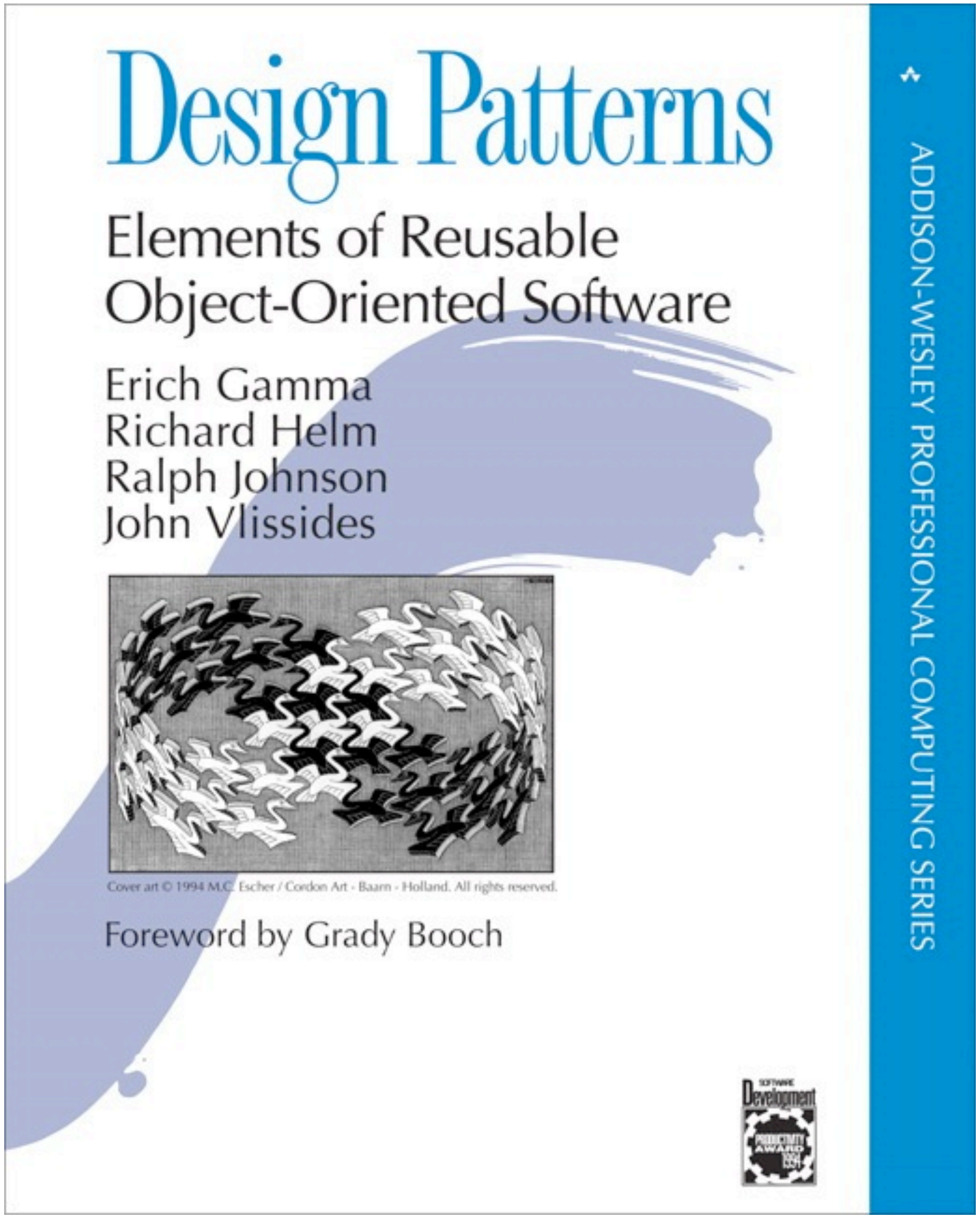
Other Data sources

- Interview with developers
- Controlled Experiments
- Observing
- Surveys
- ...

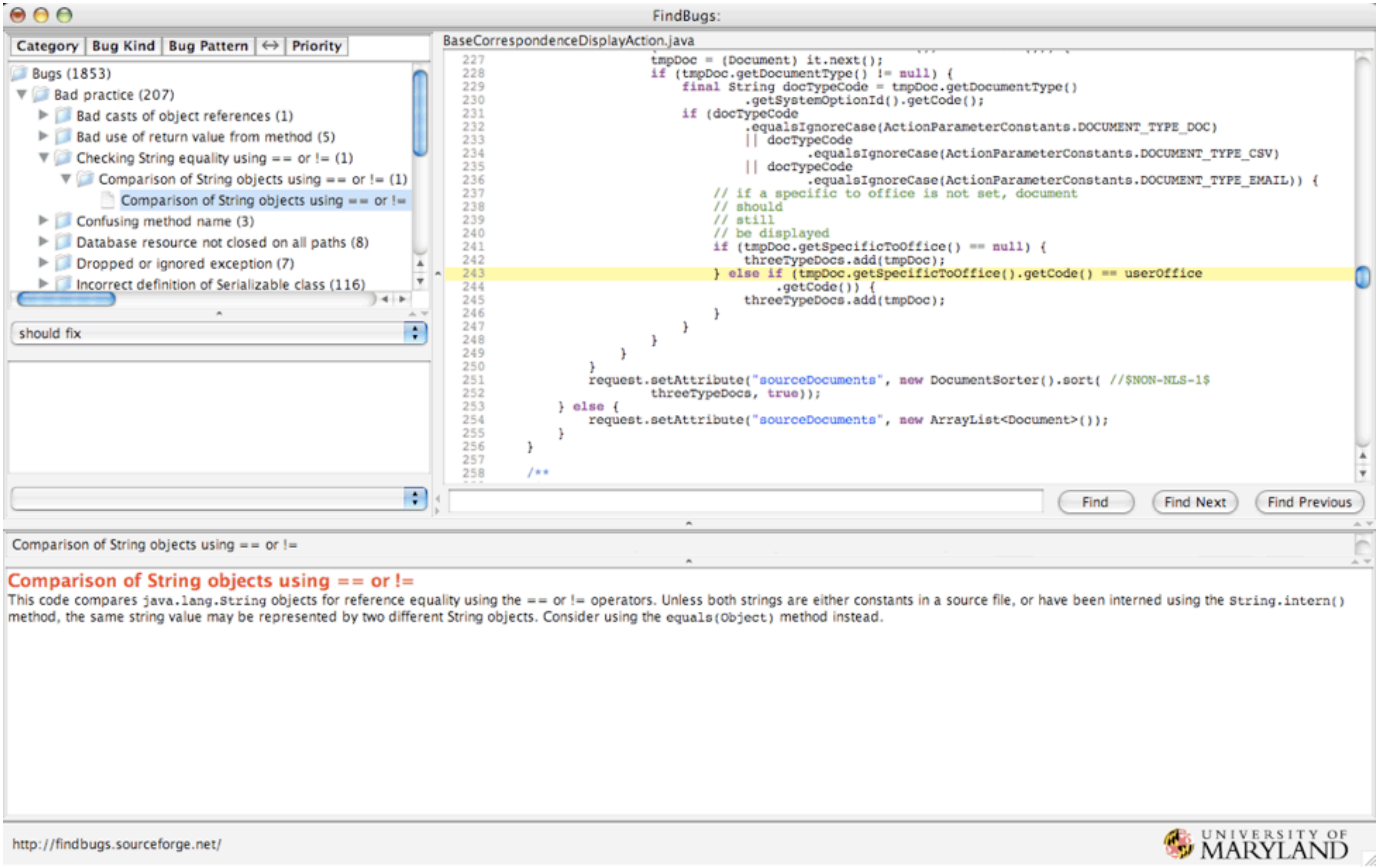
Examples of Empirical SE Results



Microsoft Team Foundation Server (TFS) provides a full-fledged data warehouse for development data.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns


FindBugs:
Category
Bug Kind
Bug Pattern
↔
Priority
Bugs (1853)
Bad practice (207)
Bad casts of object references (1)
Bad use of return value from method (5)
Checking String equality using == or != (1)
Comparison of String objects using == or != (1)
Comparison of String objects using == or !=
Confusing method name (3)
Database resource not closed on all paths (8)
Dropped or ignored exception (7)
Incorrect definition of Serializable class (116)
should fix
BaseCorrespondenceDisplayAction.java
Find
Find Next
Find Previous
http://findbugs.sourceforge.net/
UNIVERSITY OF MARYLAND

```java
                    tmpDoc = (Document) it.next();
                    if (tmpDoc.getDocumentType() != null) {
                        final String docTypeCode = tmpDoc.getDocumentType()
                                .getSystemOptionId().getCode();
                        if (docTypeCode
                                .equalsIgnoreCase(ActionParameterConstants.DOCUMENT_TYPE_DOC)
                                || docTypeCode
                                        .equalsIgnoreCase(ActionParameterConstants.DOCUMENT_TYPE_CSV)
                                || docTypeCode
                                        .equalsIgnoreCase(ActionParameterConstants.DOCUMENT_TYPE_EMAIL)) {
                            // if a specific to office is not set, document
                            // should
                            // still
                            // be displayed
                            if (tmpDoc.getSpecificToOffice() == null) {
                                threeTypeDocs.add(tmpDoc);
                            } else if (tmpDoc.getSpecificToOffice().getCode() == userOffice
                                    .getCode()) {
                                threeTypeDocs.add(tmpDoc);
                            }
                        }
                    }
                }
            }
            request.setAttribute("sourceDocuments", new DocumentSorter().sort( //$NON-NLS-1$
                    threeTypeDocs, true));
        } else {
            request.setAttribute("sourceDocuments", new ArrayList<Document>());
        }
    }

    /**
```

Comparison of String objects using == or !=

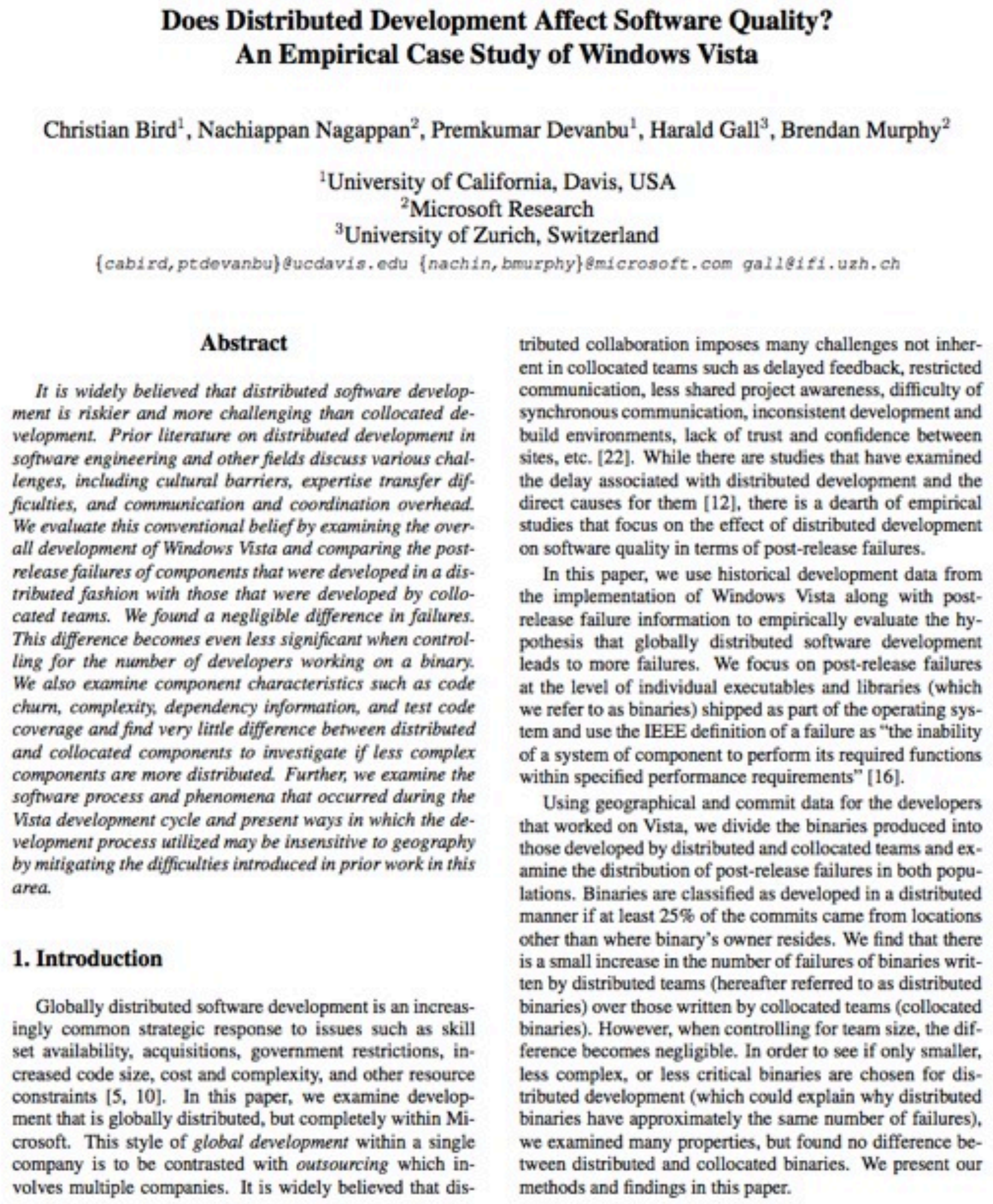
Comparison of String objects using == or !=

This code compares `java.lang.String` objects for reference equality using the == or != operators. Unless both strings are either constants in a source file, or have been interned using the `String.intern()` method, the same string value may be represented by two different String objects. Consider using the `equals(Object)` method instead.

Find Bugs: Catalogue of code idioms that often lead to bugs

The Tale of Distributed Development

- Common wisdom: Distributed development is riskier; leads to more defects
- Organizational challenges
- Knowledge sharing
- Communication problems
- *No difference* between distributed developed binaries and collocated binaries



Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista

Christian Bird[1], Nachiappan Nagappan[2], Premkumar Devanbu[1], Harald Gall[3], Brendan Murphy[2]

[1]University of California, Davis, USA
[2]Microsoft Research
[3]University of Zurich, Switzerland

{cabird,ptdevanbu}@ucdavis.edu {nachin,bmurphy}@microsoft.com gall@ifi.uzh.ch

Abstract

It is widely believed that distributed software development is riskier and more challenging than collocated development. Prior literature on distributed development in software engineering and other fields discuss various challenges, including cultural barriers, expertise transfer difficulties, and communication and coordination overhead. We evaluate this conventional belief by examining the overall development of Windows Vista and comparing the post-release failures of components that were developed in a distributed fashion with those that were developed by collocated teams. We found a negligible difference in failures. This difference becomes even less significant when controlling for the number of developers working on a binary. We also examine component characteristics such as code churn, complexity, dependency information, and test code coverage and find very little difference between distributed and collocated components to investigate if less complex components are more distributed. Further, we examine the software process and phenomena that occurred during the Vista development cycle and present ways in which the development process utilized may be insensitive to geography by mitigating the difficulties introduced in prior work in this area.

1. Introduction

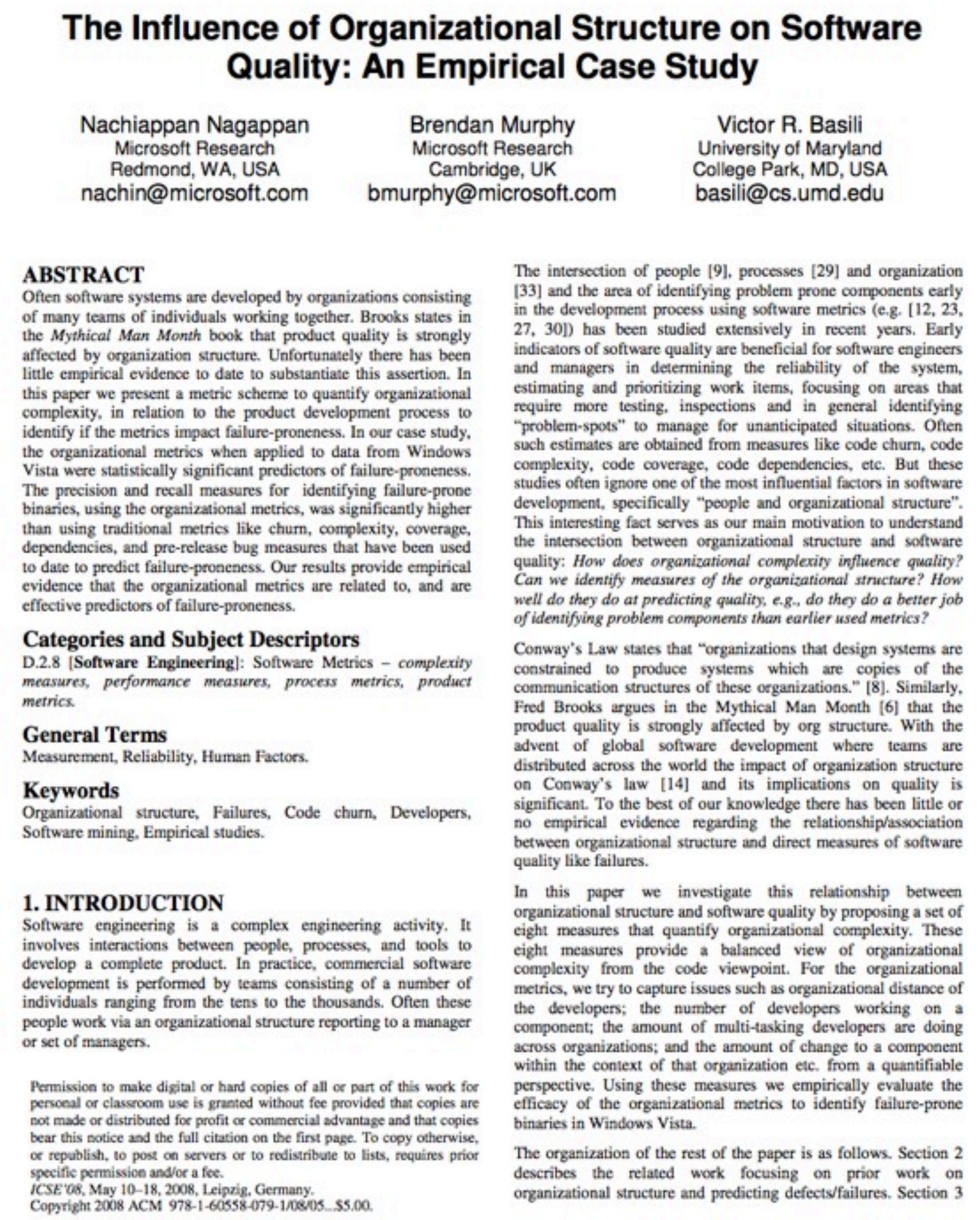
Globally distributed software development is an increasingly common strategic response to issues such as skill set availability, acquisitions, government restrictions, increased code size, cost and complexity, and other resource constraints [5, 10]. In this paper, we examine development that is globally distributed, but completely within Microsoft. This style of *global development* within a single company is to be contrasted with *outsourcing* which involves multiple companies. It is widely believed that distributed collaboration imposes many challenges not inherent in collocated teams such as delayed feedback, restricted communication, less shared project awareness, difficulty of synchronous communication, inconsistent development and build environments, lack of trust and confidence between sites, etc. [22]. While there are studies that have examined the delay associated with distributed development and the direct causes for them [12], there is a dearth of empirical studies that focus on the effect of distributed development on software quality in terms of post-release failures.

In this paper, we use historical development data from the implementation of Windows Vista along with post-release failure information to empirically evaluate the hypothesis that globally distributed software development leads to more failures. We focus on post-release failures at the level of individual executables and libraries (which we refer to as binaries) shipped as part of the operating system and use the IEEE definition of a failure as "the inability of a system of component to perform its required functions within specified performance requirements" [16].

Using geographical and commit data for the developers that worked on Vista, we divide the binaries produced into those developed by distributed and collocated teams and examine the distribution of post-release failures in both populations. Binaries are classified as developed in a distributed manner if at least 25% of the commits came from locations other than where binary's owner resides. We find that there is a small increase in the number of failures of binaries written by distributed teams (hereafter referred to as distributed binaries) over those written by collocated teams (collocated binaries). However, when controlling for team size, the difference becomes negligible. In order to see if only smaller, less complex, or less critical binaries are chosen for distributed development (which could explain why distributed binaries have approximately the same number of failures), we examined many properties, but found no difference between distributed and collocated binaries. We present our methods and findings in this paper.

The Tale of Organizational Structure

- In theory: Organizational structure could influence software quality:
- #Ex-SW-Engineers
- Strength of Code Ownership
- Degree of responsibility
- Structure *is related to defect proneness*



The Influence of Organizational Structure on Software Quality: An Empirical Case Study

Nachiappan Nagappan
Microsoft Research
Redmond, WA, USA
nachin@microsoft.com

Brendan Murphy
Microsoft Research
Cambridge, UK
bmurphy@microsoft.com

Victor R. Basili
University of Maryland
College Park, MD, USA
basili@cs.umd.edu

ABSTRACT
Often software systems are developed by organizations consisting of many teams of individuals working together. Brooks states in the *Mythical Man Month* book that product quality is strongly affected by organization structure. Unfortunately there has been little empirical evidence to date to substantiate this assertion. In this paper we present a metric scheme to quantify organizational complexity, in relation to the product development process to identify if the metrics impact failure-proneness. In our case study, the organizational metrics when applied to data from Windows Vista were statistically significant predictors of failure-proneness. The precision and recall measures for identifying failure-prone binaries, using the organizational metrics, was significantly higher than using traditional metrics like churn, complexity, coverage, dependencies, and pre-release bug measures that have been used to date to predict failure-proneness. Our results provide empirical evidence that the organizational metrics are related to, and are effective predictors of failure-proneness.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Software Metrics – *complexity measures, performance measures, process metrics, product metrics.*

General Terms
Measurement, Reliability, Human Factors.

Keywords
Organizational structure, Failures, Code churn, Developers, Software mining, Empirical studies.

1. INTRODUCTION
Software engineering is a complex engineering activity. It involves interactions between people, processes, and tools to develop a complete product. In practice, commercial software development is performed by teams consisting of a number of individuals ranging from the tens to the thousands. Often these people work via an organizational structure reporting to a manager or set of managers.

The intersection of people [9], processes [29] and organization [33] and the area of identifying problem prone components early in the development process using software metrics (e.g. [12, 23, 27, 30]) has been studied extensively in recent years. Early indicators of software quality are beneficial for software engineers and managers in determining the reliability of the system, estimating and prioritizing work items, focusing on areas that require more testing, inspections and in general identifying "problem-spots" to manage for unanticipated situations. Often such estimates are obtained from measures like code churn, code complexity, code coverage, code dependencies, etc. But these studies often ignore one of the most influential factors in software development, specifically "people and organizational structure". This interesting fact serves as our main motivation to understand the intersection between organizational structure and software quality: *How does organizational complexity influence quality? Can we identify measures of the organizational structure? How well do they do at predicting quality, e.g., do they do a better job of identifying problem components than earlier used metrics?*

Conway's Law states that "organizations that design systems are constrained to produce systems which are copies of the communication structures of these organizations." [8]. Similarly, Fred Brooks argues in the Mythical Man Month [6] that the product quality is strongly affected by org structure. With the advent of global software development where teams are distributed across the world the impact of organization structure on Conway's law [14] and its implications on quality is significant. To the best of our knowledge there has been little or no empirical evidence regarding the relationship/association between organizational structure and direct measures of software quality like failures.

In this paper we investigate this relationship between organizational structure and software quality by proposing a set of eight measures that quantify organizational complexity. These eight measures provide a balanced view of organizational complexity from the code viewpoint. For the organizational metrics, we try to capture issues such as organizational distance of the developers; the number of developers working on a component; the amount of multi-tasking developers are doing across organizations; and the amount of change to a component within the context of that organization etc. from a quantifiable perspective. Using these measures we empirically evaluate the efficacy of the organizational metrics to identify failure-prone binaries in Windows Vista.

The organization of the rest of the paper is as follows. Section 2 describes the related work focusing on prior work on organizational structure and predicting defects/failures. Section 3

The Tale of Code Metrics

- Assumption: The *bigger* and more *complex* modules are more prone to defects
- Those parts are more difficult to understand, and hence, to change
- Study with data from NASA software projects
- Size and complexity metrics are an indicator for defects

Data Mining Static Code Attributes to Learn Defect Predictors

Tim Menzies, *Member*, *IEEE*, Jeremy Greenwald, and Art Frank

Abstract—The value of using static code attributes to learn defect predictors has been widely debated. Prior work has explored issues like the merits of "McCabes versus Halstead versus lines of code counts" for generating defect predictors. We show here that such debates are irrelevant since *how* the attributes are used to build predictors is much more important than *which* particular attributes are used. Also, contrary to prior pessimism, we show that such defect predictors are demonstrably useful and, on the data studied here, yield predictors with a mean probability of detection of 71 percent and mean false alarms rates of 25 percent. These predictors would be useful for prioritizing a resource-bound exploration of code that has yet to be inspected.

Index Terms—Data mining detect prediction, McCabe, Halstead, artifical intelligence, empirical, naive Bayes.

1 Introduction

GIVEN recent research in artificial intelligence, it is now practical to use *data miners* to automatically learn predictors for software quality. When budget does not allow for complete testing of an entire system, software managers can use such predictors to focus the testing on parts of the system that seem defect-prone. These potential defect-prone trouble spots can then be examined in more detail by, say, model checking, intensive testing, etc.

The value of static code attributes as defect predictors has been widely debated. Some researchers endorse them ([1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]) while others vehemently oppose them ([21], [22]).

Prior studies may have reached different conclusions because they were based on different data. This potential conflation can now be removed since it is now possible to define a *baseline experiment* using public-domain data sets[1] which different researchers can use to compare their techniques.

This paper defines and motivates such a baseline. The baseline *definition* draws from standard practices in the data mining community [23], [24]. To *motivate* others to use our definition of a baseline experiment, we must demonstrate that it can yield interesting results. The baseline experiment of this article shows that the rule-based or decision-tree learning methods used in prior work [4], [13], [15], [16], [25] are clearly outperformed by a *naive Bayes* data miner with a *log-filtering* preprocessor on the numeric data (the terms in italics are defined later in this paper).

Further, the experiment can explain *why* our preferred Bayesian method performs best. That explanation is quite technical and comes from information theory. In this introduction, we need only say that the space of "best" predictors is "brittle," i.e., minor changes in the data (such as a slightly different sample used to learn a predictor) can make different attributes appear most useful for defect prediction.

This brittleness result offers a new insight on prior work. Prior results about defect predictors were so contradictory since they were drawn from a large space of competing conclusions with similar but distinct properties. Different studies could conclude that, say, lines of code are a better/worse predictor for defects than the McCabes complexity attribute, just because of small variations to the data. Bayesian methods smooth over the brittleness problem by polling numerous Gaussian approximations to the numerics distributions. Hence, Bayesian methods do not get confused by minor details about candidate predictors.

Our conclusion is that, contrary to prior pessimism [21], [22], data mining static code attributes to learn defect predictors is useful. Given our new results on naive Bayes and log-filtering, these predictors are much better than previously demonstrated. Also, prior contradictory results on the merits of defect predictors can be explained in terms of the brittleness of the space of "best" predictors. Further, our baseline experiment clearly shows that it is a misdirected discussion to debate, e.g., "lines of code versus McCabe" for predicting defects. As we shall see, *the choice of learning method* is far more important than *which subset of the available data* is used for learning.

2 Background

For this study, we learn defect predictors from static code attributes defined by McCabe [2] and Halstead [1]. McCabe and Halstead are "module"-based metrics, where a module

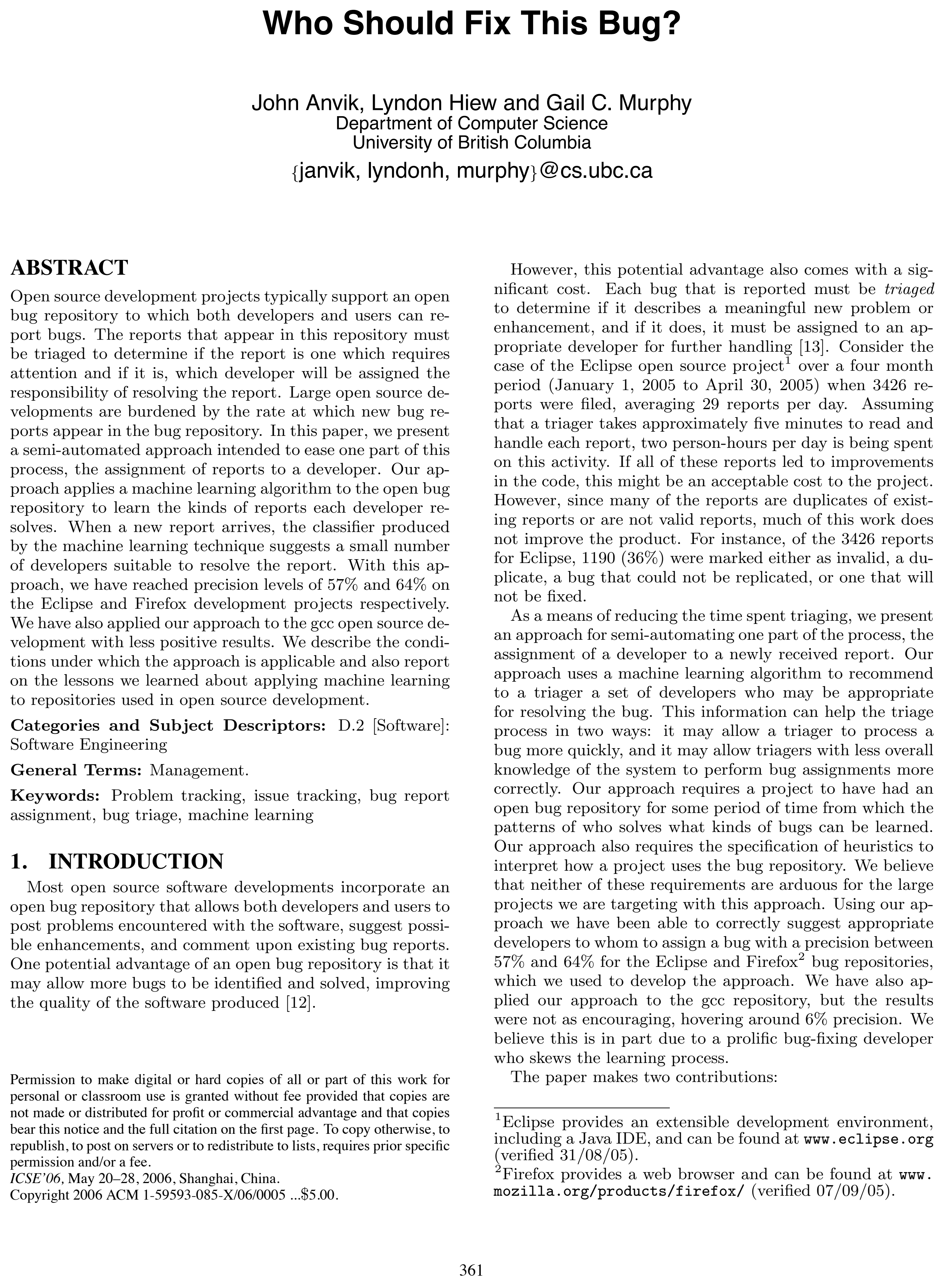
1. http://mdp.ivv.nasa.gov and http://promise.site.uottawa.ca/SERepository.

- T. Menzies is with the Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV 26506-610. E-mail: tim@menzies.us.
- J. Greenwald and A. Frank are with the Department of Computer Science, Portland State University, PO Box 751, Portland, OR 97207-0751. E-mail: jegreen@cecs.pdx.edu, arf@cs.pdx.edu.

Manuscript received 2 Jan. 2006; revised 9 Aug. 2006; accepted 13 Sept. 2006; published online DD Mmmm, YYYY.
Recommended for acceptance by M. Harman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0001-0106.

Who is the expert?

- Examining, assessing, and assigning a defect report to a developer costs time
- Who has to most knowledge?
- Learning from past fixed defect reports
- Result: Machine Learning approach suggesting to most appropriate developer



Who Should Fix This Bug?

John Anvik, Lyndon Hiew and Gail C. Murphy
Department of Computer Science
University of British Columbia
{janvik, lyndonh, murphy}@cs.ubc.ca

ABSTRACT

Open source development projects typically support an open bug repository to which both developers and users can report bugs. The reports that appear in this repository must be triaged to determine if the report is one which requires attention and if it is, which developer will be assigned the responsibility of resolving the report. Large open source developments are burdened by the rate at which new bug reports appear in the bug repository. In this paper, we present a semi-automated approach intended to ease one part of this process, the assignment of reports to a developer. Our approach applies a machine learning algorithm to the open bug repository to learn the kinds of reports each developer resolves. When a new report arrives, the classifier produced by the machine learning technique suggests a small number of developers suitable to resolve the report. With this approach, we have reached precision levels of 57% and 64% on the Eclipse and Firefox development projects respectively. We have also applied our approach to the gcc open source development with less positive results. We describe the conditions under which the approach is applicable and also report on the lessons we learned about applying machine learning to repositories used in open source development.

Categories and Subject Descriptors: D.2 [Software]: Software Engineering

General Terms: Management.

Keywords: Problem tracking, issue tracking, bug report assignment, bug triage, machine learning

1. INTRODUCTION

Most open source software developments incorporate an open bug repository that allows both developers and users to post problems encountered with the software, suggest possible enhancements, and comment upon existing bug reports. One potential advantage of an open bug repository is that it may allow more bugs to be identified and solved, improving the quality of the software produced [12].

However, this potential advantage also comes with a significant cost. Each bug that is reported must be *triaged* to determine if it describes a meaningful new problem or enhancement, and if it does, it must be assigned to an appropriate developer for further handling [13]. Consider the case of the Eclipse open source project[1] over a four month period (January 1, 2005 to April 30, 2005) when 3426 reports were filed, averaging 29 reports per day. Assuming that a triager takes approximately five minutes to read and handle each report, two person-hours per day is being spent on this activity. If all of these reports led to improvements in the code, this might be an acceptable cost to the project. However, since many of the reports are duplicates of existing reports or are not valid reports, much of this work does not improve the product. For instance, of the 3426 reports for Eclipse, 1190 (36%) were marked either as invalid, a duplicate, a bug that could not be replicated, or one that will not be fixed.

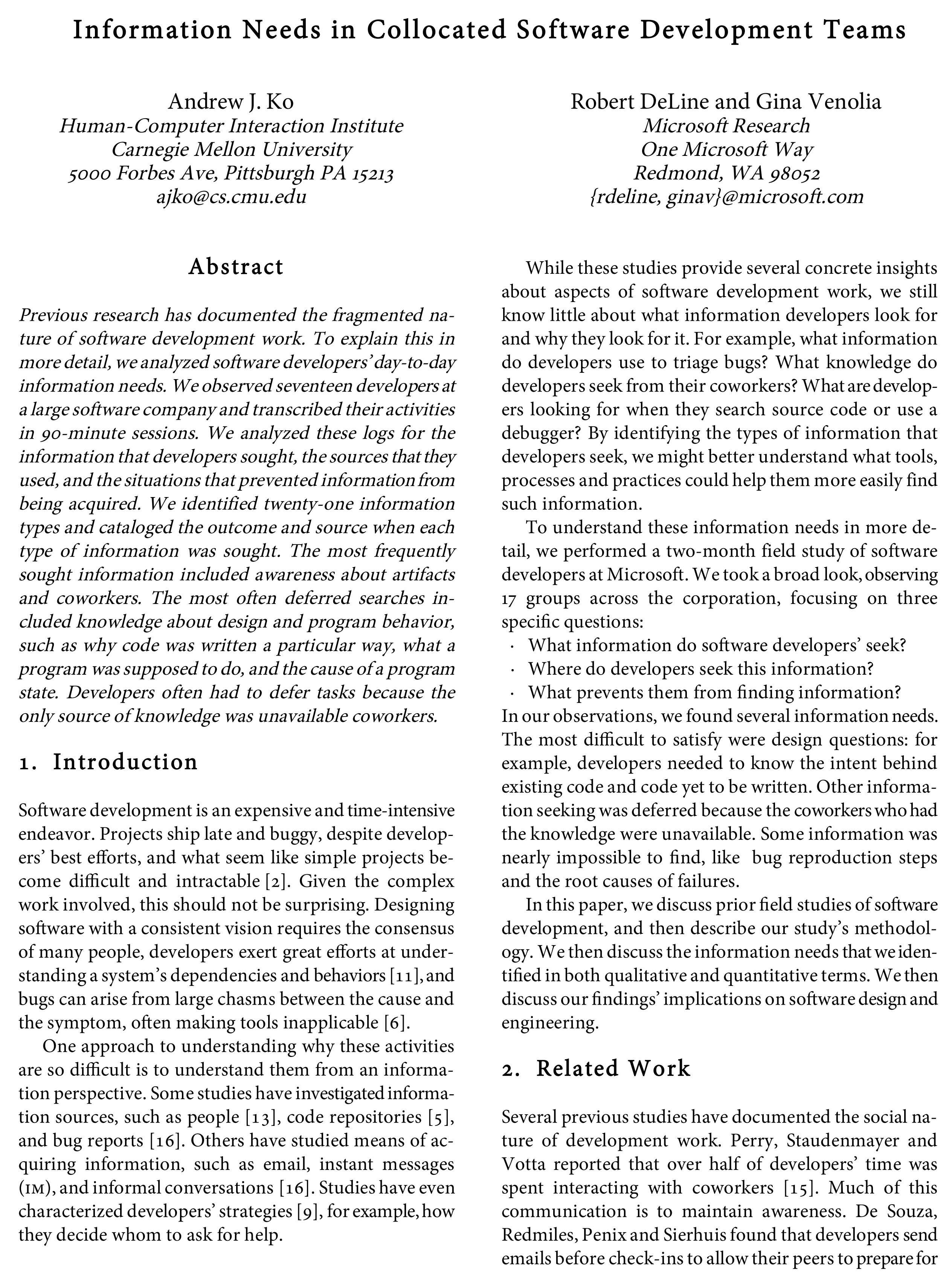
As a means of reducing the time spent triaging, we present an approach for semi-automating one part of the process, the assignment of a developer to a newly received report. Our approach uses a machine learning algorithm to recommend to a triager a set of developers who may be appropriate for resolving the bug. This information can help the triage process in two ways: it may allow a triager to process a bug more quickly, and it may allow triagers with less overall knowledge of the system to perform bug assignments more correctly. Our approach requires a project to have had an open bug repository for some period of time from which the patterns of who solves what kinds of bugs can be learned. Our approach also requires the specification of heuristics to interpret how a project uses the bug repository. We believe that neither of these requirements are arduous for the large projects we are targeting with this approach. Using our approach we have been able to correctly suggest appropriate developers to whom to assign a bug with a precision between 57% and 64% for the Eclipse and Firefox[2] bug repositories, which we used to develop the approach. We have also applied our approach to the gcc repository, but the results were not as encouraging, hovering around 6% precision. We believe this is in part due to a prolific bug-fixing developer who skews the learning process.

The paper makes two contributions:

[1]Eclipse provides an extensible development environment, including a Java IDE, and can be found at `www.eclipse.org` (verified 31/08/05).
[2]Firefox provides a web browser and can be found at `www.mozilla.org/products/firefox/` (verified 07/09/05).

Information? - *What information?*

- What information do developers seek when programming?
- How much time do developers spend for information gathering?
- Result: 1# “What are my co-developers working on?”
- Building tools that show what other developers are doing
- Facebook for developers?



Information Needs in Collocated Software Development Teams

Andrew J. Ko
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh PA 15213
ajko@cs.cmu.edu

Robert DeLine and Gina Venolia
Microsoft Research
One Microsoft Way
Redmond, WA 98052
{rdeline, ginav}@microsoft.com

Abstract

Previous research has documented the fragmented nature of software development work. To explain this in more detail, we analyzed software developers' day-to-day information needs. We observed seventeen developers at a large software company and transcribed their activities in 90-minute sessions. We analyzed these logs for the information that developers sought, the sources that they used, and the situations that prevented information from being acquired. We identified twenty-one information types and cataloged the outcome and source when each type of information was sought. The most frequently sought information included awareness about artifacts and coworkers. The most often deferred searches included knowledge about design and program behavior, such as why code was written a particular way, what a program was supposed to do, and the cause of a program state. Developers often had to defer tasks because the only source of knowledge was unavailable coworkers.

1. Introduction

Software development is an expensive and time-intensive endeavor. Projects ship late and buggy, despite developers' best efforts, and what seem like simple projects become difficult and intractable [2]. Given the complex work involved, this should not be surprising. Designing software with a consistent vision requires the consensus of many people, developers exert great efforts at understanding a system's dependencies and behaviors [11], and bugs can arise from large chasms between the cause and the symptom, often making tools inapplicable [6].

One approach to understanding why these activities are so difficult is to understand them from an information perspective. Some studies have investigated information sources, such as people [13], code repositories [5], and bug reports [16]. Others have studied means of acquiring information, such as email, instant messages (IM), and informal conversations [16]. Studies have even characterized developers' strategies [9], for example, how they decide whom to ask for help.

While these studies provide several concrete insights about aspects of software development work, we still know little about what information developers look for and why they look for it. For example, what information do developers use to triage bugs? What knowledge do developers seek from their coworkers? What are developers looking for when they search source code or use a debugger? By identifying the types of information that developers seek, we might better understand what tools, processes and practices could help them more easily find such information.

To understand these information needs in more detail, we performed a two-month field study of software developers at Microsoft. We took a broad look, observing 17 groups across the corporation, focusing on three specific questions:

- What information do software developers' seek?
- Where do developers seek this information?
- What prevents them from finding information?

In our observations, we found several information needs. The most difficult to satisfy were design questions: for example, developers needed to know the intent behind existing code and code yet to be written. Other information seeking was deferred because the coworkers who had the knowledge were unavailable. Some information was nearly impossible to find, like bug reproduction steps and the root causes of failures.

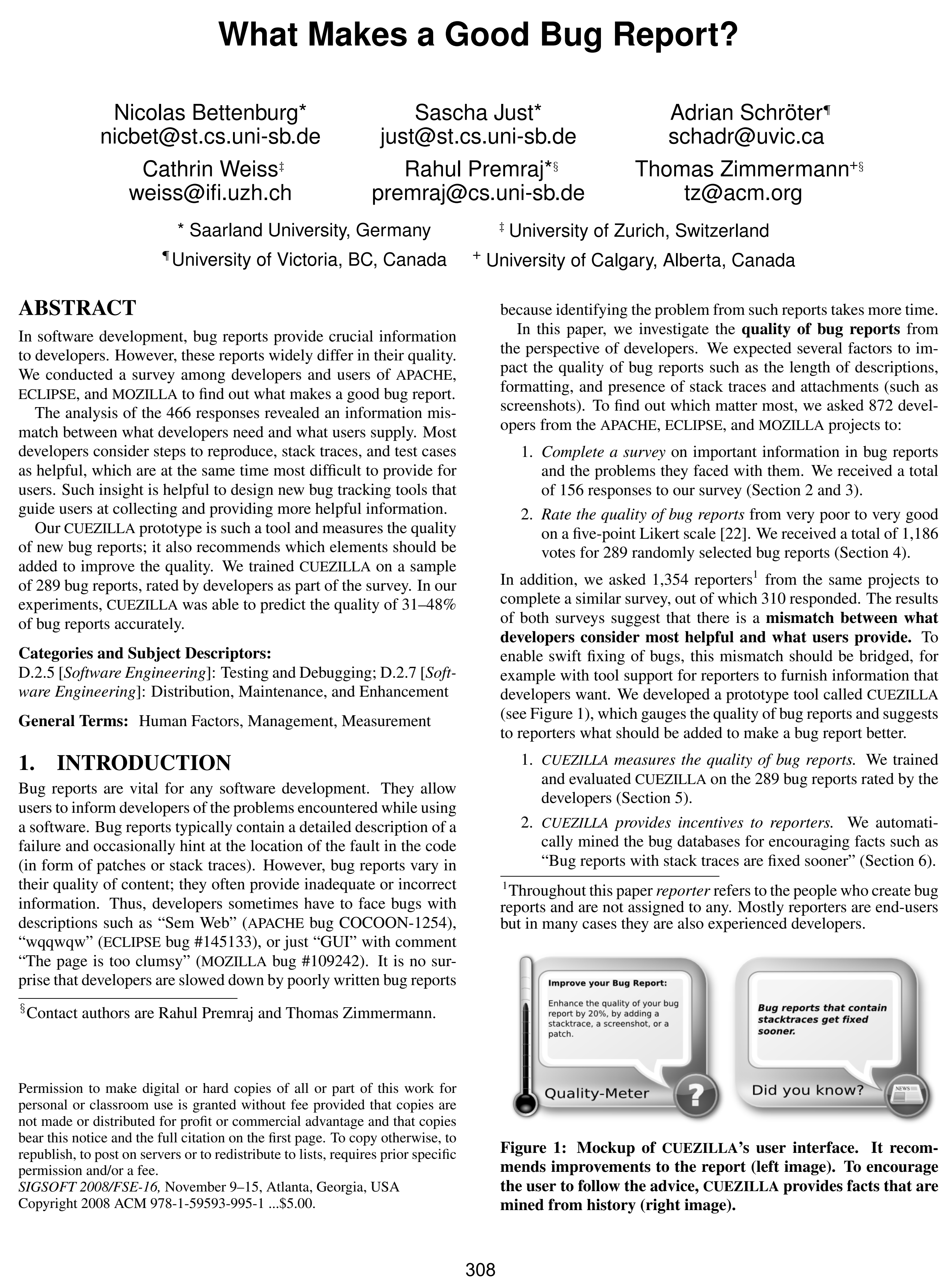
In this paper, we discuss prior field studies of software development, and then describe our study's methodology. We then discuss the information needs that we identified in both qualitative and quantitative terms. We then discuss our findings' implications on software design and engineering.

2. Related Work

Several previous studies have documented the social nature of development work. Perry, Staudenmayer and Votta reported that over half of developers' time was spent interacting with coworkers [15]. Much of this communication is to maintain awareness. De Souza, Redmiles, Penix and Sierhuis found that developers send emails before check-ins to allow their peers to prepare for

How to report defects?

- The *information in a bug* report is crucial to fixing defects
- Is defect reproducible?
- What OS & platform are affected?
- Which version of the software?
- Study shows what information makes a good bug report



What Makes a Good Bug Report?

Nicolas Bettenburg* nicbet@st.cs.uni-sb.de
Sascha Just* just@st.cs.uni-sb.de
Adrian Schröter¶ schadr@uvic.ca
Cathrin Weiss‡ weiss@ifi.uzh.ch
Rahul Premraj*§ premraj@cs.uni-sb.de
Thomas Zimmermann+§ tz@acm.org

* Saarland University, Germany
‡ University of Zurich, Switzerland
¶ University of Victoria, BC, Canada
+ University of Calgary, Alberta, Canada

ABSTRACT

In software development, bug reports provide crucial information to developers. However, these reports widely differ in their quality. We conducted a survey among developers and users of APACHE, ECLIPSE, and MOZILLA to find out what makes a good bug report.

The analysis of the 466 responses revealed an information mismatch between what developers need and what users supply. Most developers consider steps to reproduce, stack traces, and test cases as helpful, which are at the same time most difficult to provide for users. Such insight is helpful to design new bug tracking tools that guide users at collecting and providing more helpful information.

Our CUEZILLA prototype is such a tool and measures the quality of new bug reports; it also recommends which elements should be added to improve the quality. We trained CUEZILLA on a sample of 289 bug reports, rated by developers as part of the survey. In our experiments, CUEZILLA was able to predict the quality of 31–48% of bug reports accurately.

Categories and Subject Descriptors:
D.2.5 [*Software Engineering*]: Testing and Debugging; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement

General Terms: Human Factors, Management, Measurement

1. INTRODUCTION

Bug reports are vital for any software development. They allow users to inform developers of the problems encountered while using a software. Bug reports typically contain a detailed description of a failure and occasionally hint at the location of the fault in the code (in form of patches or stack traces). However, bug reports vary in their quality of content; they often provide inadequate or incorrect information. Thus, developers sometimes have to face bugs with descriptions such as "Sem Web" (APACHE bug COCOON-1254), "wqqwqw" (ECLIPSE bug #145133), or just "GUI" with comment "The page is too clumsy" (MOZILLA bug #109242). It is no surprise that developers are slowed down by poorly written bug reports because identifying the problem from such reports takes more time.

In this paper, we investigate the **quality of bug reports** from the perspective of developers. We expected several factors to impact the quality of bug reports such as the length of descriptions, formatting, and presence of stack traces and attachments (such as screenshots). To find out which matter most, we asked 872 developers from the APACHE, ECLIPSE, and MOZILLA projects to:

1. *Complete a survey* on important information in bug reports and the problems they faced with them. We received a total of 156 responses to our survey (Section 2 and 3).
2. *Rate the quality of bug reports* from very poor to very good on a five-point Likert scale [22]. We received a total of 1,186 votes for 289 randomly selected bug reports (Section 4).

In addition, we asked 1,354 reporters[1] from the same projects to complete a similar survey, out of which 310 responded. The results of both surveys suggest that there is a **mismatch between what developers consider most helpful and what users provide.** To enable swift fixing of bugs, this mismatch should be bridged, for example with tool support for reporters to furnish information that developers want. We developed a prototype tool called CUEZILLA (see Figure 1), which gauges the quality of bug reports and suggests to reporters what should be added to make a bug report better.

1. *CUEZILLA measures the quality of bug reports.* We trained and evaluated CUEZILLA on the 289 bug reports rated by the developers (Section 5).
2. *CUEZILLA provides incentives to reporters.* We automatically mined the bug databases for encouraging facts such as "Bug reports with stack traces are fixed sooner" (Section 6).

§Contact authors are Rahul Premraj and Thomas Zimmermann.

[1]Throughout this paper *reporter* refers to the people who create bug reports and are not assigned to any. Mostly reporters are end-users but in many cases they are also experienced developers.

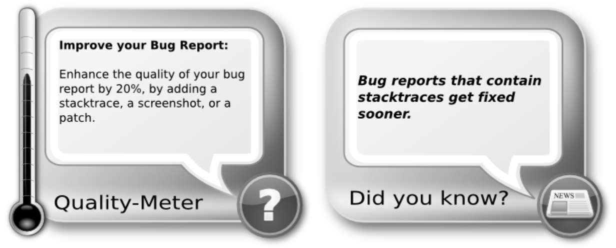
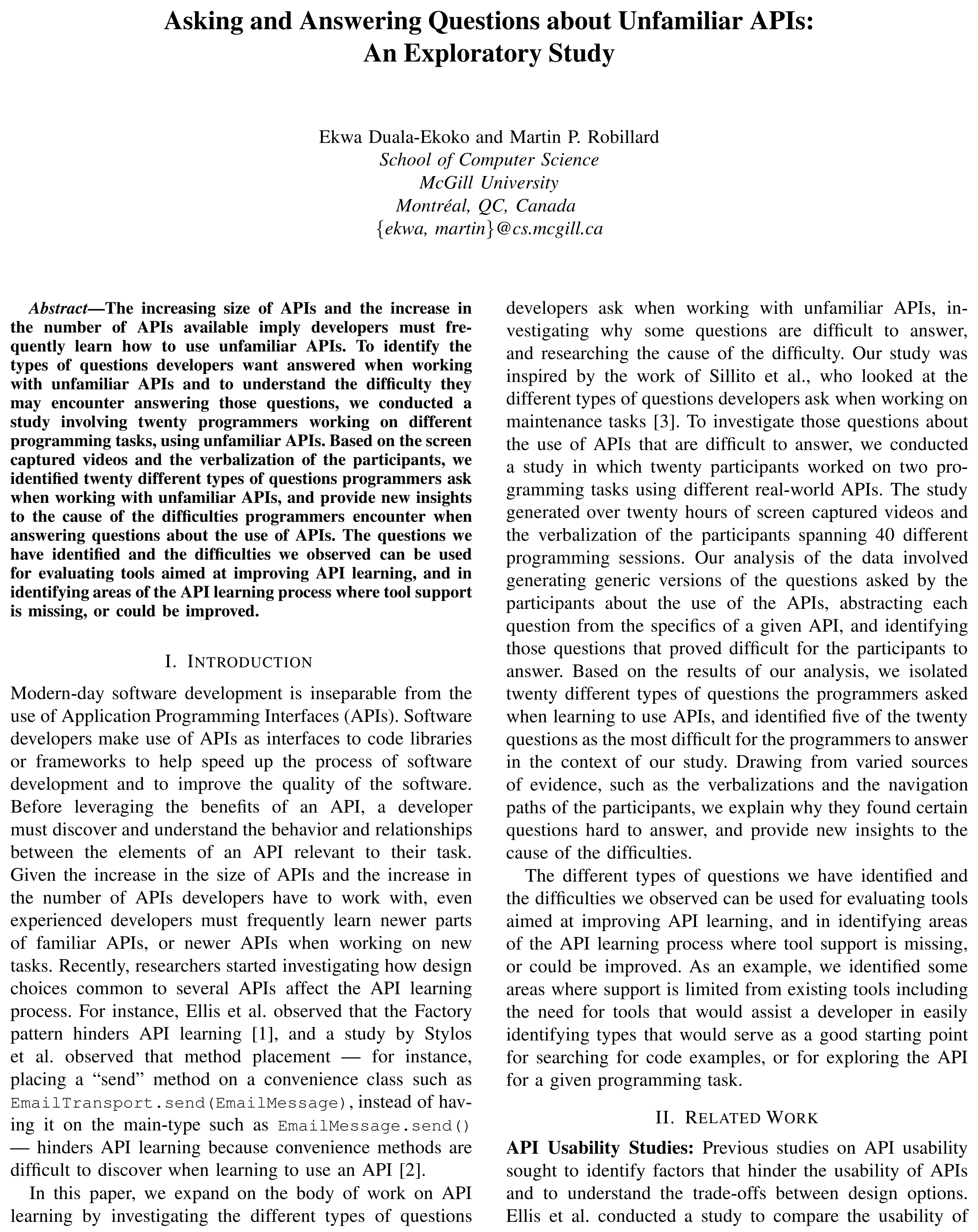

Improve your Bug Report:
Enhance the quality of your bug report by 20%, by adding a stacktrace, a screenshot, or a patch.
Quality-Meter
?
Bug reports that contain stacktraces get fixed sooner.
Did you know?

Figure 1: Mockup of CUEZILLA's user interface. It recommends improvements to the report (left image). To encourage the user to follow the advice, CUEZILLA provides facts that are mined from history (right image).

A new API?

- Getting used to a completely unfamiliar API is difficult
- What objects need to be instantiated?
- Which methods are required to implement a task?
- Based on the study they build a tool that supports developers on how to use an API



Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study

Ekwa Duala-Ekoko and Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
{*ekwa, martin*}*@cs.mcgill.ca*

***Abstract*—The increasing size of APIs and the increase in the number of APIs available imply developers must frequently learn how to use unfamiliar APIs. To identify the types of questions developers want answered when working with unfamiliar APIs and to understand the difficulty they may encounter answering those questions, we conducted a study involving twenty programmers working on different programming tasks, using unfamiliar APIs. Based on the screen captured videos and the verbalization of the participants, we identified twenty different types of questions programmers ask when working with unfamiliar APIs, and provide new insights to the cause of the difficulties programmers encounter when answering questions about the use of APIs. The questions we have identified and the difficulties we observed can be used for evaluating tools aimed at improving API learning, and in identifying areas of the API learning process where tool support is missing, or could be improved.**

I. Introduction

Modern-day software development is inseparable from the use of Application Programming Interfaces (APIs). Software developers make use of APIs as interfaces to code libraries or frameworks to help speed up the process of software development and to improve the quality of the software. Before leveraging the benefits of an API, a developer must discover and understand the behavior and relationships between the elements of an API relevant to their task. Given the increase in the size of APIs and the increase in the number of APIs developers have to work with, even experienced developers must frequently learn newer parts of familiar APIs, or newer APIs when working on new tasks. Recently, researchers started investigating how design choices common to several APIs affect the API learning process. For instance, Ellis et al. observed that the Factory pattern hinders API learning [1], and a study by Stylos et al. observed that method placement — for instance, placing a “send” method on a convenience class such as `EmailTransport.send(EmailMessage)`, instead of having it on the main-type such as `EmailMessage.send()` — hinders API learning because convenience methods are difficult to discover when learning to use an API [2].

In this paper, we expand on the body of work on API learning by investigating the different types of questions developers ask when working with unfamiliar APIs, investigating why some questions are difficult to answer, and researching the cause of the difficulty. Our study was inspired by the work of Sillito et al., who looked at the different types of questions developers ask when working on maintenance tasks [3]. To investigate those questions about the use of APIs that are difficult to answer, we conducted a study in which twenty participants worked on two programming tasks using different real-world APIs. The study generated over twenty hours of screen captured videos and the verbalization of the participants spanning 40 different programming sessions. Our analysis of the data involved generating generic versions of the questions asked by the participants about the use of the APIs, abstracting each question from the specifics of a given API, and identifying those questions that proved difficult for the participants to answer. Based on the results of our analysis, we isolated twenty different types of questions the programmers asked when learning to use APIs, and identified five of the twenty questions as the most difficult for the programmers to answer in the context of our study. Drawing from varied sources of evidence, such as the verbalizations and the navigation paths of the participants, we explain why they found certain questions hard to answer, and provide new insights to the cause of the difficulties.

The different types of questions we have identified and the difficulties we observed can be used for evaluating tools aimed at improving API learning, and in identifying areas of the API learning process where tool support is missing, or could be improved. As an example, we identified some areas where support is limited from existing tools including the need for tools that would assist a developer in easily identifying types that would serve as a good starting point for searching for code examples, or for exploring the API for a given programming task.

II. Related Work

API Usability Studies: Previous studies on API usability sought to identify factors that hinder the usability of APIs and to understand the trade-offs between design options. Ellis et al. conducted a study to compare the usability of

Lehman's Laws of SW Evolution & Eclipse

- Several Metrics of Eclipse for releases 2.0, 2.1, and 3.0[1]
- Law of *Continuing Growth*
- Law of *Increasing Complexity*
- Law of *Declining Quality*

[1]Predicting Defects for Eclipse *by T. Zimmermann, R. Premraj, A. Zeller*