Search-based Refactoring: Metrics are Not Enough

Chris Simons[1], Jeremy Singer[2], David White[2]

[1] Department of Computer Science and Creative Technologies,
University of the West of England, Bristol, BS16 1QY, United Kingdom
chris.simons@uwe.ac.uk

[2] School of Computing Science, University of the Glasgow, G12 8RZ, United Kingdom
{jeremy.singer,david.r.white}@glasgow.ac.uk

ssbse 2015

Symposium on Search-Based Software Engineering, Bergamo, Italy.
5 – 7 September 2015

Agenda

- Motivation
    - Role of structural metrics in search for refactoring
- Hypotheses
- Experimental Methodology
    - Survey of industrial software development practitioners
- Results
    - Quantitative and qualitative analysis
- Related Work
- Conclusions

Motivation

SBSE extensively applied to refactor object-oriented software. Metrics quantify structural properties such as cohesion, coupling, module dependencies, inheritance hierarchies etc.

SBSE refactoring of Apache Ant project does not improve design as assessed by an expert

Barros, M.0. and Farzat, F.A.: What Can a Big Program Teach Us about Optimization? *Proceedings of SSBSE 2013*, LCNS 8084. Springer (2013)

When SBSE refactoring is conducted using a number of cohesion metrics, there can be disagreement between metrics

O'Cinneide, M., Tratt, L., Harman, M., Counsell, S. and Moghadam, I.: Experimental Assessment of Software Metrics using Automated Refactoring. *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement.* ACM (2012)

Questions

Q: What’s the relationship between metrics and design quality?

Q: Qualitatively, how do software engineers make such judgements?

We try to answer these questions, by:
- placing ultimate judgement of software quality with industrial software engineers,
- measuring any correlation between metrics used in SBSE and human judgement,
- examining the articulated justifications for those judgements.

Signpost

- Motivation
    - Role of structural metrics in software refactoring
- Hypotheses
- Experimental Methodology
    - Survey of industrial software development practitioners
- Results
    - Quantitative and qualitative analysis reported
- Related Work
- Conclusions

We formulate our hypotheses such that the null hypothesis makes no assumption of an effect:

H_0: There is no correlation between software metric values and software engineer evaluation of quality for a given software metric.

H_1: There is a correlation between software metric values and software engineer evaluation of quality for a given software metric.

Signpost

- Motivation
  - Role of structural metrics in software refactoring
- Hypotheses
- Experimental Methodology
  - Survey of industrial software development practitioners
- Results
  - Quantitative and qualitative analysis reported
- Related Work
- Conclusions

Experimental Methodology

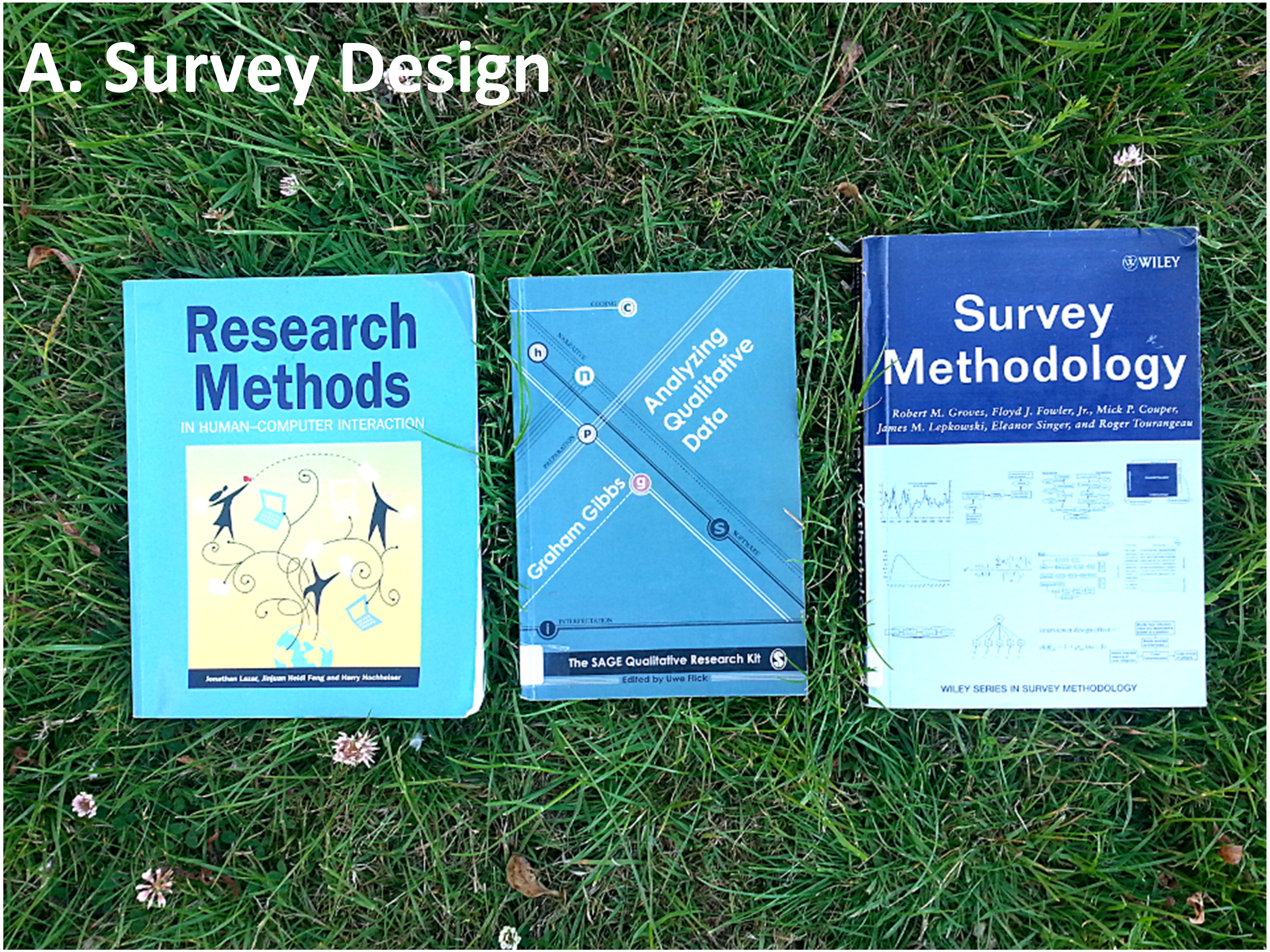
Three components:

A. Survey Design

B. Questionnaire Design

C. Survey Process

A. Survey Design


Research Methods
IN HUMAN–COMPUTER INTERACTION
Jonathan Lazar, Jinjuan Heidi Feng and Harry Hochheiser
Analyzing Qualitative Data
Graham Gibbs
The SAGE Qualitative Research Kit
Edited by Uwe Flick
WILEY
Survey Methodology
Robert M. Groves, Floyd J. Fowler, Jr., Mick P. Couper, James M. Lepkowski, Eleanor Singer, and Roger Tourangeau
WILEY SERIES IN SURVEY METHODOLOGY

A. Survey Design - selection of Software Designs

Balance of meaningful design versus cognitive overload

Two problem domains to strengthen generality of findings
- Automated Teller Machine (ATM)
- Nautical Cruise Booking System

Bjork, R.C., ATM Simulation, http://www.math-cs.gordon.edu/courses/cs211/ATMExample/

Apperly, H., Simons, C.L. et al. *Service- and Component-Based Development*. Addison-Wesley (2003)

Five experienced software engineers produced class diagrams for each problem domain – 10 diagrams in total – and metric values calculated for each class diagram.

All designs are available at: www.cems.uwe.ac.uk/~clsimons/MetricsAreNotEnough

A. Survey Design - selection of Design Qualities

Quality Model for Object-Oriented Design (QMOOD) relates six design quality attributes to corresponding properties and metrics.

Bansiya, J. and Davis, C.G.: A Hierarchical Model for Object-Oriented Design Quality Assessment. IEEE Transactions on Software Engineering. 28(1), 4-17 (2002)

To reduce cognitive load in survey, we focussed on the most problem-domain independent i.e.

Attribute	Definition
Reusability	Reflects the presence of object-oriented design characteristics that allow a design to be reapplied to a new problem without significant effort.
Flexibility	Characteristics that allow the incorporation of changes in a design. The ability of a design to be adapted to provide functionally related capabilities.
Understandability	The properties of a design that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure.

A. Survey Design - selection of Metrics

Q: What is the distribution of software metrics among the SBSE refactoring literature?

Search query of “software metrics” in SBSE Repository…

Zhang, Y.: SBSE Repository. http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/

…yielded ***57*** papers. Excluding non-refactoring sources narrowed the list to ***23***.

118 different metrics used. Only ***3*** (LCOM, MQ, EVM) used more than once, and ***1*** suite (QMOOD) used more than once.

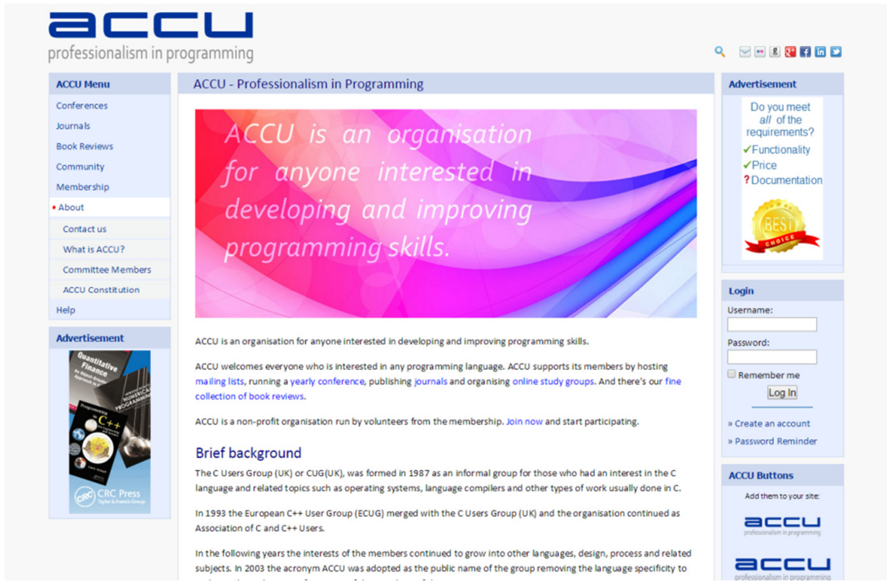
From these, we selected the QMOOD metrics Design Size in Classes (DSC), Direct Class Coupling (DCC) and Numbers of Methods (NOM).

Also selected:
- elegance metric Numbers Among Classes (NAC) (used by Barros & Farzat)
- Numbers of Attributes and Methods (NOAM) (to cater for attributes)

A. Survey Design – Target Population, Sample Frame

Association of C and C++ Users

www.accu.org



ACCU is an organisation for anyone interested in developing and improving programming skills.

ACCU welcomes everyone who is interested in any programming language. ACCU supports its members by hosting mailing lists, running a yearly conference, publishing journals and organising online study groups. And there's our fine collection of book reviews.

ACCU is a non-profit organisation run by volunteers from the membership. Join now and start participating.

Brief background

The C Users Group (UK) or CUG(UK), was formed in 1987 as an informal group for those who had an interest in the C language and related topics such as operating systems, language compilers and other types of work usually done in C.

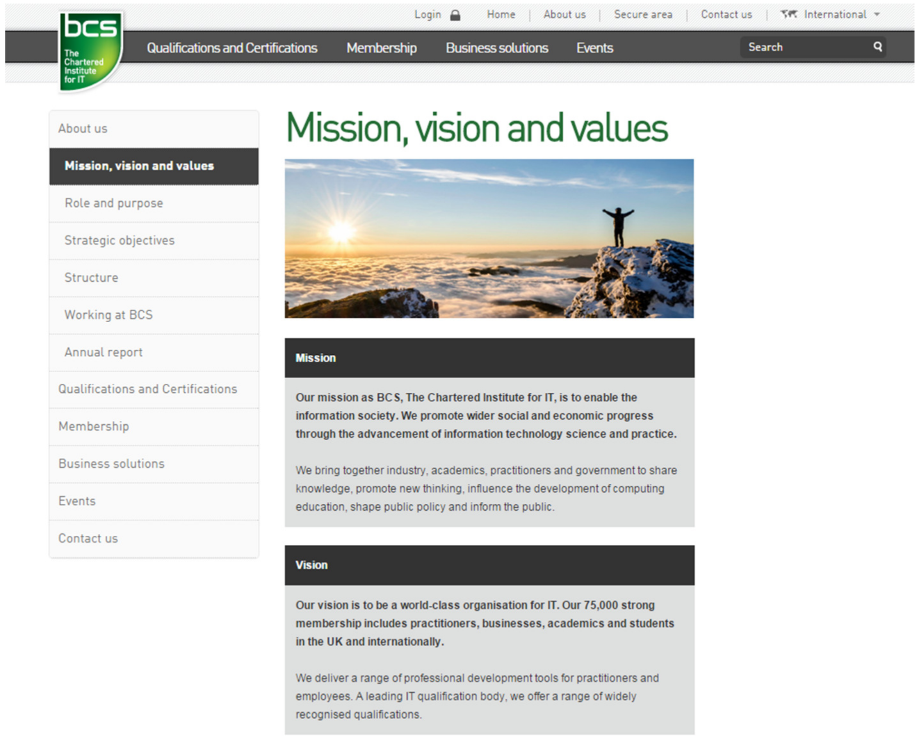
In 1993 the European C++ User Group (ECUG) merged with the C Users Group (UK) and the organisation continued as Association of C and C++ Users.

In the following years the interests of the members continued to grow into other languages, design, process and related subjects. In 2003 the acronym ACCU was adopted as the public name of the group removing the language specificity to

Approx. 900 members via email list

British Computer Society

www.bcs.org



Mission, vision and values

Mission

Our mission as BCS, The Chartered Institute for IT, is to enable the information society. We promote wider social and economic progress through the advancement of information technology science and practice.

We bring together industry, academics, practitioners and government to share knowledge, promote new thinking, influence the development of computing education, shape public policy and inform the public.

Vision

Our vision is to be a world-class organisation for IT. Our 75,000 strong membership includes practitioners, businesses, academics and students in the UK and internationally.

We deliver a range of professional development tools for practitioners and employees. A leading IT qualification body, we offer a range of widely recognised qualifications.

Approx. 11,000 members via LinkedIn Forum

B. Questionnaire Design

Designs presented at random to participants, one model per participant.

Likert Scale used to evaluate designs with seven levels:
"strongly disagree", "disagree", "somewhat disagree", "neutral", "somewhat agree", "agree" and "strongly agree"

https://en.wikipedia.org/wiki/Likert_scale

Informed consent from participants obtained. All survey data strictly confidential and published information reported either as aggregated data or anonymised.

Pretesting conducted with five experienced software engineers.

SurveyGismo used as survey platform.

Questionnaire available at: www.cems.uwe.ac.uk/~clsimons/MetricsAreNotEnough

C. Survey Process

Survey open from 18 January to 28 February 2015.

Participants posted lively comments to the BCS LinkedIn Forum e.g.

“it’s difficult to form an impression of design qualities using a class diagram in isolation of other development aspects e.g. dynamic models of behaviour, requirements, test plan etc.”

One forum contributor remarked:

“it seems that your idea of what quality is and how to judge it is not the same as many of us in the industry”

Signpost

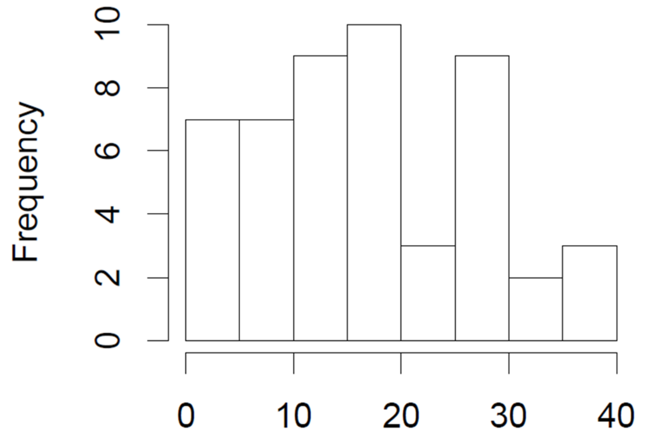
- Motivation
    - Role of structural metrics in software refactoring
- Hypotheses
- Experimental Methodology
    - Survey of industrial software development practitioners
- Results
    - Quantitative and qualitative analysis reported
- Related Work
- Conclusions

Quantitative Results

Anonymised data available

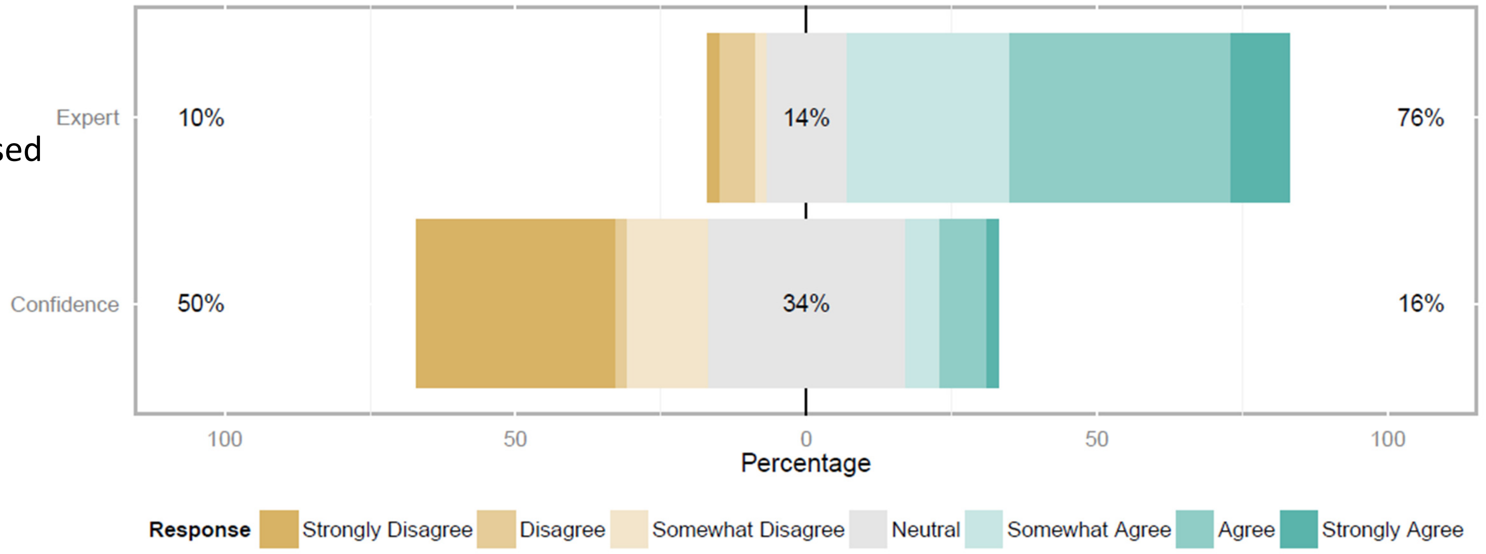
http://www.cems.uwe.ac.uk/~clsimons/MetricsAreNotEnough/

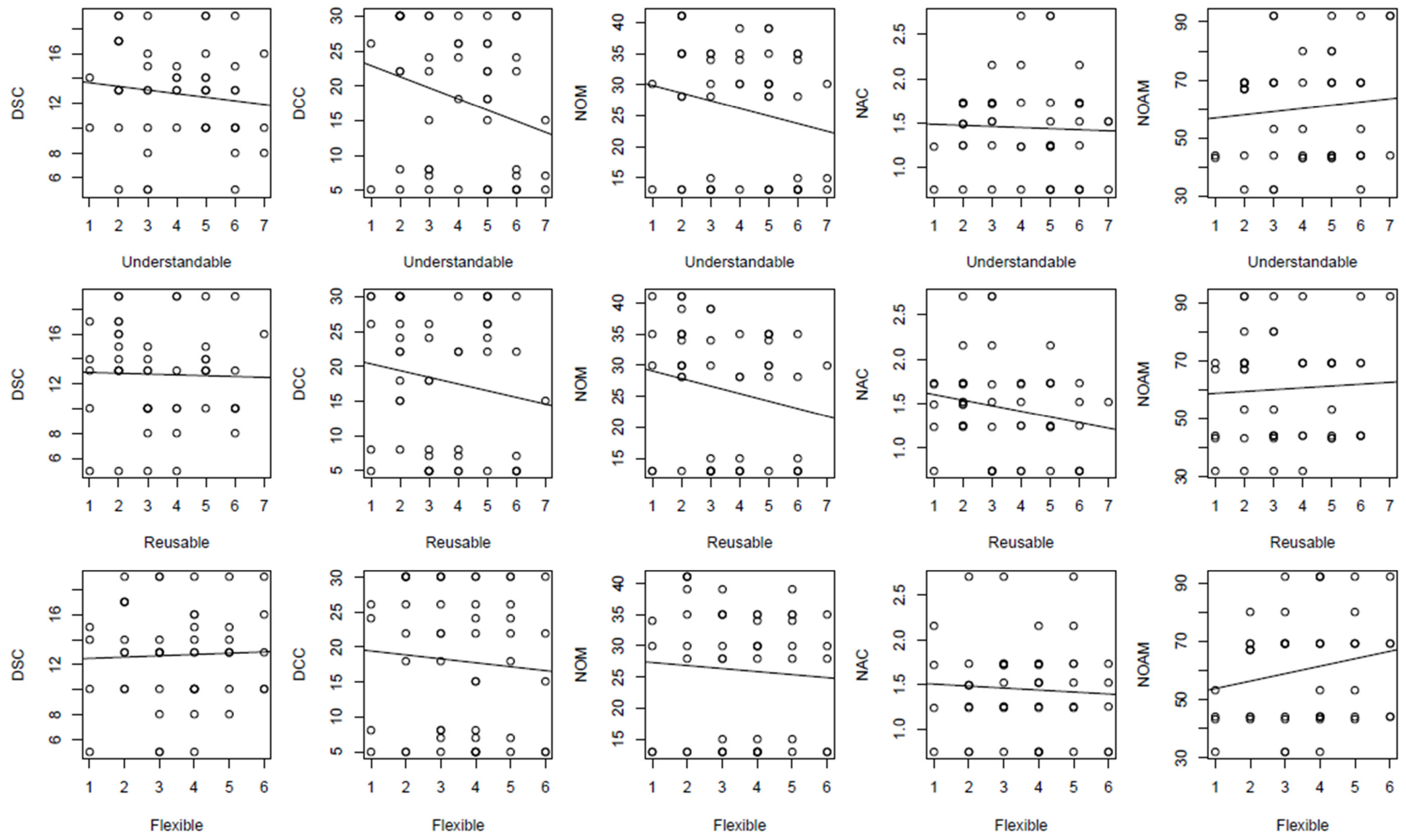
50 responses received



Histogram of number of year's experience of respondents

Respondents self-assessed expertise in software design and confidence in their ratings





Scatter Plots and Correlation between Human Judgment of Software Qualities and Software Metrics

Quality	DSC	DCC	NOM	NAC	NOAM
Understandable	-0.128	-0.271	-0.203	-0.0400	0.103
Reusable	-0.0280	-0.158	-0.195	-0.200	0.0572
Flexible	0.0386	-0.0806	-0.0677	-0.0613	0.202

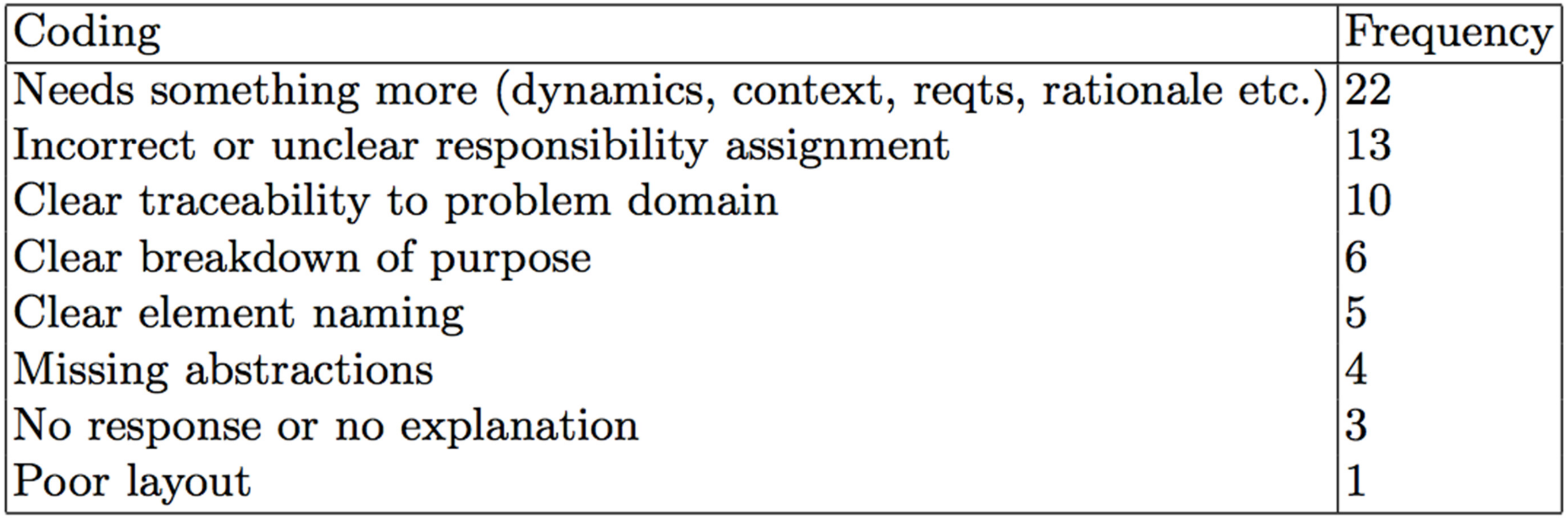
Spearman’s Rank Coefficients for the Correlation between Metrics and Human Judgement (to 3 s.f.)

Quality	DSC	DCC	NOM	NAC	NOAM
Understandable	0.375	0.0571	0.156	0.783	0.487
Reusable	0.847	0.272	0.174	0.163	0.693
Flexible	0.790	0.578	0.640	0.672	0.160

P-Values for a Two-Sided Test of the Spearman’s Rank Correlation Coefficients (to 3 s.f.)

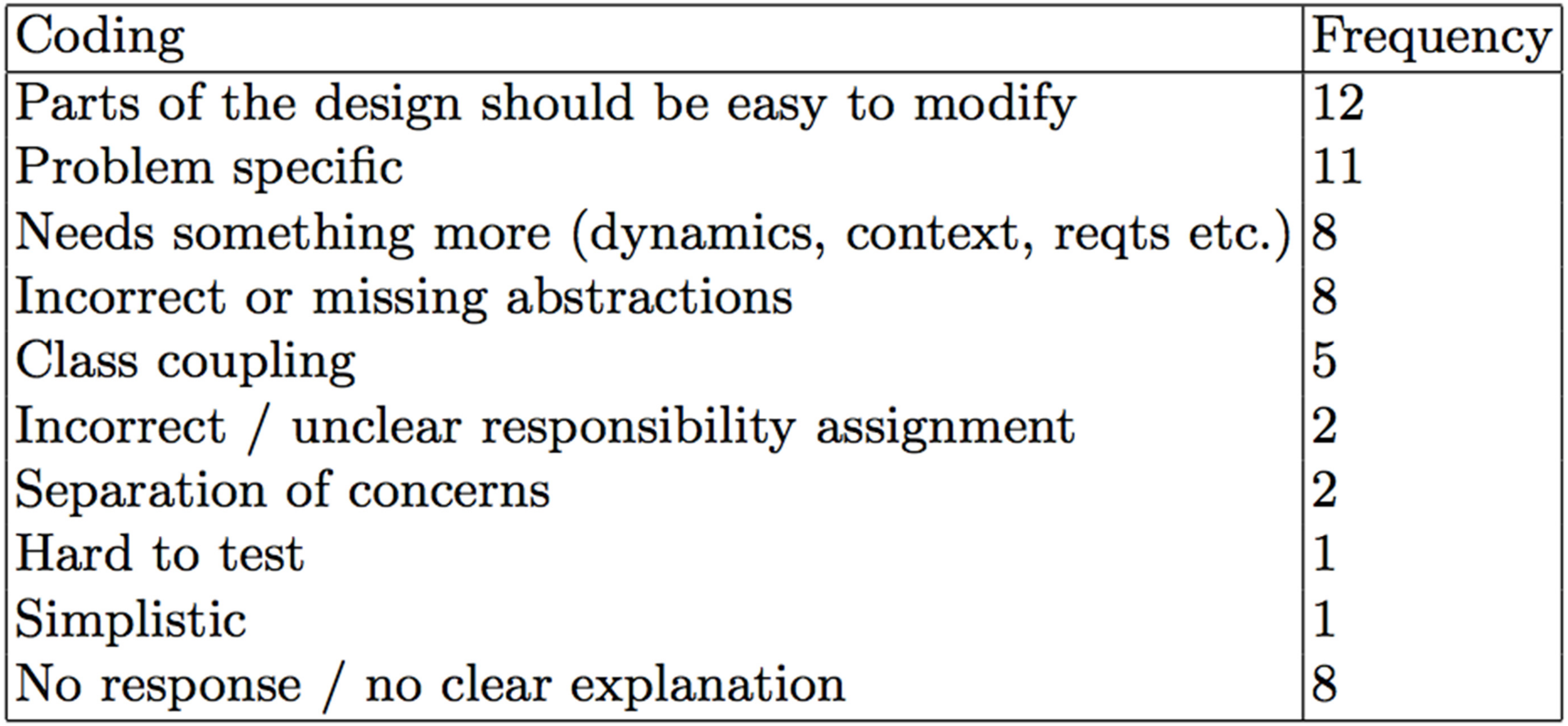
Qualitative Results

We asked the engineers to justify their judgments in prose.
We then coded their responses as per Grounded Theory.



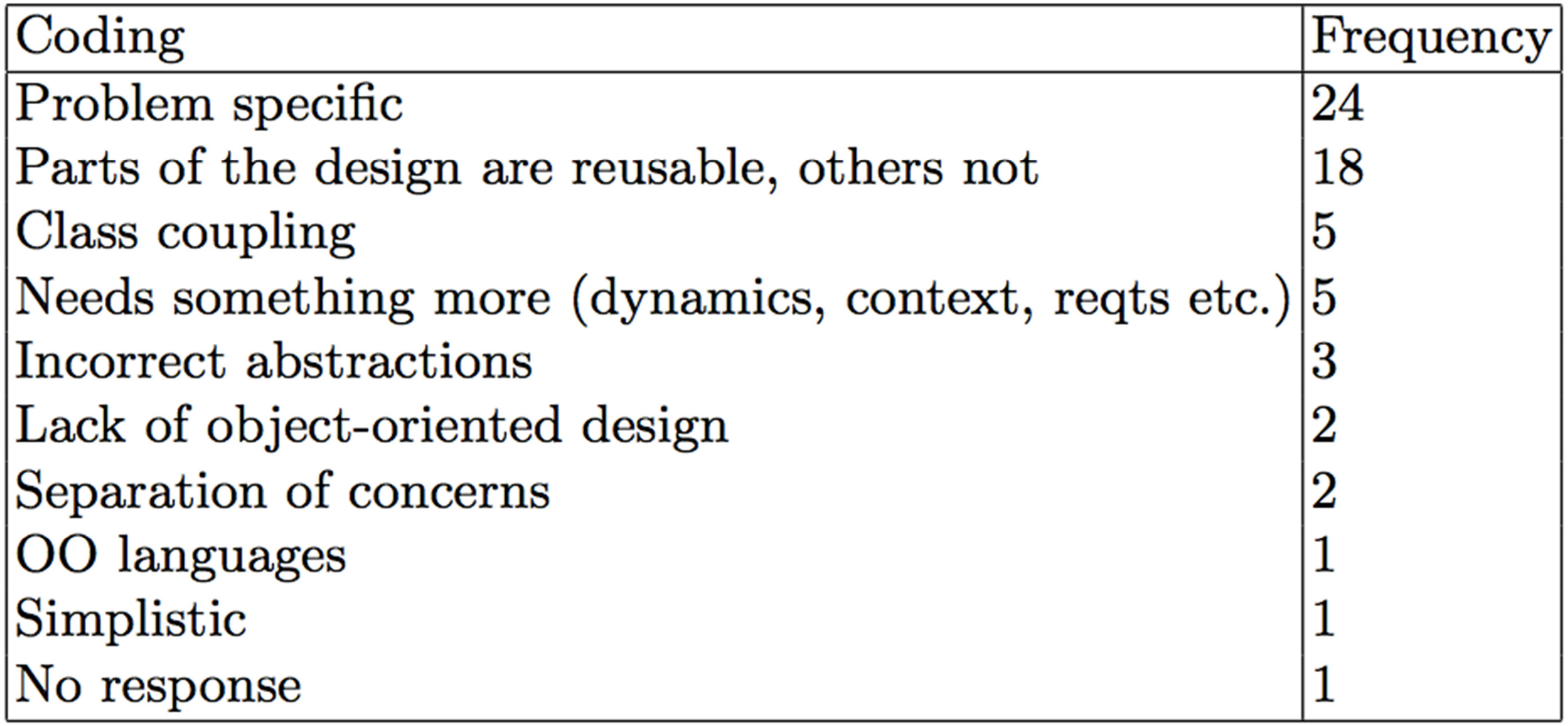
Coding	Frequency
Needs something more (dynamics, context, reqts, rationale etc.)	22
Incorrect or unclear responsibility assignment	13
Clear traceability to problem domain	10
Clear breakdown of purpose	6
Clear element naming	5
Missing abstractions	4
No response or no explanation	3
Poor layout	1

Classifications for Rationale behind Judgment of **"Understandable"**



Coding	Frequency
Parts of the design should be easy to modify	12
Problem specific	11
Needs something more (dynamics, context, reqts etc.)	8
Incorrect or missing abstractions	8
Class coupling	5
Incorrect / unclear responsibility assignment	2
Separation of concerns	2
Hard to test	1
Simplistic	1
No response / no clear explanation	8

Classifications for Rationale behind Judgment of **“Flexible”**



Coding	Frequency
Problem specific	24
Parts of the design are reusable, others not	18
Class coupling	5
Needs something more (dynamics, context, reqts etc.)	5
Incorrect abstractions	3
Lack of object-oriented design	2
Separation of concerns	2
OO languages	1
Simplistic	1
No response	1

Classifications for Rationale behind Judgment of **“Reuseable”**

We make the following observations:

A class diagram is ***not enough***

The Problem Domain ***matters***

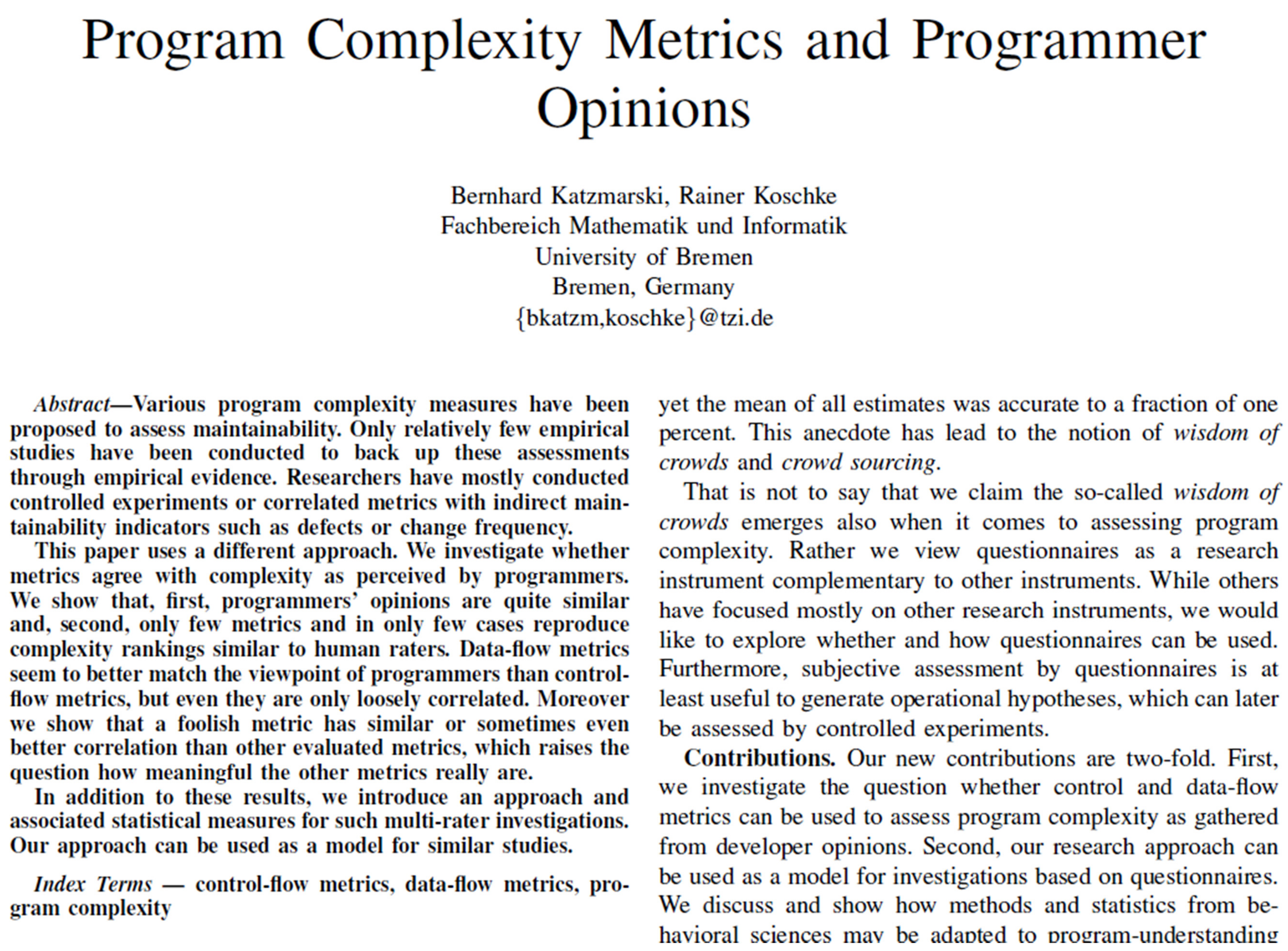
Qualities have meaning ***only in a given context***

Good design is ***intuitive***

Our standard metrics ***play a minor part***

Signpost

- Motivation
    - Role of structural metrics in software refactoring
- Hypotheses
- Experimental Methodology
    - Survey of industrial software development practitioners
- Results
    - Quantitative and qualitative analysis reported
- Related Work
- Conclusions



Program Complexity Metrics and Programmer Opinions

Bernhard Katzmarski, Rainer Koschke
Fachbereich Mathematik und Informatik
University of Bremen
Bremen, Germany
{bkatzm,koschke}@tzi.de

***Abstract*—Various program complexity measures have been proposed to assess maintainability. Only relatively few empirical studies have been conducted to back up these assessments through empirical evidence. Researchers have mostly conducted controlled experiments or correlated metrics with indirect maintainability indicators such as defects or change frequency.**

This paper uses a different approach. We investigate whether metrics agree with complexity as perceived by programmers. We show that, first, programmers' opinions are quite similar and, second, only few metrics and in only few cases reproduce complexity rankings similar to human raters. Data-flow metrics seem to better match the viewpoint of programmers than control-flow metrics, but even they are only loosely correlated. Moreover we show that a foolish metric has similar or sometimes even better correlation than other evaluated metrics, which raises the question how meaningful the other metrics really are.

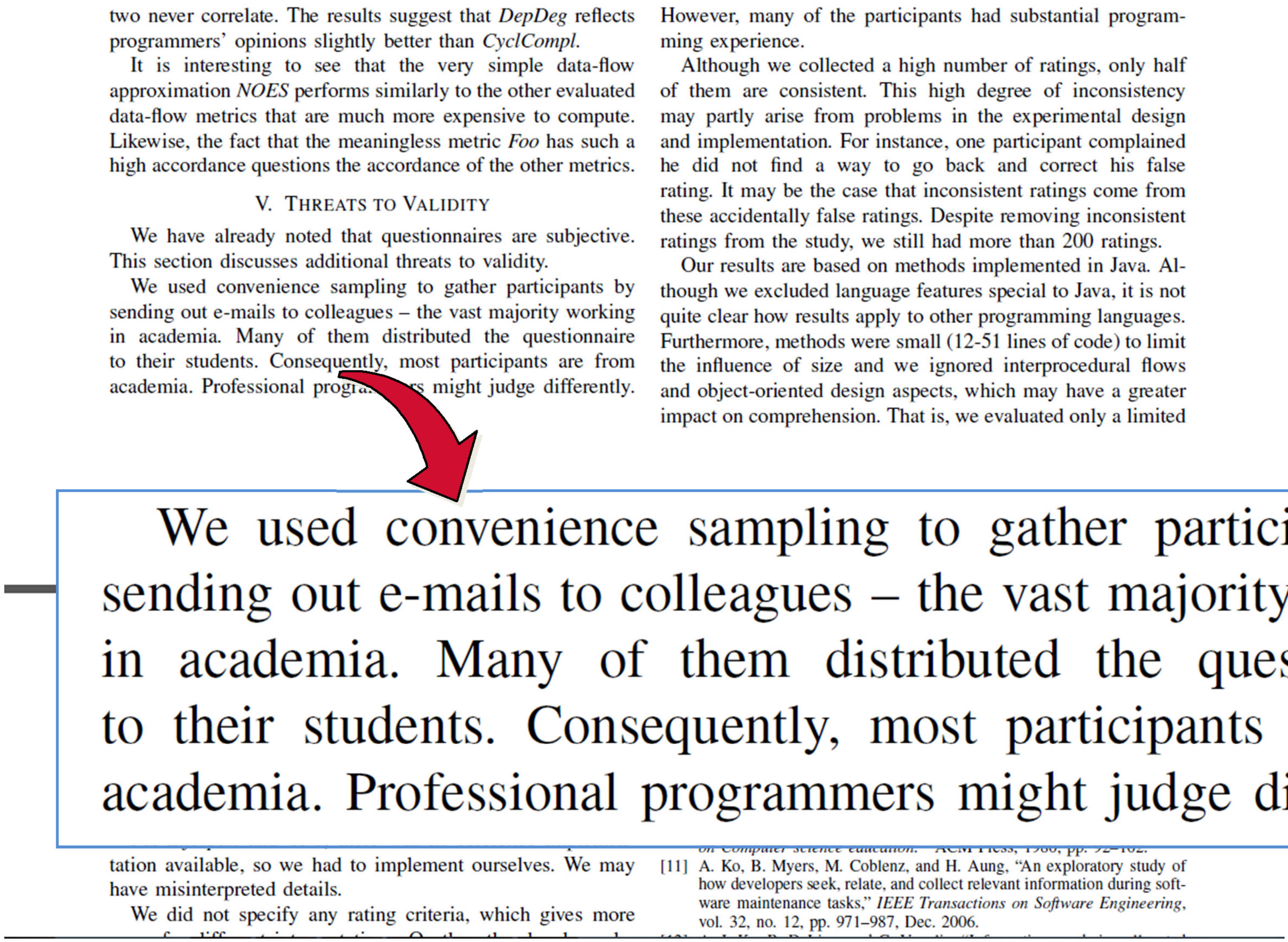
In addition to these results, we introduce an approach and associated statistical measures for such multi-rater investigations. Our approach can be used as a model for similar studies.

***Index Terms* — control-flow metrics, data-flow metrics, program complexity**

yet the mean of all estimates was accurate to a fraction of one percent. This anecdote has lead to the notion of *wisdom of crowds* and *crowd sourcing*.

That is not to say that we claim the so-called *wisdom of crowds* emerges also when it comes to assessing program complexity. Rather we view questionnaires as a research instrument complementary to other instruments. While others have focused mostly on other research instruments, we would like to explore whether and how questionnaires can be used. Furthermore, subjective assessment by questionnaires is at least useful to generate operational hypotheses, which can later be assessed by controlled experiments.

Contributions. Our new contributions are two-fold. First, we investigate the question whether control and data-flow metrics can be used to assess program complexity as gathered from developer opinions. Second, our research approach can be used as a model for investigations based on questionnaires. We discuss and show how methods and statistics from behavioral sciences may be adapted to program-understanding



two never correlate. The results suggest that *DepDeg* reflects programmers' opinions slightly better than *CyclCompl*.

It is interesting to see that the very simple data-flow approximation *NOES* performs similarly to the other evaluated data-flow metrics that are much more expensive to compute. Likewise, the fact that the meaningless metric *Foo* has such a high accordance questions the accordance of the other metrics.

V. Threats to Validity

We have already noted that questionnaires are subjective. This section discusses additional threats to validity.

We used convenience sampling to gather participants by sending out e-mails to colleagues – the vast majority working in academia. Many of them distributed the questionnaire to their students. Consequently, most participants are from academia. Professional progra... might judge differently.

tation available, so we had to implement ourselves. We may have misinterpreted details.

We did not specify any rating criteria, which gives more

However, many of the participants had substantial programming experience.

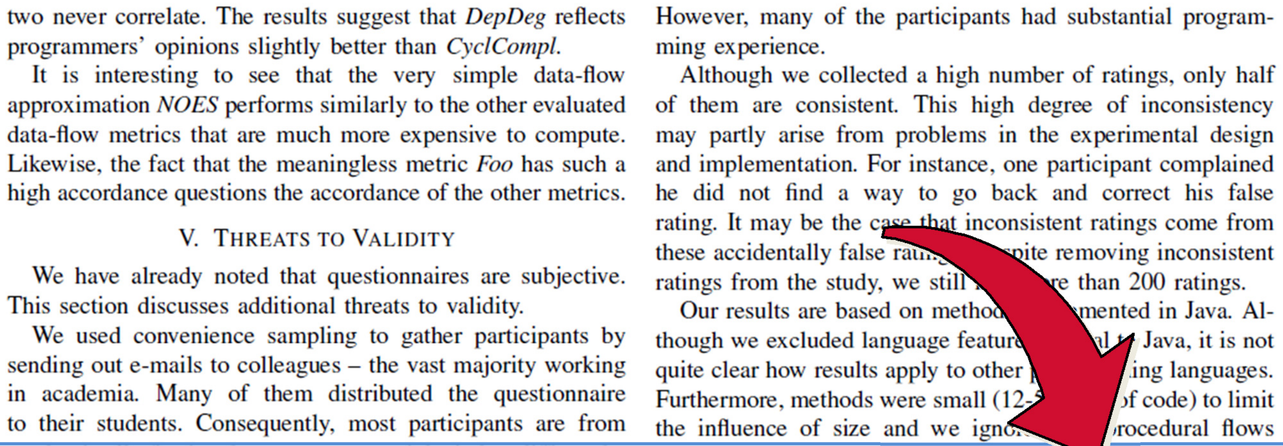
Although we collected a high number of ratings, only half of them are consistent. This high degree of inconsistency may partly arise from problems in the experimental design and implementation. For instance, one participant complained he did not find a way to go back and correct his false rating. It may be the case that inconsistent ratings come from these accidentally false ratings. Despite removing inconsistent ratings from the study, we still had more than 200 ratings.

Our results are based on methods implemented in Java. Although we excluded language features special to Java, it is not quite clear how results apply to other programming languages. Furthermore, methods were small (12-51 lines of code) to limit the influence of size and we ignored interprocedural flows and object-oriented design aspects, which may have a greater impact on comprehension. That is, we evaluated only a limited

on Computer science education." ACM Press, 1986, pp. 92–102.

[11] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

We used convenience sampling to gather participants by sending out e-mails to colleagues – the vast majority working in academia. Many of them distributed the questionnaire to their students. Consequently, most participants are from academia. Professional programmers might judge differently.



two never correlate. The results suggest that *DepDeg* reflects programmers' opinions slightly better than *CyclCompl*.

It is interesting to see that the very simple data-flow approximation *NOES* performs similarly to the other evaluated data-flow metrics that are much more expensive to compute. Likewise, the fact that the meaningless metric *Foo* has such a high accordance questions the accordance of the other metrics.

V. Threats to Validity

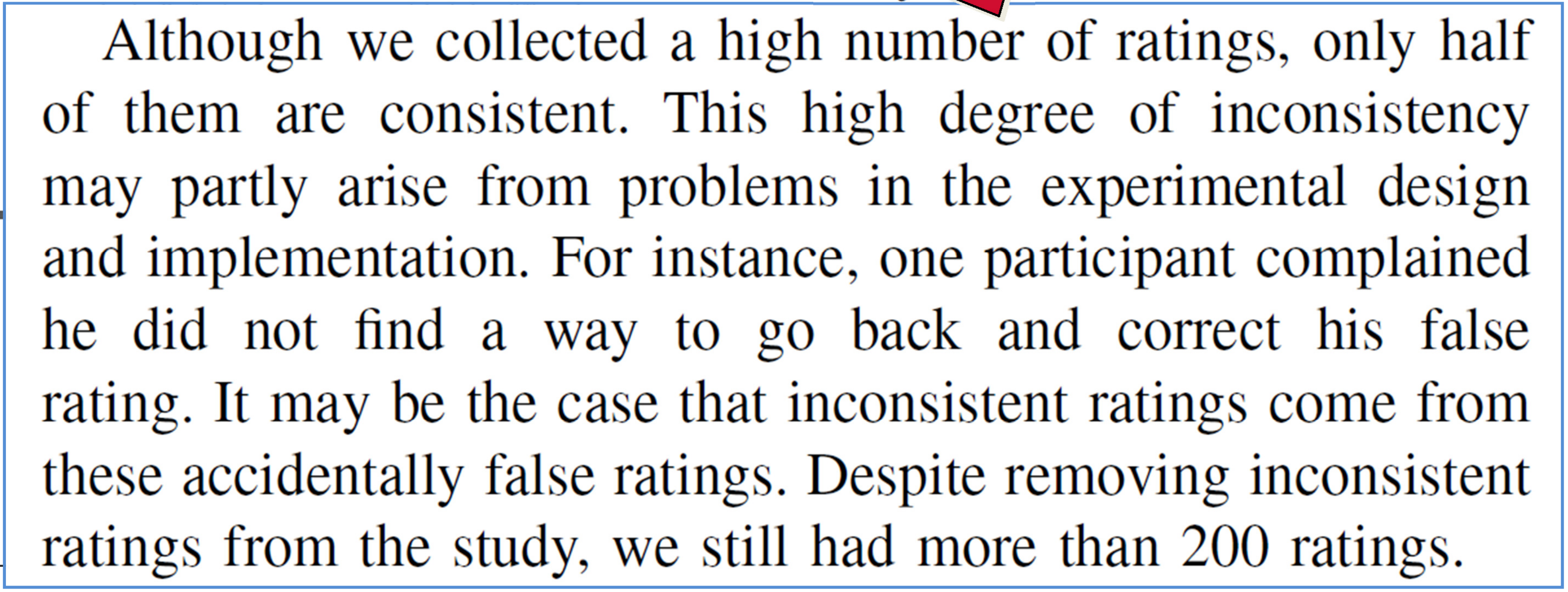
We have already noted that questionnaires are subjective. This section discusses additional threats to validity.

We used convenience sampling to gather participants by sending out e-mails to colleagues – the vast majority working in academia. Many of them distributed the questionnaire to their students. Consequently, most participants are from

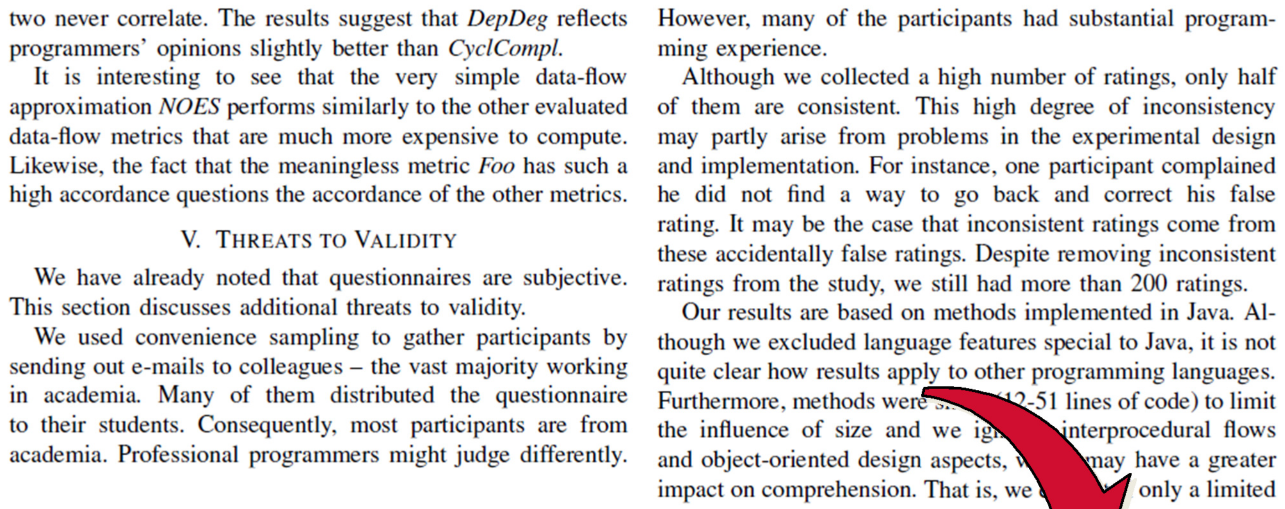
However, many of the participants had substantial programming experience.

Although we collected a high number of ratings, only half of them are consistent. This high degree of inconsistency may partly arise from problems in the experimental design and implementation. For instance, one participant complained he did not find a way to go back and correct his false rating. It may be the case that inconsistent ratings come from these accidentally false rati spite removing inconsistent ratings from the study, we still re than 200 ratings.

Our results are based on metho mented in Java. Although we excluded language feature al to Java, it is not quite clear how results apply to other ing languages. Furthermore, methods were small (12- of code) to limit the influence of size and we igno rocedural flows



Although we collected a high number of ratings, only half of them are consistent. This high degree of inconsistency may partly arise from problems in the experimental design and implementation. For instance, one participant complained he did not find a way to go back and correct his false rating. It may be the case that inconsistent ratings come from these accidentally false ratings. Despite removing inconsistent ratings from the study, we still had more than 200 ratings.



two never correlate. The results suggest that *DepDeg* reflects programmers' opinions slightly better than *CyclCompl*.

It is interesting to see that the very simple data-flow approximation *NOES* performs similarly to the other evaluated data-flow metrics that are much more expensive to compute. Likewise, the fact that the meaningless metric *Foo* has such a high accordance questions the accordance of the other metrics.

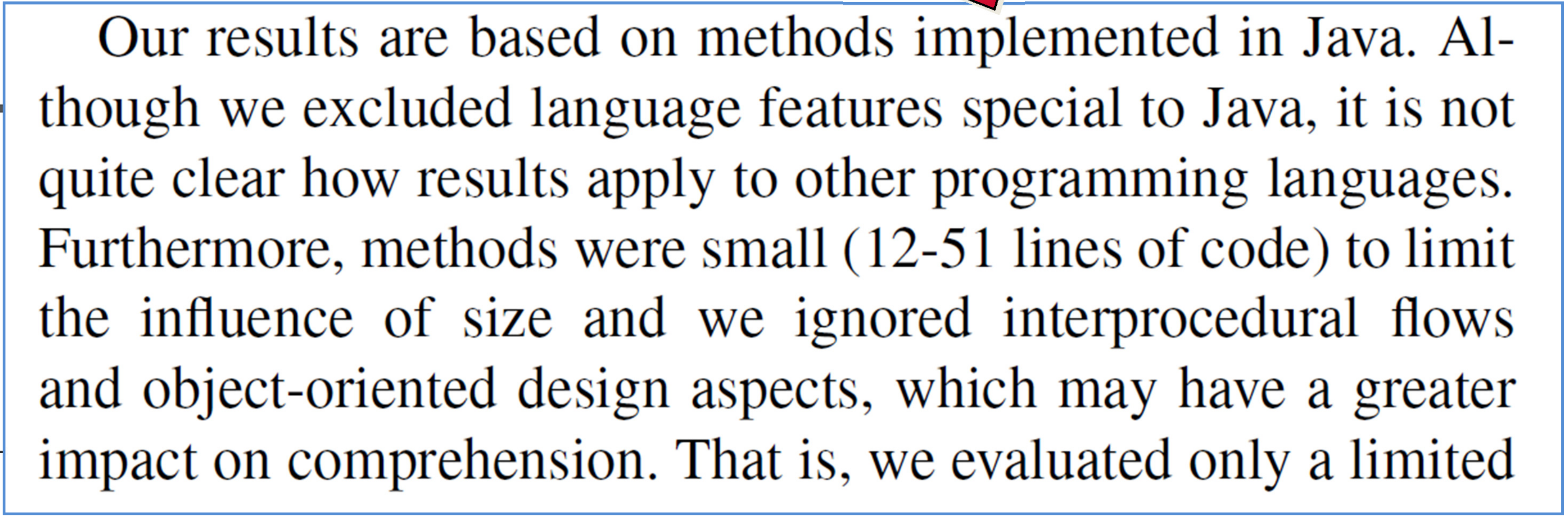
V. Threats to Validity

We have already noted that questionnaires are subjective. This section discusses additional threats to validity.

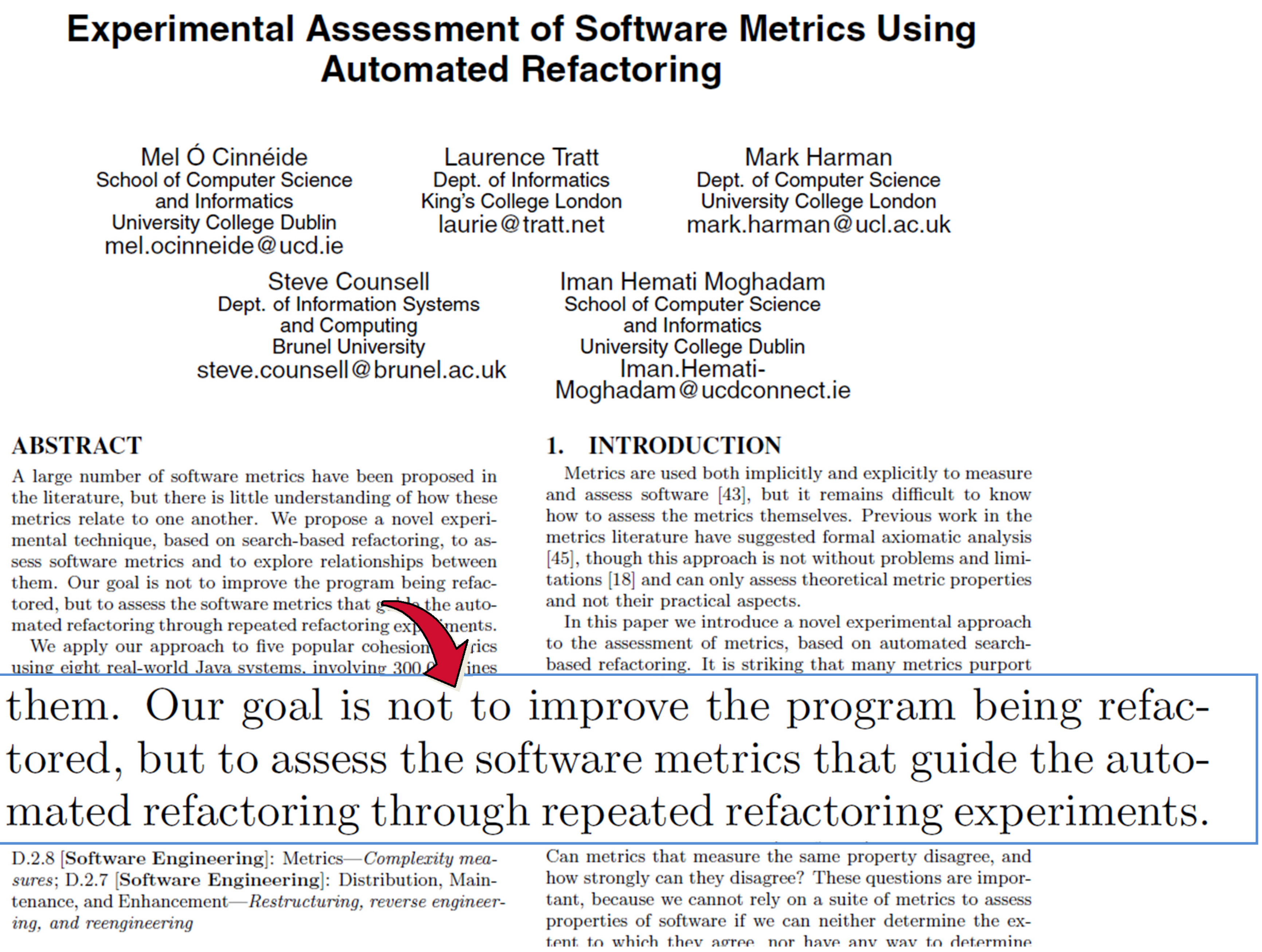
We used convenience sampling to gather participants by sending out e-mails to colleagues – the vast majority working in academia. Many of them distributed the questionnaire to their students. Consequently, most participants are from academia. Professional programmers might judge differently. However, many of the participants had substantial programming experience.

Although we collected a high number of ratings, only half of them are consistent. This high degree of inconsistency may partly arise from problems in the experimental design and implementation. For instance, one participant complained he did not find a way to go back and correct his false rating. It may be the case that inconsistent ratings come from these accidentally false ratings. Despite removing inconsistent ratings from the study, we still had more than 200 ratings.

Our results are based on methods implemented in Java. Although we excluded language features special to Java, it is not quite clear how results apply to other programming languages. Furthermore, methods were s... (12-51 lines of code) to limit the influence of size and we ig... interprocedural flows and object-oriented design aspects, w... may have a greater impact on comprehension. That is, we ... only a limited



Our results are based on methods implemented in Java. Although we excluded language features special to Java, it is not quite clear how results apply to other programming languages. Furthermore, methods were small (12-51 lines of code) to limit the influence of size and we ignored interprocedural flows and object-oriented design aspects, which may have a greater impact on comprehension. That is, we evaluated only a limited



Experimental Assessment of Software Metrics Using Automated Refactoring

Mel Ó Cinnéide
School of Computer Science and Informatics
University College Dublin
mel.ocinneide@ucd.ie

Laurence Tratt
Dept. of Informatics
King's College London
laurie@tratt.net

Mark Harman
Dept. of Computer Science
University College London
mark.harman@ucl.ac.uk

Steve Counsell
Dept. of Information Systems and Computing
Brunel University
steve.counsell@brunel.ac.uk

Iman Hemati Moghadam
School of Computer Science and Informatics
University College Dublin
Iman.Hemati-Moghadam@ucdconnect.ie

ABSTRACT

A large number of software metrics have been proposed in the literature, but there is little understanding of how these metrics relate to one another. We propose a novel experimental technique, based on search-based refactoring, to assess software metrics and to explore relationships between them. Our goal is not to improve the program being refactored, but to assess the software metrics that guide the automated refactoring through repeated refactoring experiments.

We apply our approach to five popular cohesion metrics using eight real-world Java systems, involving 300 [illegible] ines

them. Our goal is not to improve the program being refactored, but to assess the software metrics that guide the automated refactoring through repeated refactoring experiments.

D.2.8 [**Software Engineering**]: Metrics—*Complexity measures*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

1. INTRODUCTION

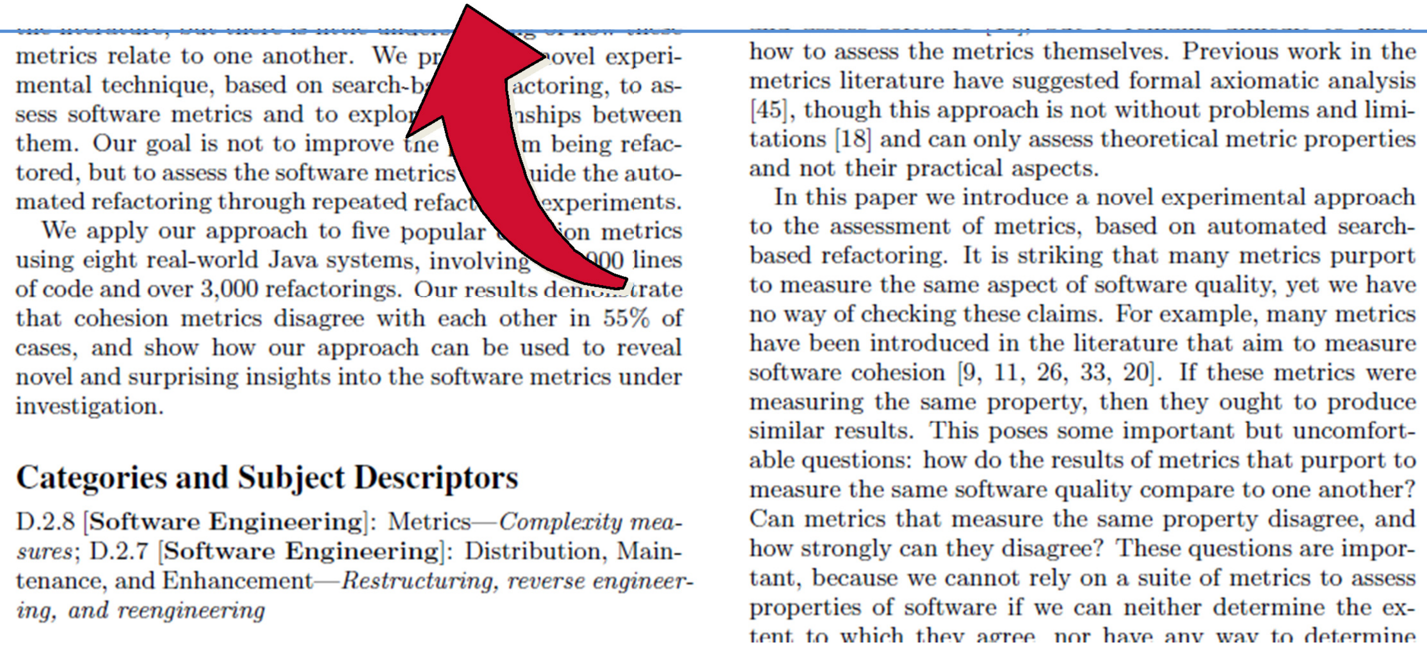
Metrics are used both implicitly and explicitly to measure and assess software [43], but it remains difficult to know how to assess the metrics themselves. Previous work in the metrics literature have suggested formal axiomatic analysis [45], though this approach is not without problems and limitations [18] and can only assess theoretical metric properties and not their practical aspects.

In this paper we introduce a novel experimental approach to the assessment of metrics, based on automated search-based refactoring. It is striking that many metrics purport

Can metrics that measure the same property disagree, and how strongly can they disagree? These questions are important, because we cannot rely on a suite of metrics to assess properties of software if we can neither determine the extent to which they agree, nor have any way to determine

Experimental Assessment of Software Metrics Using Automated Refactoring

We apply our approach to five popular cohesion metrics using eight real-world Java systems, involving 300,000 lines of code and over 3,000 refactorings. Our results demonstrate that cohesion metrics disagree with each other in 55% of cases, and show how our approach can be used to reveal novel and surprising insights into the software metrics under investigation.



metrics relate to one another. We pr[illegible]ovel experimental technique, based on search-b[illegible]actoring, to assess software metrics and to explor[illegible]nships between them. Our goal is not to improve the [illegible]m being refactored, but to assess the software metrics [illegible]uide the automated refactoring through repeated refact[illegible]experiments.

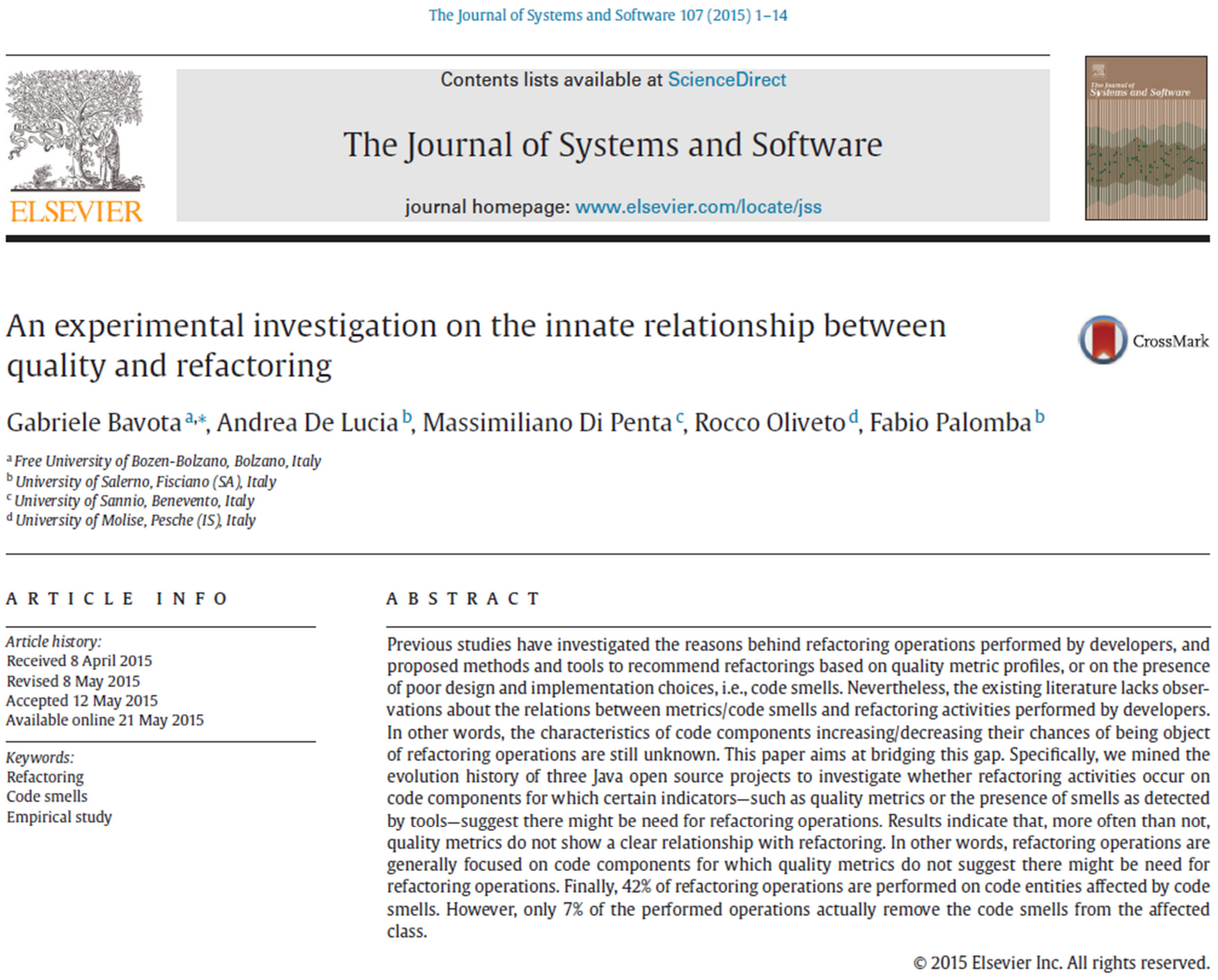
We apply our approach to five popular c[illegible]ion metrics using eight real-world Java systems, involving [illegible]000 lines of code and over 3,000 refactorings. Our results dem[illegible]trate that cohesion metrics disagree with each other in 55% of cases, and show how our approach can be used to reveal novel and surprising insights into the software metrics under investigation.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Complexity measures*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

how to assess the metrics themselves. Previous work in the metrics literature have suggested formal axiomatic analysis [45], though this approach is not without problems and limitations [18] and can only assess theoretical metric properties and not their practical aspects.

In this paper we introduce a novel experimental approach to the assessment of metrics, based on automated search-based refactoring. It is striking that many metrics purport to measure the same aspect of software quality, yet we have no way of checking these claims. For example, many metrics have been introduced in the literature that aim to measure software cohesion [9, 11, 26, 33, 20]. If these metrics were measuring the same property, then they ought to produce similar results. This poses some important but uncomfortable questions: how do the results of metrics that purport to measure the same software quality compare to one another? Can metrics that measure the same property disagree, and how strongly can they disagree? These questions are important, because we cannot rely on a suite of metrics to assess properties of software if we can neither determine the extent to which they agree, nor have any way to determine



The Journal of Systems and Software 107 (2015) 1–14

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

An experimental investigation on the innate relationship between quality and refactoring

Gabriele Bavota[a,*], Andrea De Lucia[b], Massimiliano Di Penta[c], Rocco Oliveto[d], Fabio Palomba[b]

[a] Free University of Bozen-Bolzano, Bolzano, Italy
[b] University of Salerno, Fisciano (SA), Italy
[c] University of Sannio, Benevento, Italy
[d] University of Molise, Pesche (IS), Italy

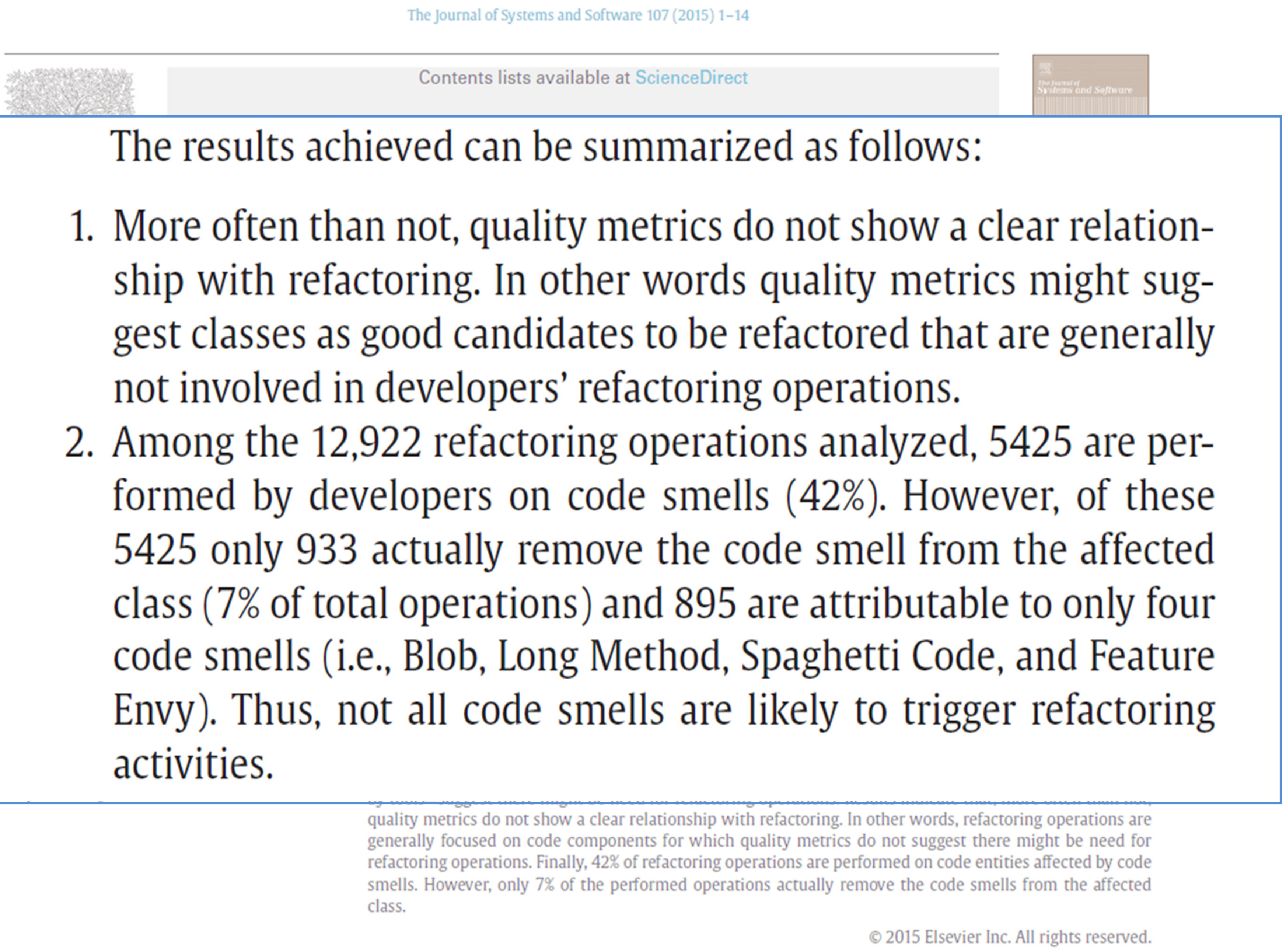
A R T I C L E   I N F O

Article history:
Received 8 April 2015
Revised 8 May 2015
Accepted 12 May 2015
Available online 21 May 2015

Keywords:
Refactoring
Code smells
Empirical study

A B S T R A C T

Previous studies have investigated the reasons behind refactoring operations performed by developers, and proposed methods and tools to recommend refactorings based on quality metric profiles, or on the presence of poor design and implementation choices, i.e., code smells. Nevertheless, the existing literature lacks observations about the relations between metrics/code smells and refactoring activities performed by developers. In other words, the characteristics of code components increasing/decreasing their chances of being object of refactoring operations are still unknown. This paper aims at bridging this gap. Specifically, we mined the evolution history of three Java open source projects to investigate whether refactoring activities occur on code components for which certain indicators—such as quality metrics or the presence of smells as detected by tools—suggest there might be need for refactoring operations. Results indicate that, more often than not, quality metrics do not show a clear relationship with refactoring. In other words, refactoring operations are generally focused on code components for which quality metrics do not suggest there might be need for refactoring operations. Finally, 42% of refactoring operations are performed on code entities affected by code smells. However, only 7% of the performed operations actually remove the code smells from the affected class.



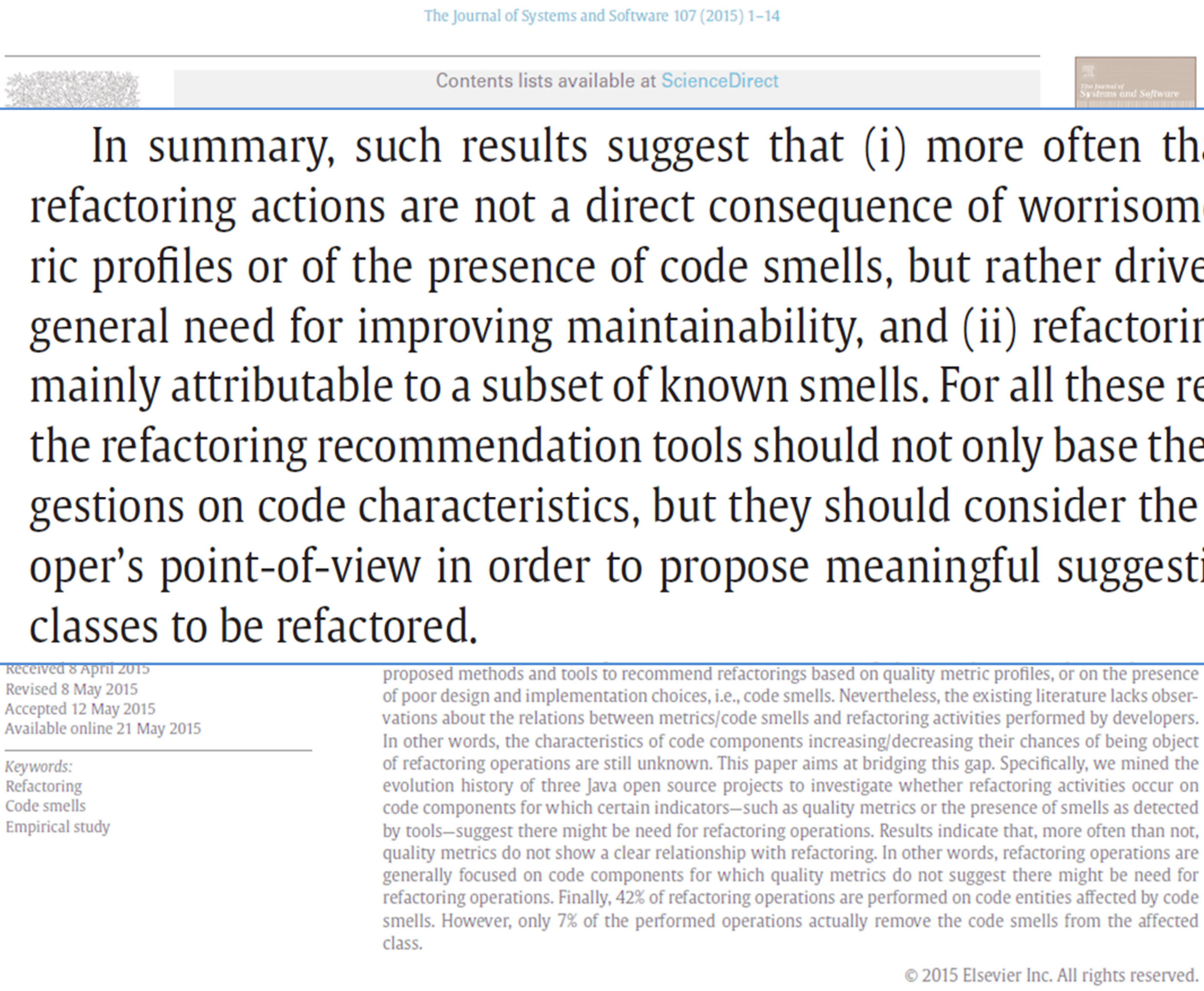
The Journal of Systems and Software 107 (2015) 1–14

Contents lists available at ScienceDirect

The results achieved can be summarized as follows:

1. More often than not, quality metrics do not show a clear relationship with refactoring. In other words quality metrics might suggest classes as good candidates to be refactored that are generally not involved in developers' refactoring operations.
2. Among the 12,922 refactoring operations analyzed, 5425 are performed by developers on code smells (42%). However, of these 5425 only 933 actually remove the code smell from the affected class (7% of total operations) and 895 are attributable to only four code smells (i.e., Blob, Long Method, Spaghetti Code, and Feature Envy). Thus, not all code smells are likely to trigger refactoring activities.

quality metrics do not show a clear relationship with refactoring. In other words, refactoring operations are generally focused on code components for which quality metrics do not suggest there might be need for refactoring operations. Finally, 42% of refactoring operations are performed on code entities affected by code smells. However, only 7% of the performed operations actually remove the code smells from the affected class.



The Journal of Systems and Software 107 (2015) 1–14

Contents lists available at ScienceDirect

In summary, such results suggest that (i) more often than not refactoring actions are not a direct consequence of worrisome metric profiles or of the presence of code smells, but rather driven by a general need for improving maintainability, and (ii) refactorings are mainly attributable to a subset of known smells. For all these reasons, the refactoring recommendation tools should not only base their suggestions on code characteristics, but they should consider the developer's point-of-view in order to propose meaningful suggestions of classes to be refactored.

Received 8 April 2015
Revised 8 May 2015
Accepted 12 May 2015
Available online 21 May 2015

Keywords:
Refactoring
Code smells
Empirical study

proposed methods and tools to recommend refactorings based on quality metric profiles, or on the presence of poor design and implementation choices, i.e., code smells. Nevertheless, the existing literature lacks observations about the relations between metrics/code smells and refactoring activities performed by developers. In other words, the characteristics of code components increasing/decreasing their chances of being object of refactoring operations are still unknown. This paper aims at bridging this gap. Specifically, we mined the evolution history of three Java open source projects to investigate whether refactoring activities occur on code components for which certain indicators—such as quality metrics or the presence of smells as detected by tools—suggest there might be need for refactoring operations. Results indicate that, more often than not, quality metrics do not show a clear relationship with refactoring. In other words, refactoring operations are generally focused on code components for which quality metrics do not suggest there might be need for refactoring operations. Finally, 42% of refactoring operations are performed on code entities affected by code smells. However, only 7% of the performed operations actually remove the code smells from the affected class.

Our Threats to Validity

Class models are small:

constrained by cognitive overload and screen space

Participant understanding of qualities:

we provided definitions and navigation to revisit

Bias of target population and sample frame:

(as with any survey)

targeted professional institutions and practitioners

Pilot survey could have been more extensive

Signpost

- Motivation
  - Role of structural metrics in software refactoring
- Hypotheses
- Experimental Methodology
  - Survey of industrial software development practitioners
- Results
  - Quantitative and qualitative analysis reported
- Related Work
- Conclusions

Conclusions (1)

Refactoring metrics are not correlated with human engineer judgement

Unable to refute the null hypothesis; thus unable to support conjecture that SBSE refactoring tools relying solely on these metrics will consistently propose useful refactored models to engineers.

Conclusions (2) – wider lessons

Simple metrics are not able to entirely capture essential qualities of software design used by human engineers

Software is inextricably connected to a problem domain

We note recent advances in machine learning
and automatic programming to address such concerns…

…but without their inclusion,
human-in-the-loop automated refactoring systems
may be required for meaningful solutions.

Software Refactoring: Metrics are Not Enough

questions?

@chrislsimons
chris.simons@uwe.ac.uk

@jsinger_compsci
jeremy.singer@glasgow.ac.uk

@davidwhitecs
david.r.white@glasgow.ac.uk

ssbse 2015