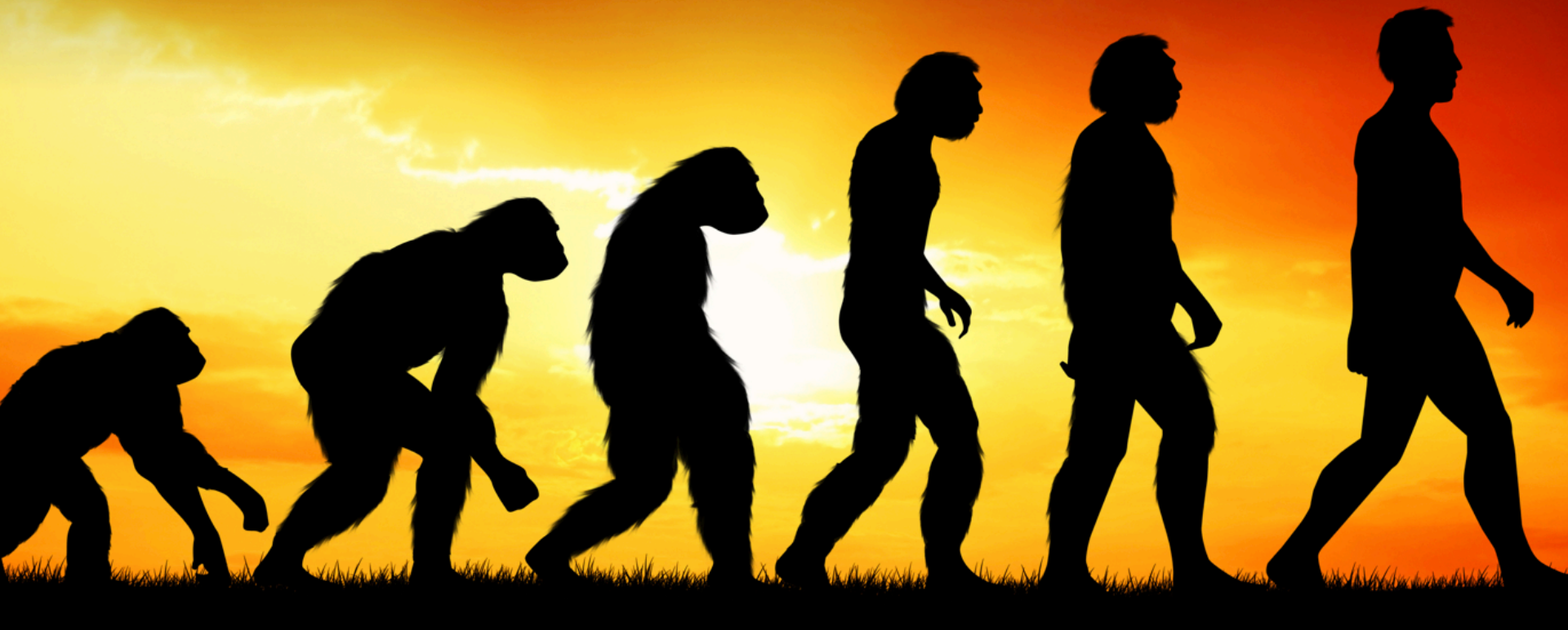
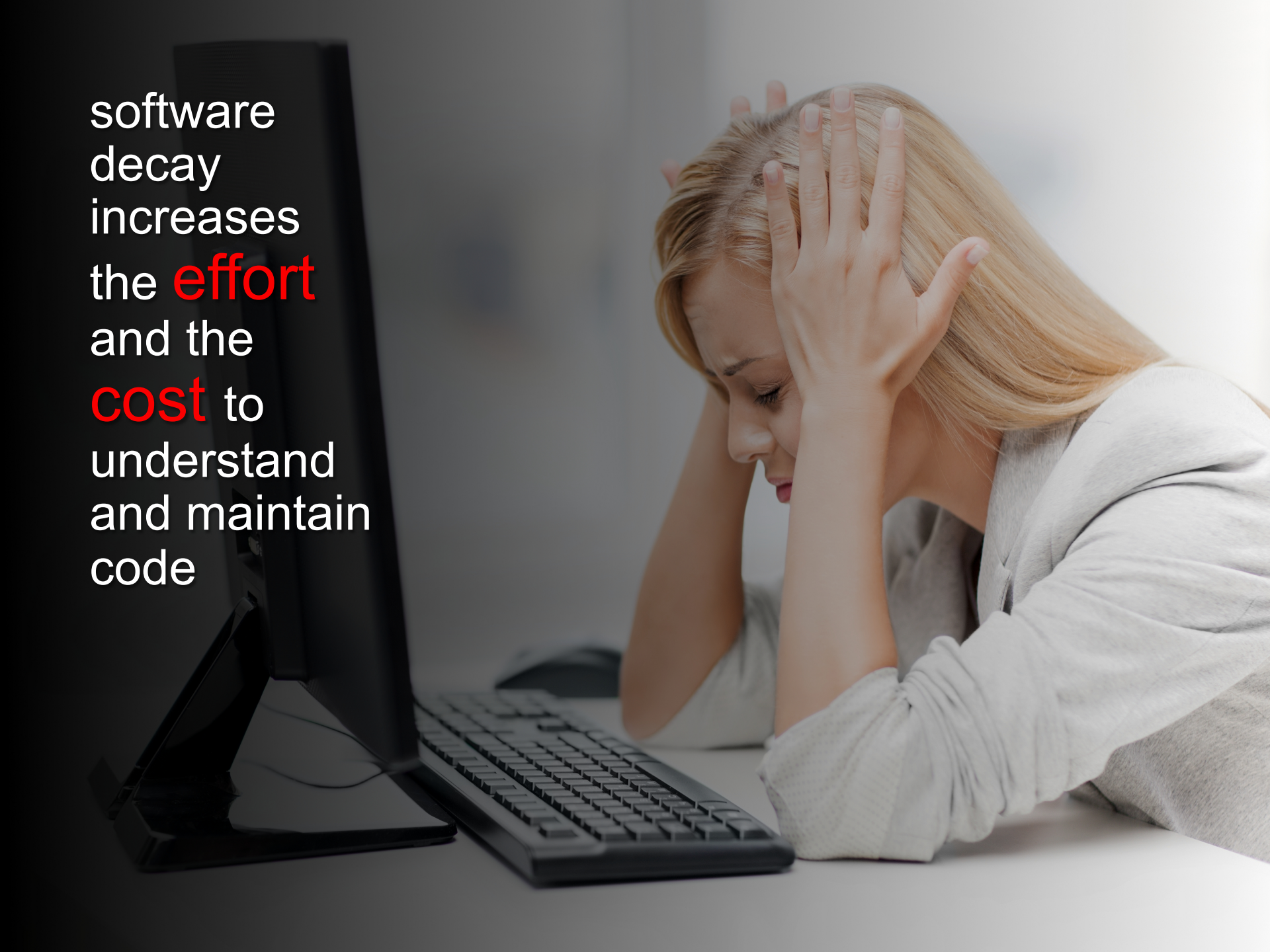
Are Unreachable Methods Harmful? Results from a Controlled Experiment

Simone Romano, Christopher Vendome, *Giuseppe Scanniello*, and Denys Poshyvanyk

software evolution

changes cause a decay of software design



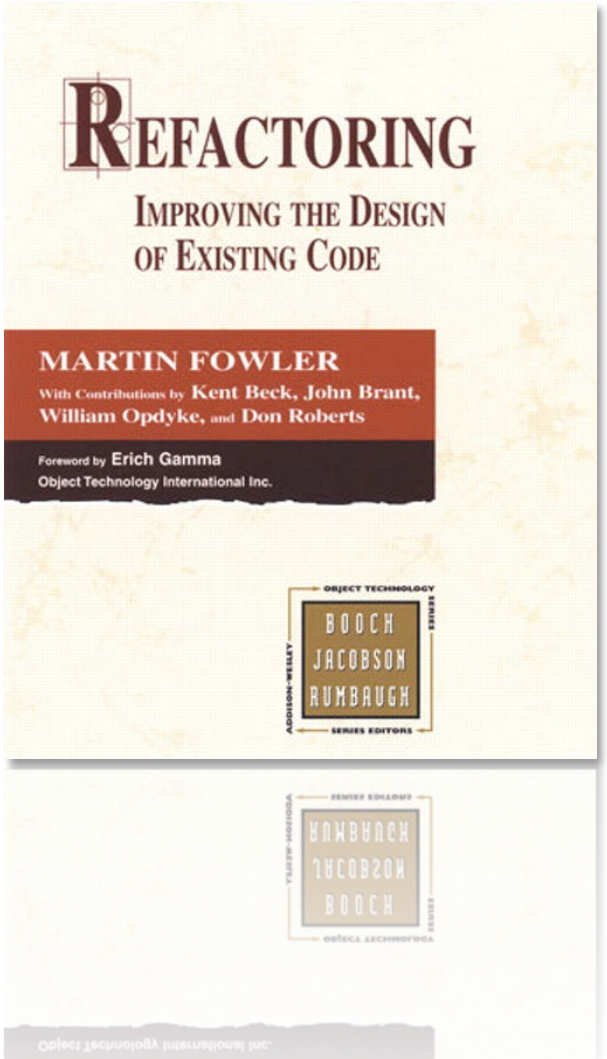

software
decay
increases
the effort
and the
cost to
understand
and maintain
code

what is one of the therapy for software decay?

a lightweight therapy

"The process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure"

--M. Fowler



what are the symptoms?

bad smells in code

"Symptoms of poor design or implementation choices"

--M. Fowler

Shotgun Surgery

azy Class

Primitive Obsession

Middle Man

Dead Code

Feature Envy

Long Parameter List

Data Class

many kinds of bad smells in code

Long Method

Parallel Inheritance Hierarchie

Large Class

sage Chains

Duplicate Code

Switch Statements

Inappropriate Intima

programming language field

"Dead code refers to computations whose results are never used"

--S. K. Debray et al.

Compiler Techniques for Code Compaction

SAUMYA K. DEBRAY and WILLIAM EVANS
The University of Arizona
ROBERT MUTH
Compaq Computer Corp.
and
BJORN DE SUTTER
University of Ghent

In recent years there has been an increasing trend toward the incorporation of computers into a variety of devices where the amount of memory available is limited. This makes it desirable to try to reduce the size of applications where possible. This article explores the use of compiler techniques to accomplish code compaction to yield smaller executables. The main contribution of this article is to show that careful, aggressive, interprocedural optimization, together with procedural abstraction of repeated code fragments, can yield significantly better reductions in code size than previous approaches, which have generally focused on abstraction of repeated instruction sequences. We also show how "equivalent" code fragments can be detected and factored out using conventional compiler techniques, and without having to resort to purely linear treatments of code sequences as in suffix-tree-based approaches, thereby setting up a framework for code compaction that can be more flexible in its treatment of what code fragments are considered equivalent. Our ideas have been implemented in the form of a binary-rewriting tool that reduces the size of executables by about 30% on the average.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation; compilers; optimization*; E.4 [**Coding and Information Theory**]: Data Compaction and Compression—*program representation*

General Terms: Experimentation, Performance

Additional Key Words and Phrases: Code compaction, code compression, code size reduction

1. INTRODUCTION

In recent years there has been an increasing trend towards the incorporation of computers into a wide variety of devices, such as palm-tops, telephones, embedded

The work of Saumya Debray and Robert Muth was supported in part by the National Science Foundation under grants CCR-9711166, CDA-9500991, and ASC-9720738. The work of Bjorn De Sutter was supported in part by the Fund for Scientific Research—Flanders under grant 3G001998.
Authors' addresses: S. Debray and W. Evans, Department of Computer Science, University of Arizona, Tucson, AZ 85721; email: {debray, will}@cs.arizona.edu; R. Muth, Alpha Development Group, Compaq Computer Corporation, Shrewsbury, MA 01749; email: Robert.Muth@compaq.com; B. De Sutter, Department of Electronics and Information Systems, University of Ghent, B-9000 Gent, Belgium; email: brdsutte@elis.rug.ac.be.

ACM Transactions on Programming Languages and Systems, Vol. 22, No. 2, March 2000, Pages 378–415.

programming language field

"Code that is **unreachable** can never be **executed**"

--S. K. Debray et al.

"dead" **different** from "unreachable"

Compiler Techniques for Code Compaction

SAUMYA K. DEBRAY and WILLIAM EVANS
The University of Arizona
ROBERT MUTH
Compaq Computer Corp.
and
BJORN DE SUTTER
University of Ghent

In recent years there has been an increasing trend toward the incorporation of computers into a variety of devices where the amount of memory available is limited. This makes it desirable to try to reduce the size of applications where possible. This article explores the use of compiler techniques to accomplish code compaction to yield smaller executables. The main contribution of this article is to show that careful, aggressive, interprocedural optimization, together with procedural abstraction of repeated code fragments, can yield significantly better reductions in code size than previous approaches, which have generally focused on abstraction of repeated instruction sequences. We also show how "equivalent" code fragments can be detected and factored out using conventional compiler techniques, and without having to resort to purely linear treatments of code sequences as in suffix-tree-based approaches, thereby setting up a framework for code compaction that can be more flexible in its treatment of what code fragments are considered equivalent. Our ideas have been implemented in the form of a binary-rewriting tool that reduces the size of executables by about 30% on the average.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation; compilers; optimization*; E.4 [**Coding and Information Theory**]: Data Compaction and Compression—*program representation*

General Terms: Experimentation, Performance

Additional Key Words and Phrases: Code compaction, code compression, code size reduction

1. INTRODUCTION

In recent years there has been an increasing trend towards the incorporation of computers into a wide variety of devices, such as palm-tops, telephones, embedded

The work of Saumya Debray and Robert Muth was supported in part by the National Science Foundation under grants CCR-9711166, CDA-9500991, and ASC-9720738. The work of Bjorn De Sutter was supported in part by the Fund for Scientific Research—Flanders under grant 3G001998.
Authors' addresses: S. Debray and W. Evans, Department of Computer Science, University of Arizona, Tucson, AZ 85721; email: {debray, will}@cs.arizona.edu; R. Muth, Alpha Development Group, Compaq Computer Corporation, Shrewsbury, MA 01749; email: Robert.Muth@compaq.com; B. De Sutter, Department of Electronics and Information Systems, University of Ghent, B-9000 Gent, Belgium; email: brdsutte@elis.rug.ac.be.

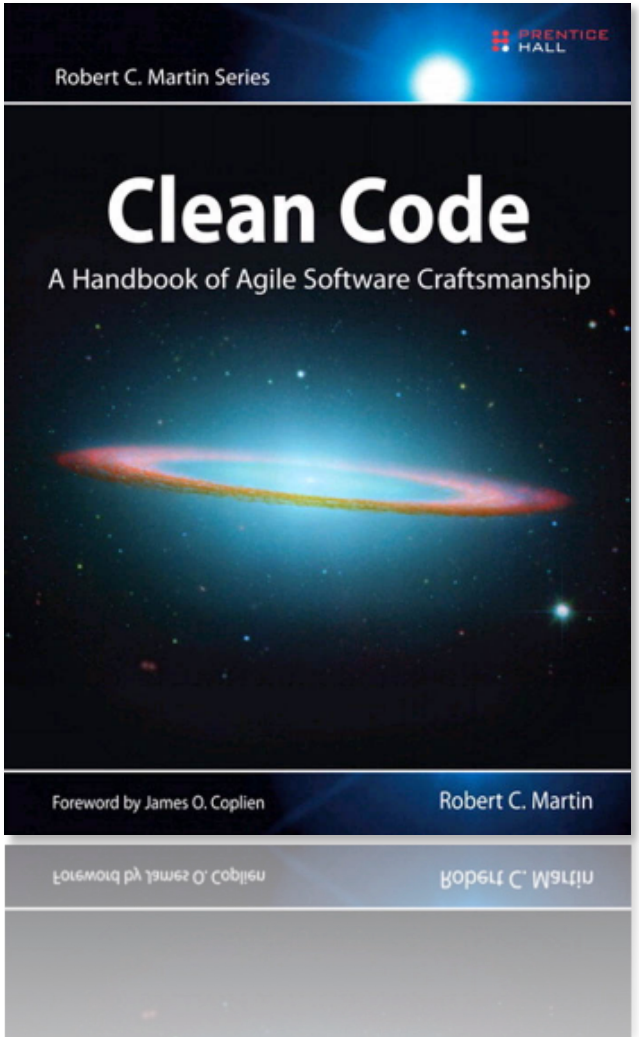
ACM Transactions on Programming Languages and Systems, Vol. 22, No. 2, March 2000, Pages 378–415.

software engineering

"Dead code is code that isn't executed"

--R. C. Martin

dead instead of unreachable




to avoid confusion we will use the term
unreachable

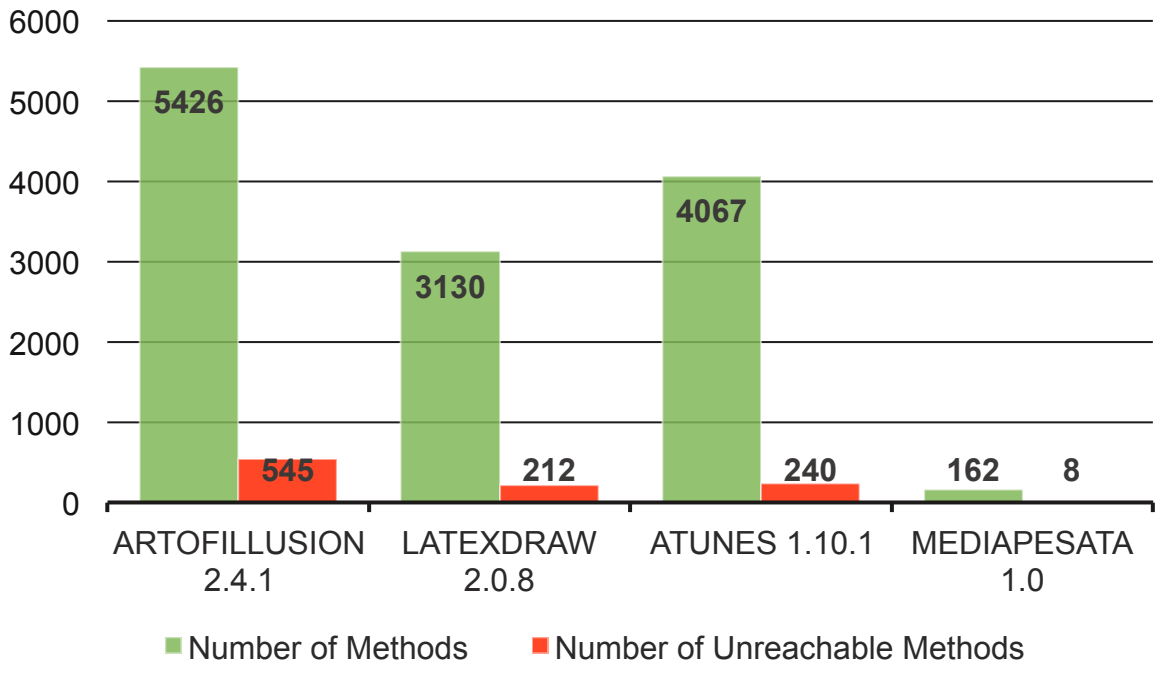
an example

```
public class M{
    public static void main(String[] args){
        C c = new C1();
        c.m1();
    }
}

public class C{
    public void m1(){…}
}

public class C1 extends C{
    public void m1(){
        this.m2();
    }
    private void m2(){…}
    private void m3(){…}
}
```

M.Main()
C1.C1()
C1.m1()
C1.m2()
C1.m3()
C.m1()

- Reachable Methods
- Unreachable Methods

percentage of unreachable methods ranges from 5% to 10%



A Graph-Based Approach to Detect Unreachable Methods in Java Software

Simone Romano
University of Basilicata
Potenza, Italy
simone.romano@unibas.it

Giuseppe Scanniello
University of Basilicata
Potenza, Italy
giuseppe.scanniello@unibas.it

Carlo Sartiani
University of Basilicata
Potenza, Italy
carlo.sartiani@unibas.it

Michele Risi
University of Salerno
Fisciano, Italy
mrisi@unisa.it

ABSTRACT

CCS Concepts

Keywords

1. INTRODUCTION

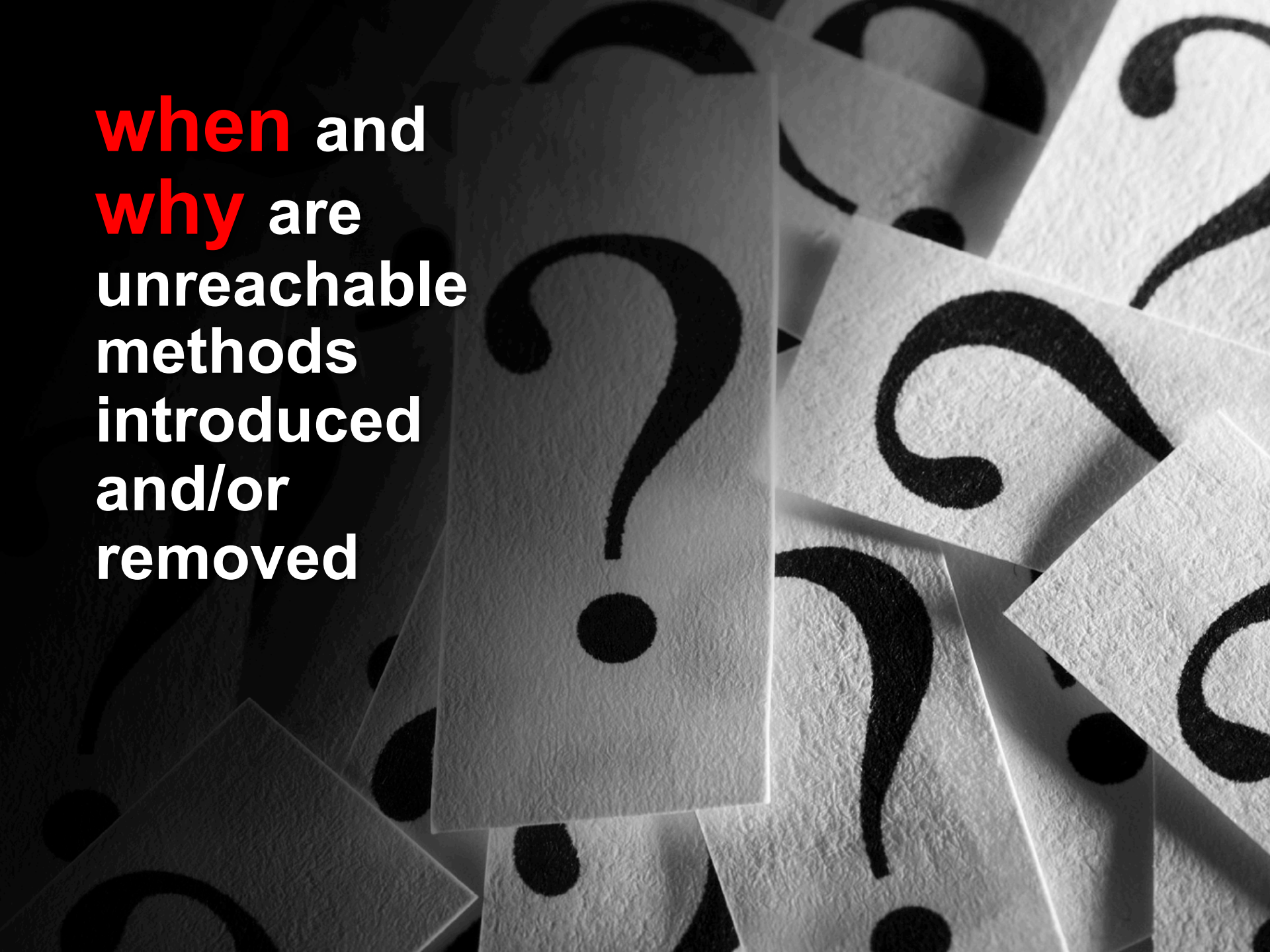
so what?

A long-term investigation

are unreachable methods really harmful?


how actual
developers
perceive
unreachable
methods?

how do developers
deal with
unreachable
methods?


when and
why are
unreachable
methods
introduced
and/or
removed

Impact of unreachable methods on comprehensibility and modifiability

Analyze their presence **for the purpose of** evaluating their effect **w.r.t.** comprehensibility of unknown code and **w.r.t.** modifiability of familiar code **from the point of view of** researchers and practitioners **in the context of** novice developers and Java code

design: one factor
with two treatments
NoUM vs. UM

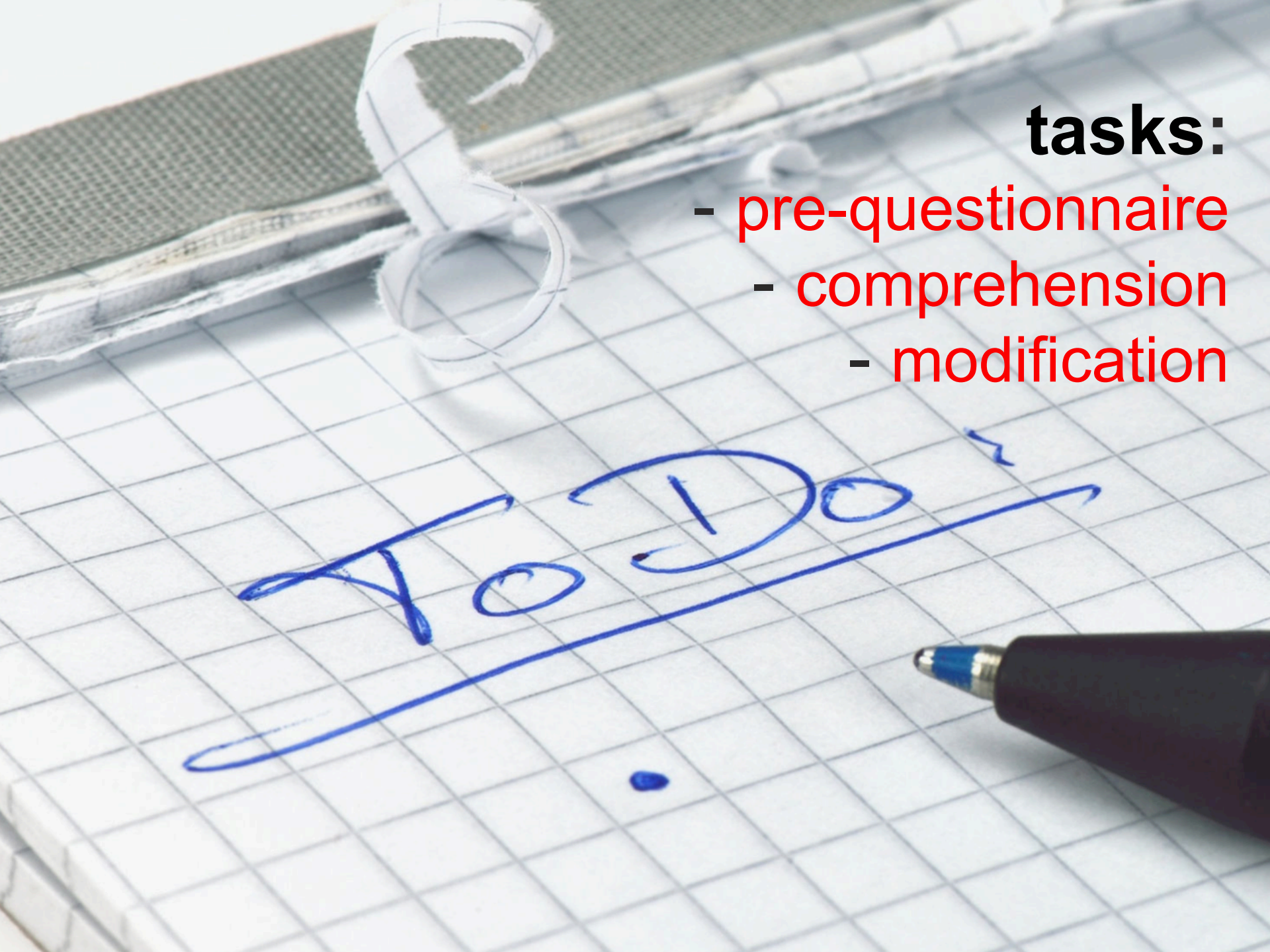
costructs:

correctness of

understanding and

of modification, and

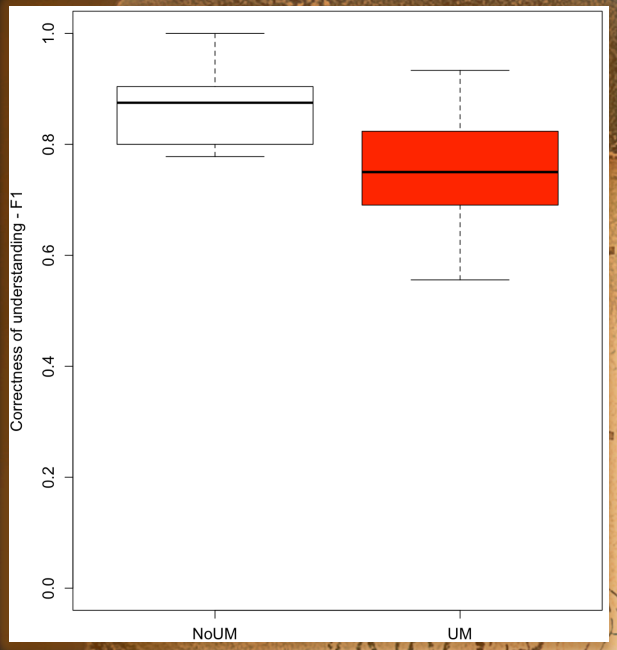
effort

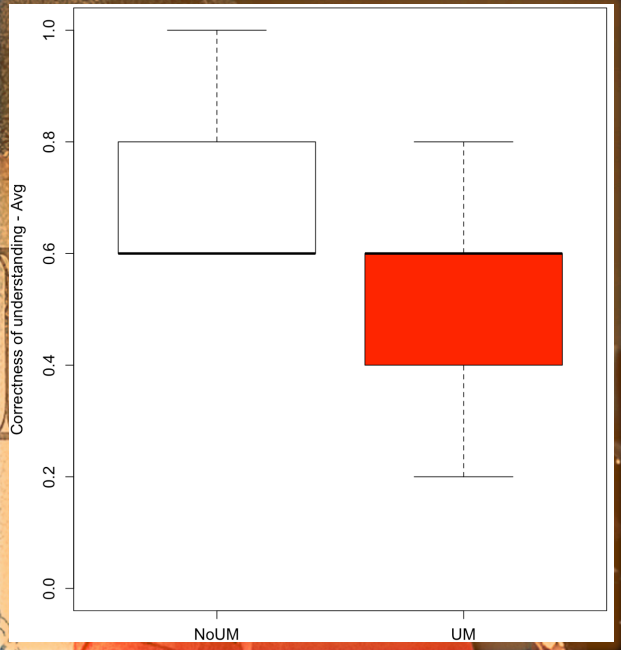


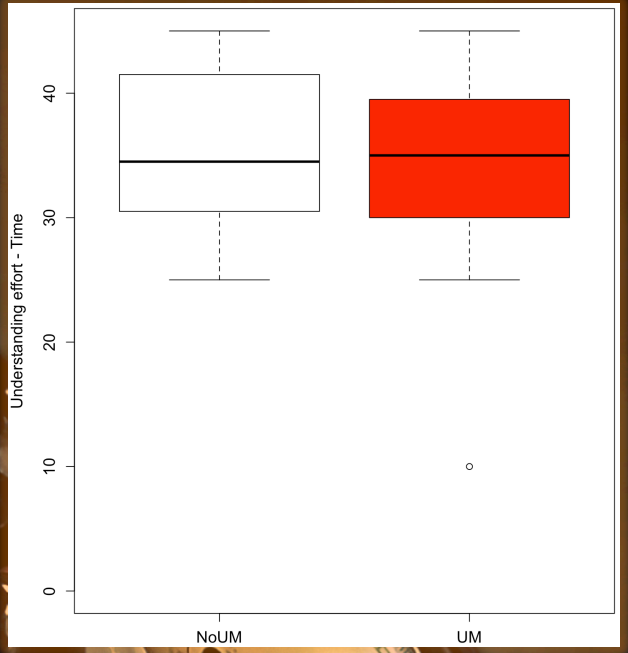
tasks:

- pre-questionnaire
- comprehension
- modification

comprehensibility

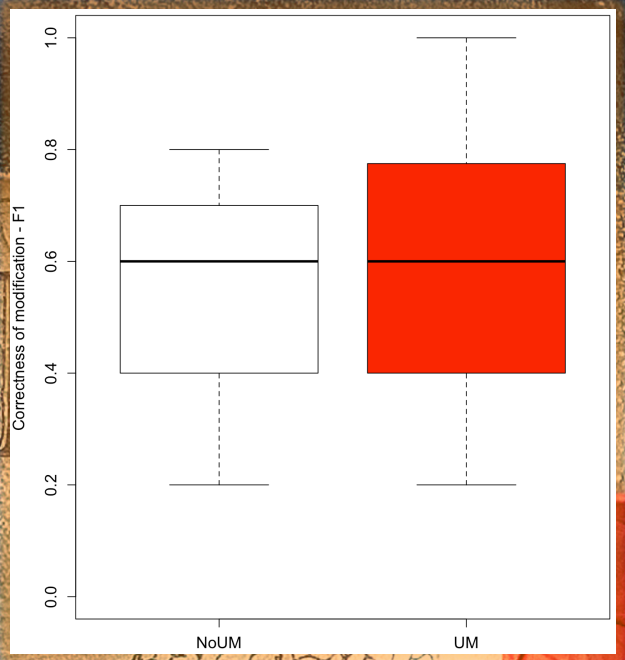


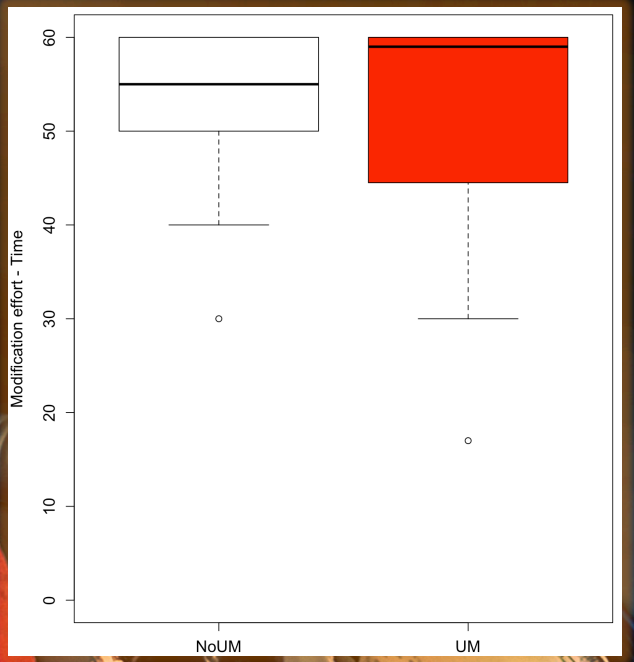




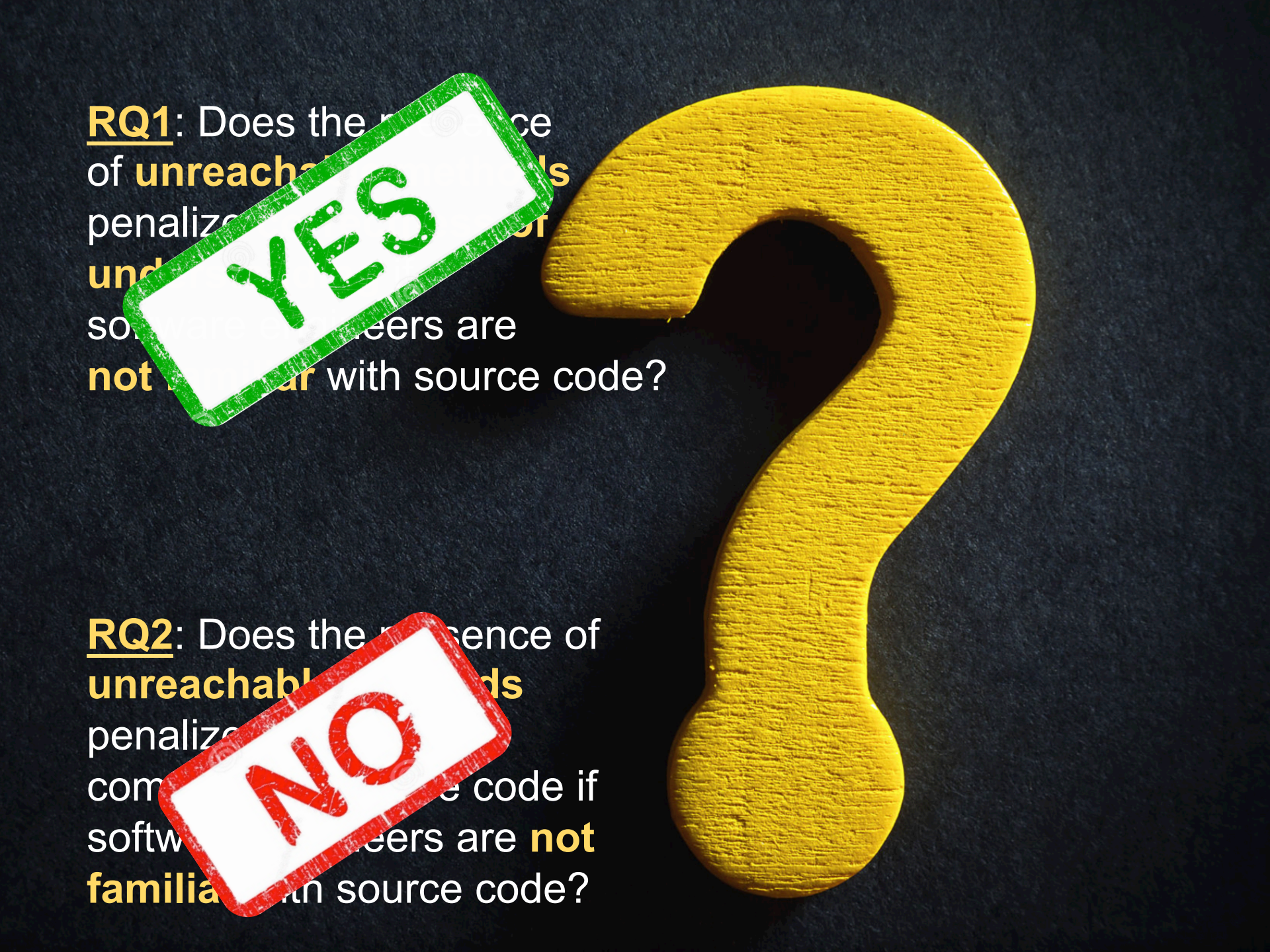
Hypothesis	Metric	p-value	Cohen/Cliff's d	Perc. difference	Stat. Power/β-value
Hn0	F_1 (+)	$<$ **0.0001**	large (1.317)	14.395%	0.925
	Avg	**0.008**	medium (0.428)	24.065%	0.835
Hn1	Time (+)	0.542	negligible (0.179)	4.886%	0.946
Hn2	F_1	0.956	negligible (0.011)	0.424%	0.955
Hn3	Time	0.575	negligible (0.094)	2.417%	0.937

maintainability





Hypothesis	Metric	p-value	Cohen/Cliff's d	Perc. difference	Stat. Power/β-value
Hn0	F_1 (+)	$<$ **0.0001**	large (1.317)	14.395%	0.925
	Avg	**0.008**	medium (0.428)	24.065%	0.835
Hn1	Time (+)	0.542	negligible (0.179)	4.096%	0.946
Hn2	F_1	0.956	negligible (0.011)	0.424%	0.955
Hn3	Time	0.575	negligible (0.094)	2.417%	0.937

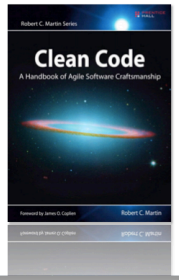

RQ1: Does the
of unreacha
penaliz
und
so
not
with source code?
YES
RQ2: Does the
unreachab
penaliz
com
code if
softw
familia
source code?
NO


RQ3: Does the
of unreach
penaliz
mod
if soft
familia
ence
ods
of
code
ineers are
ith source code?
NO
RQ4: Does the
unreachab
penaliz
sou
engi
source
sence of
ds
maintain
ware
familiar with
de?
NO

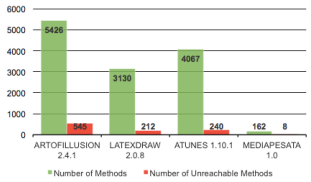
software engineering

"Dead code is code that isn't **executed"**
--R. C. Martin

dead **instead** of unreachable



percentage of unreachable methods ranges from **5%** to **10%**



are unreachable methods really harmful?

RQ3: Does the presence of unreachable methods penalize the modifiability of code if software engineers are familiar with source code?

NO

RQ4: Does the presence of unreachable methods penalize the ability to maintain source code if software engineers are familiar with source code?

NO

RQ1: Does the presence of unreachable methods penalize the understandability of code if software engineers are not familiar with source code?

YES

RQ2: Does the presence of unreachable methods penalize the comprehensibility of code if software engineers are not familiar with source code?

NO

Impact of unreachable methods on comprehensibility and modifiability

giuseppe.scanniello@unibas.it