WHAT CAN WE LEARN ABOUT

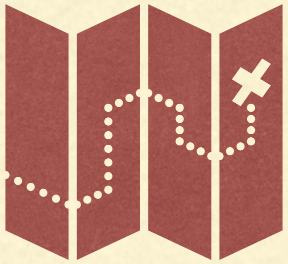
LOCATING REFACTORING OPPORTUNITIES

FROM

DECOMPOSING SOFTWARE TO MICROSERVICES

EXTRACT AND MOVE METHOD

LOCATING REFACTORING OPPORTUNITIES



SOFTWARE DECOMPOSITION

MICROSERVICES

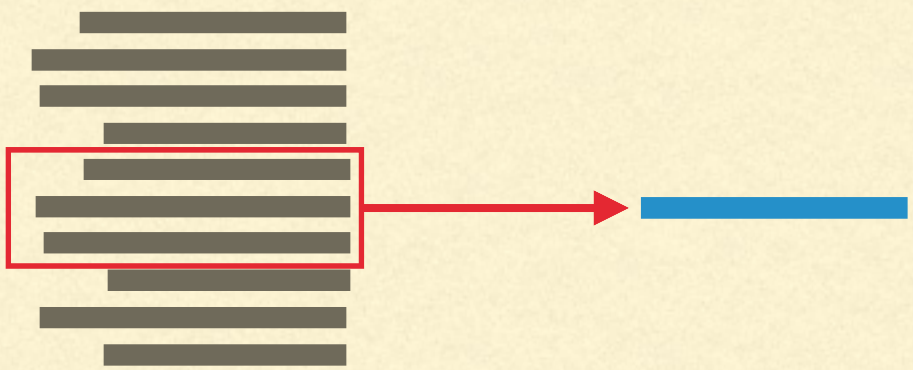
ME: PAST-CURRENT

- B.S. University of Bergen, computer science
- M.S. joint University of Bergen & Western Norwegian University of Applied Sciences (Volker Stolz)
- PhD University of Bergen, Language / Processor Co-Evolution (Anya Bagge)
- currently on research stay at (the very empirical) Software Engineering group at University of British Columbia, Canada (Gail C. Murphy)

(EXTRACT AND) MOVE METHOD

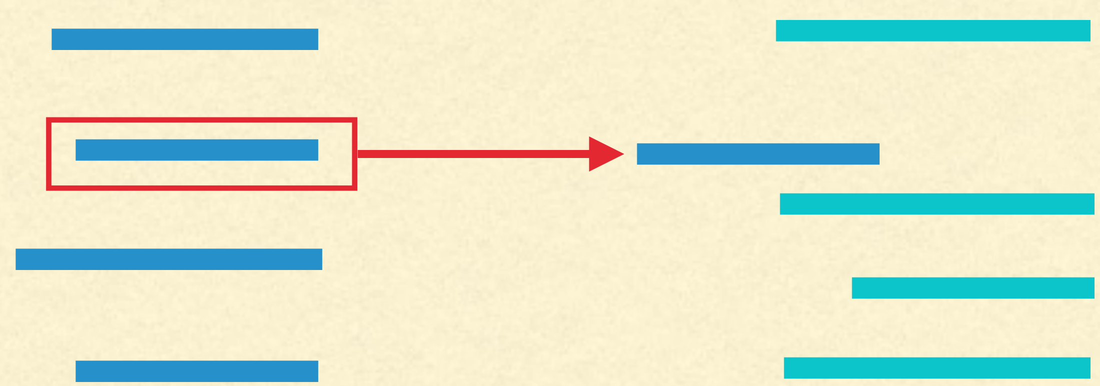
(EXTRACT AND) MOVE METHOD

Extract method



class A

Move method



class A

class B

(EXTRACT AND) MOVE METHOD

```java
public class Customer {

    public double getOwing(List<Invoice> invoices) {
        double outstanding = 0;
        for(Invoice invoice : invoices)
            outstanding+=invoice.getOutstanding();
        return outstanding + outstanding * 0.2;
    }
}
```

Extract method

```java
public class Customer {

    public double getOwing(List<Invoice> invoices) {
        double outstanding = 0;
        for(Invoice invoice : invoices)
            outstanding+=invoice.getOutstanding();
        return outstanding + getInterest(outstanding);
    }

    private double getInterest(double outstanding) {
        return outstanding * 0.2;
    }
}
```

Move method

```java
public class Customer {

    public double getOwing(List<Invoice> invoices) {
        double outstanding = 0;
        for(Invoice invoice : invoices)
            outstanding+=invoice.getOutstanding()
                            + invoice.getInterest();
        return outstanding;
    }
```

```java
public class Invoice {
    double outstanding = 0;

    public double getOutstanding() {
        return outstanding;
    }
    double getInterest() {
        return outstanding * 0.2;
    }
}
```

LOCATING REFACTORING OPPORTUNITIES

```java
public class Customer {

    public double getOwing(List<Invoice> invoices) {
        double outstanding = 0;
        for(Invoice invoice : invoices)
            outstanding+=invoice.getOutstanding();
        return outstanding + outstanding * 0.2;
    }
}
```

Extract method

```java
public class Customer {

    public double getOwing(List<Invoice> invoices) {
        double outstanding = 0;
        for(Invoice invoice : invoices)
            outstanding+=invoice.getOutstanding();
        return outstanding + getInterest(outstanding);
    }

    private double getInterest(double outstanding) {
        return outstanding * 0.2;
    }
}
```

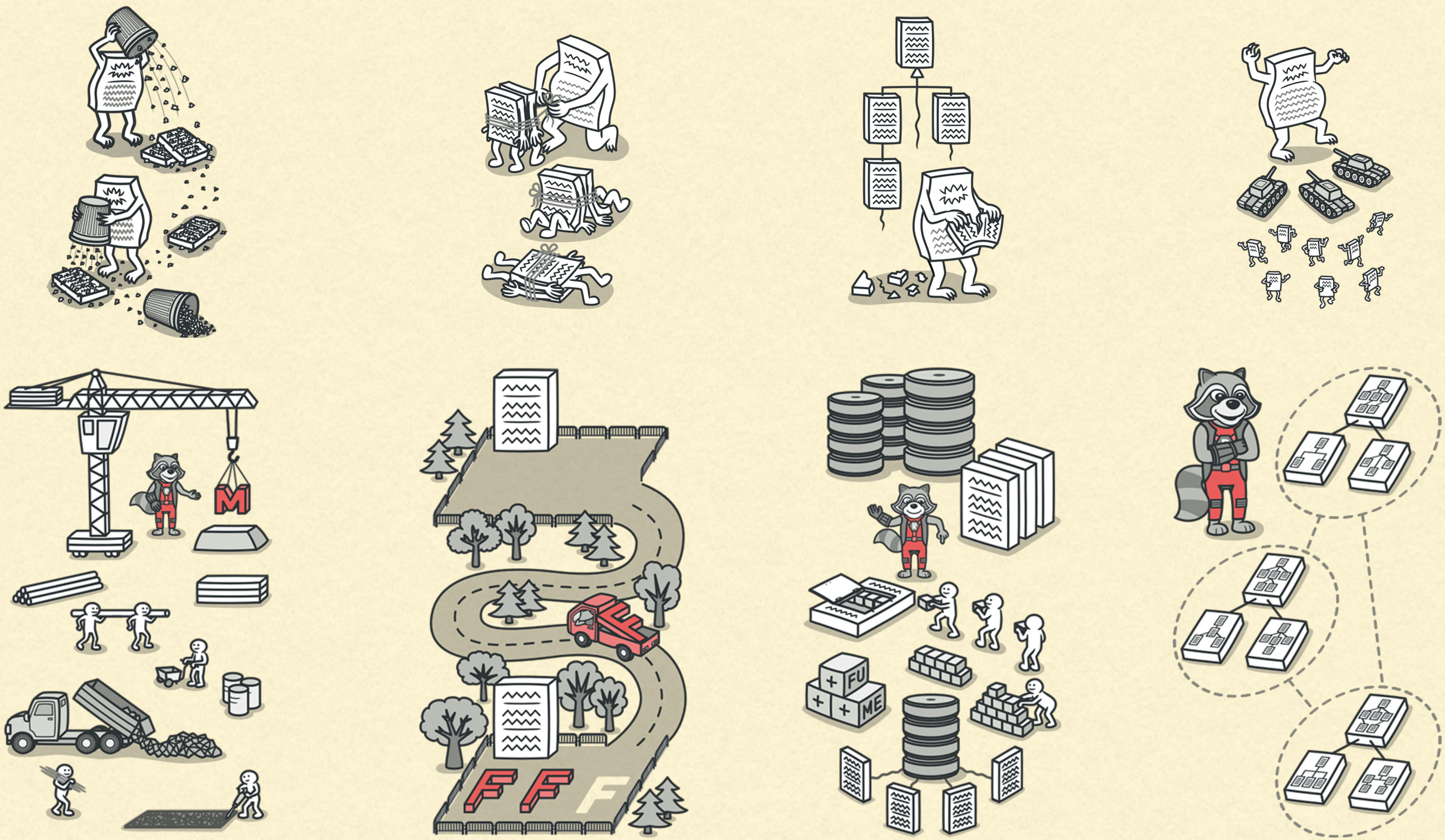
Move method

```java
public class Customer {

    public double getOwing(List<Invoice> invoices) {
        double outstanding = 0;
        for(Invoice invoice : invoices)
            outstanding+=invoice.getOutstanding()
                        + invoice.getInterest();
        return outstanding;
    }
```

```java
public class Invoice {
    double outstanding = 0;

    public double getOutstanding() {
        return outstanding;
    }
    double getInterest() {
        return outstanding * 0.2;
    }
}
```

LOCATING REFACTORING OPPORTUNITIES

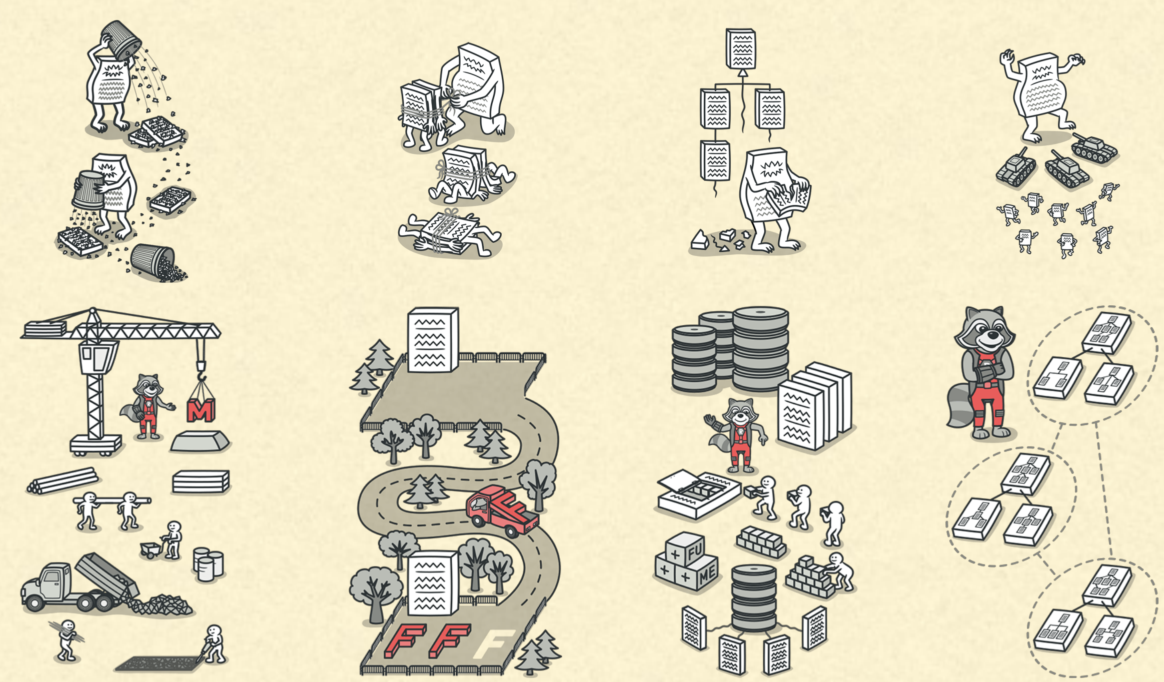
WHEN TO REFACTOR



https://refactoring.guru/refactoring/catalog

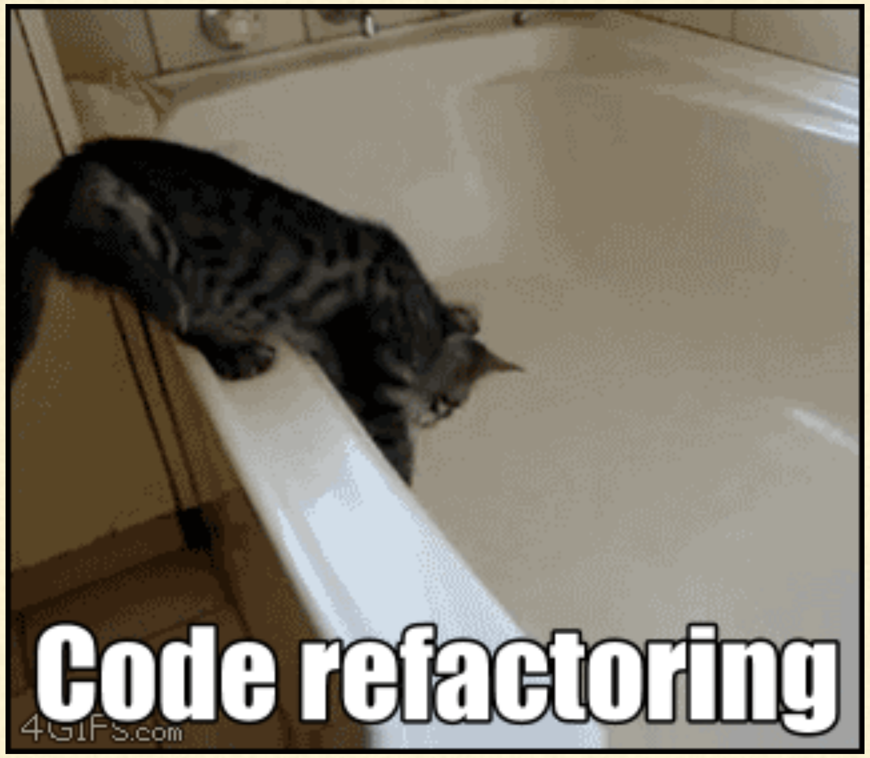
LOCATING REFACTORING OPPORTUNITIES

- all possible
- smell-reducing
- machine learning
- metrics-aware
- heuristics



https://refactoring.guru/refactoring/catalog

LOCATING REFACTORING OPPORTUNITIES



USE, DISUSE, MISUSE

Use, Disuse, and Misuse of Automated Refactorings

Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, Ralph E. Johnson
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{*mvakili2, nchen, snegara2, rajkuma1, bpbailey, rjohnson*}*@illinois.edu*

***Abstract*—Though refactoring tools have been available for more than a decade, research has shown that programmers underutilize such tools. However, little is known about why programmers do not take advantage of these tools. We have conducted a field study on programmers in their natural settings working on their code. As a result, we collected a set of interaction data from about 1268 hours of programming using our minimally intrusive data collectors. Our quantitative data show that programmers prefer lightweight methods of invoking refactorings, usually perform small changes using the refactoring tool, proceed with an automated refactoring even when it may change the behavior of the program, and rarely preview the automated refactorings. We also interviewed nine of our participants to provide deeper insight about the patterns that we observed in the behavioral data. We found that programmers use predictable automated refactorings even if they have rare bugs or change the behavior of the program. This paper reports some of the factors that affect the use of automated refactorings such as invocation method, awareness, naming, trust, and predictability and the major mismatches between programmers' expectations and automated refactorings. The results of this work contribute to producing more effective tools for refactoring complex software.**

***Keywords*-Software engineering; Software maintenance; Programming environments; Human factors; User interfaces; Human computer interaction**

I. Introduction

Refactoring is defined as changing the design of software without affecting its observable behavior [1]. Refactorings rename, move, split, and join program elements such as fields, methods, packages, and classes. Agile software processes such as eXtreme Programming (XP) prescribe refactoring [2], because it enables evolutionary software design and is the key to modifiable and readable code [3]. Programmers refactor their code frequently [4], [5]. Some refactorings are tedious and error-prone to perform manually. Thus, automated refactorings were invented more than a decade ago to make the process of refactoring more efficient and reliable [6]. Today, modern Integrated Development Environments (IDEs), such as Eclipse [7], NetBeans [8], IntelliJ IDEA [9], Xcode [10], and ReSharper [11], support many automated refactorings.

Recently, there has been much interest in improving the reliability of existing automated refactorings and building new ones to automate sophisticated program transformations [12]–[16]. This is not surprising, given the tedium and error-proneness of some refactorings and the perceived benefits of their automation. In spite of the growing interest in improving the usability of automated refactorings [17]–[19], this aspect of refactoring has not received enough attention. For example, the user interfaces of refactoring tools have changed little since they were first introduced, and recent studies suggest that programmers greatly underutilize the existing refactoring tools [5]. We need to understand the problems programmers have with today's refactoring tools to design future generations of these tools that fit programmers' needs.

We conducted a study consisting of both quantitative and qualitative data collection. We studied 26 developers working in their natural settings on their code for a total of 1268 programming hours over three months, and collected data about their interactions with automated refactorings. We observed patterns of interaction in our quantitative data and interviewed nine of our participants to take a more detailed qualitative look at our behavioral data. Then, we adapted a general framework of human-automation interaction [20] to frame the use, disuse, and misuse of automated refactorings. *Use* of automated refactorings refers to programmers applying automated refactorings to perform code changes they might otherwise do manually. *Disuse* of automated refactorings is programmers' neglect or underuse of automated refactorings. *Misuse* of automated refactorings refers to programmers' use of these tools in ways not recommended by the designers.

Our empirical study sheds light on how users interact with automated refactorings. First, we have found that a single context-aware and lightweight method of invoking refactorings accounts for a significant number of refactoring invocations (See Section III). Second, we have found several factors that lead to the underutilization of automated refactorings such as need, awareness, naming, trust, predictability, and configuration (See Section IV). Third, we have found that programmers usually continue an automated refactoring that has reported some error or warning. This finding casts doubt on the main property of automated refactorings, namely, behavior-preservation. In addition, we have observed some unjustified uses of the refactoring tool (See Section V). Finally, we have proposed alternative ways of designing refactoring tools based on the findings of our study (See Subsections III-B, IV-G, and V-C).

978-1-4673-1067-3/12/$31.00 © 2012 IEEE 233 ICSE 2012, Zurich, Switzerland

Our interviewees did not use automated refactorings that they had found to have complex user interfaces and unclear benefits. In general, if the benefits of automation are not readily apparent, humans are less likely to use the automation because of the cognitive overhead involved in evaluating and using the automation

On the other hand, programmers appreciate the tools that propose applicable refactorings, and are willing to use automated refactorings even when they may change the program's behavior.

LARGE-SCALE REFACTORING

A Field Study of Refactoring Challenges and Benefits

Miryung Kim * miryung@ece.utexas.edu
Thomas Zimmermann + tzimmer@microsoft.com
Nachiappan Nagappan + nachin@microsoft.com

* The University of Texas at Austin, TX, USA
+ Microsoft Research, Redmond, WA, USA

ABSTRACT

It is widely believed that refactoring improves software quality and developer productivity. However, few empirical studies quantitatively assess refactoring benefits or investigate developers' perception towards these benefits. This paper presents a field study of refactoring benefits and challenges at Microsoft through three complementary study methods: a survey, semi-structured interviews with professional software engineers, and quantitative analysis of version history data. Our survey finds that the refactoring definition in practice is not confined to a rigorous definition of *semantics-preserving code transformations* and that developers perceive that refactoring involves substantial cost and risks. We also report on interviews with a designated refactoring team that has led a multi-year, centralized effort on refactoring Windows. The quantitative analysis of Windows 7 version history finds that the binary modules refactored by this team experienced significant reduction in the number of inter-module dependencies and post-release defects, indicating a visible benefit of refactoring.

Categories and Subject Descriptors:
D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—*restructuring*

General Terms: Measurement, Experimentation

Keywords: Refactoring; empirical study; software evolution; component dependencies; defects; churn.

1. INTRODUCTION

It is widely believed that refactoring improves software quality and developer productivity by making it easier to maintain and understand software systems [13]. Many believe that a lack of refactoring incurs technical debt to be repaid in the form of increased maintenance cost [5]. For example, eXtreme Programming claims that refactoring saves development cost [4] and advocates the rule of *refactor mercilessly* throughout the entire project life cycles. On the other hand, there exists a conventional wisdom that software engineers often avoid refactoring, when they are constrained by a lack of resources (e.g., right before major software releases). Some also believe that refactoring does not provide immediate benefit unlike new features or bug fixes.

Recent empirical studies show contradicting evidence on the benefit of refactoring as well. Ratzinger et al. [29] found that, if the number of refactoring edits increases in the preceding time period, the number of defects decreases. On the other hand, Weißgerber and Diehl found that a high ratio of refactoring edits is often followed by an increasing ratio of bug reports [34, 35] and that incomplete or incorrect refactorings cause bugs [14]. In our previous study, we found similar evidence that refactoring edits have a strong temporal and spatial correlation with bug fixes [18].

These contradicting findings motivated us to conduct a field study of refactoring definition, benefits, and challenges in a large software development organization and investigate whether there is a visible benefit of refactoring a large system. In this paper, we address the following research questions: (1) What is the definition of refactoring from developers' perspectives? By refactoring, do developers indeed mean behavior-preserving code transformations or changes to a program structure [23, 13]? (2) What is the developers' perception about refactoring benefits and risks, and in which contexts do developers refactor code? (3) As claimed in the literature, are there visible refactoring benefits such as reduction in the number of bugs, reduction in the average size of code changes after refactoring, and reduction in the number of component dependencies?

To investigate the definition of refactoring in practice and the value perception toward refactoring, we conducted a survey with over three hundred engineers whose check-in comments included a keyword "*refactor**" in the last two years. From our survey participants, we also came to know about a multi-year refactoring effort on Windows. Because Windows is one of the largest, long-surviving software systems within Microsoft and a designated team led an intentional effort of system-wide refactoring, we focused on the case study of Windows. We interviewed the refactoring team and then assessed the impact of the team's refactoring on reduction of inter-module dependencies and post-release defects using Windows 7 version history.

Our field study found the following results:

- The refactoring definition in practice seems to differ from a rigorous academic definition of *behavior-preserving program transformations*. Our survey participants perceived that refactoring involves substantial

SIGSOFT/FSE'12 November 10 - 18 2012, Raleigh, NC, USA

328 engineers participated in the survey.

"The value of refactoring is difficult to measure. How do you measure the value of a bug that never existed, or the time saved on a later undetermined feature? How does this value bubble up to management? Because there's no way to place immediate value on the practice of refactoring, it makes it difficult to justify to management."

"These (Fowler's refactoring types or refactoring types supported by Visual Studio) are the small code transformation tasks often performed, but they are unlikely to be performed alone. There's usually a bigger architectural change behind them."

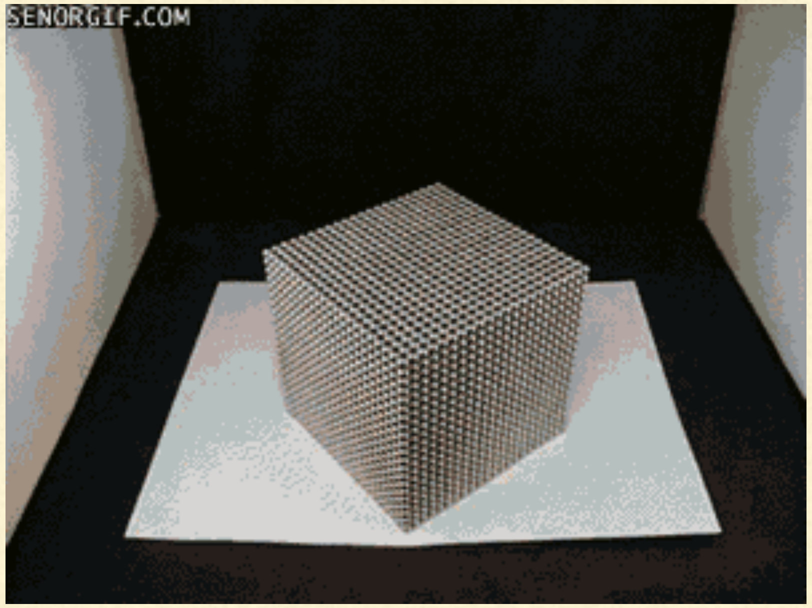
"I'd love a tool that could estimate the benefits of refactoring.

SOFTWARE DECOMPOSITION

SOFTWARE DECOMPOSITION

"Finding, or creating, 'seams' in your code base"

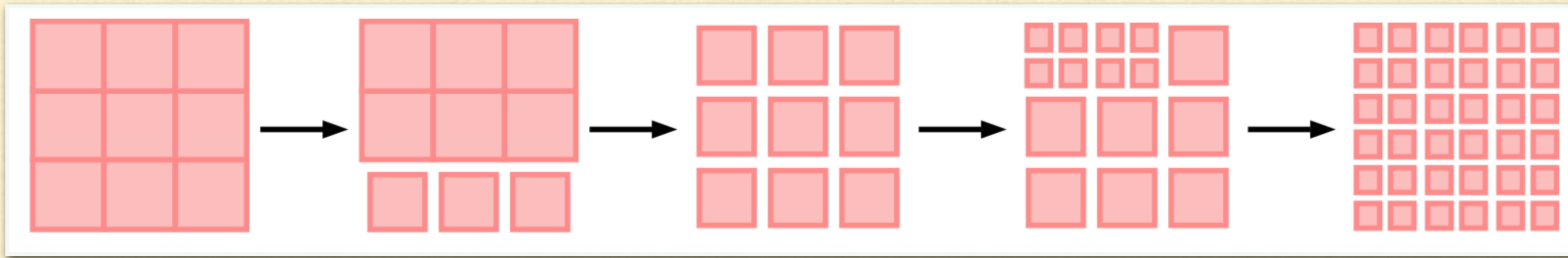
- Michael Feathers, Working Effectively with Legacy Code

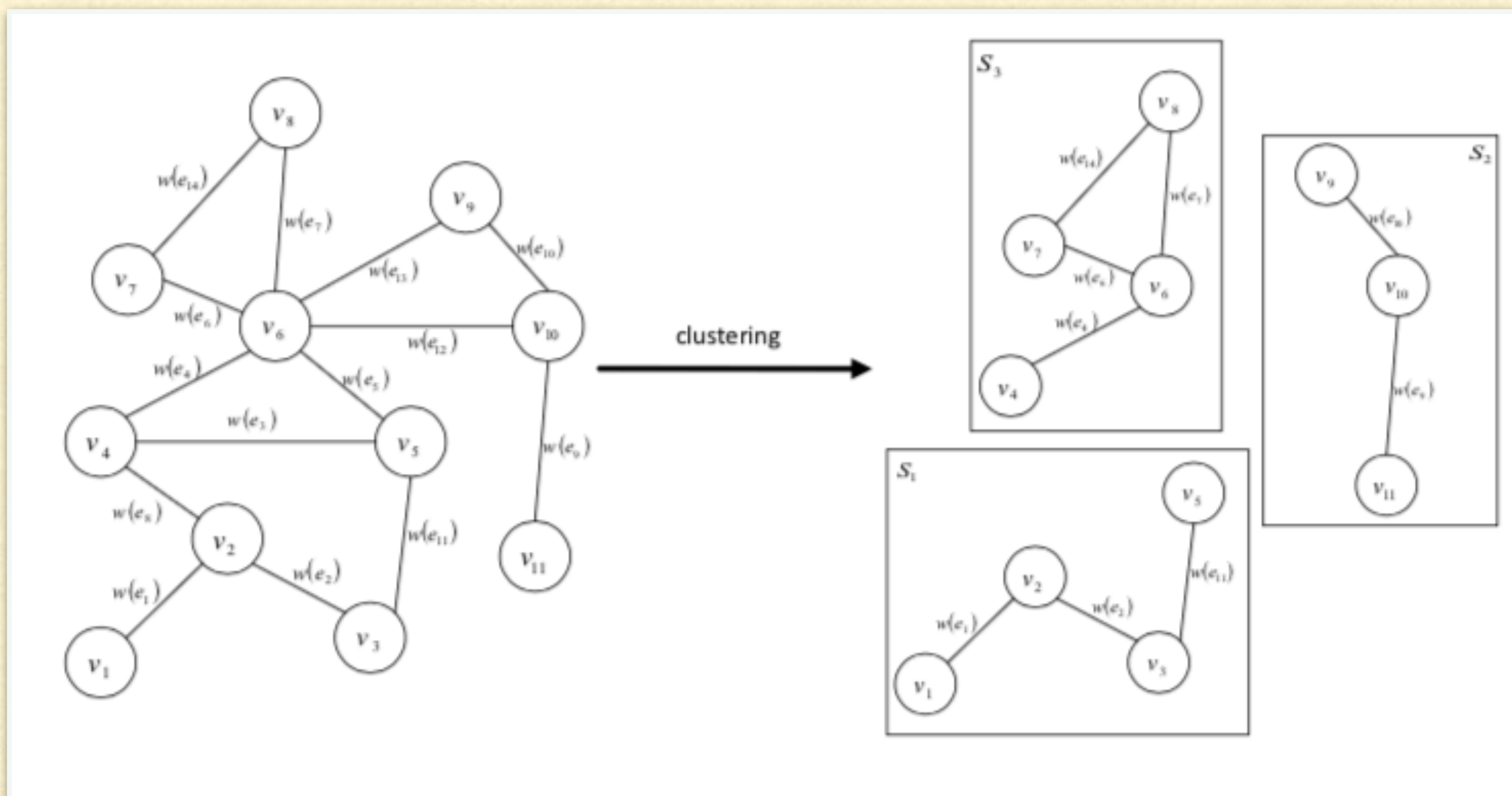


MICROSERVICES

- Clear areas of responsibility
- Strong encapsulation
- Individually deployable

DECOMPOSING MONOLITHS TO MICROSERVICES





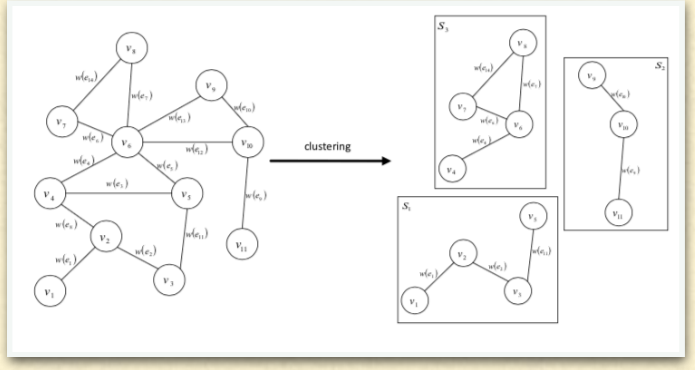
Top: Yarygina2018 Bottom: Mazlami2017

DECOMPOSITION TO MICROSERVICES

2016-2018

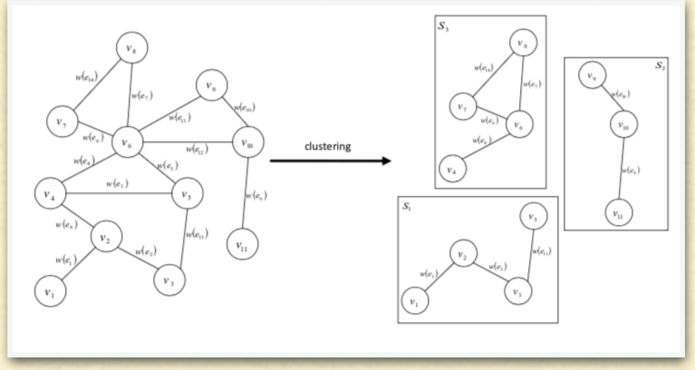
DECOMPOSITION TO MICROSERVICES

- MVC-based [Levcovitz2016]
- resource based [Levcovitz2016, Mazlami2017, Gysel2016]
- metrics-based, source code analysis (k-clustering) [Mazlami2017, Gysel2016, Selmadji2018]
- team structure [Mazlami2017]
- interface analysis (semantic) [Baresi2017]

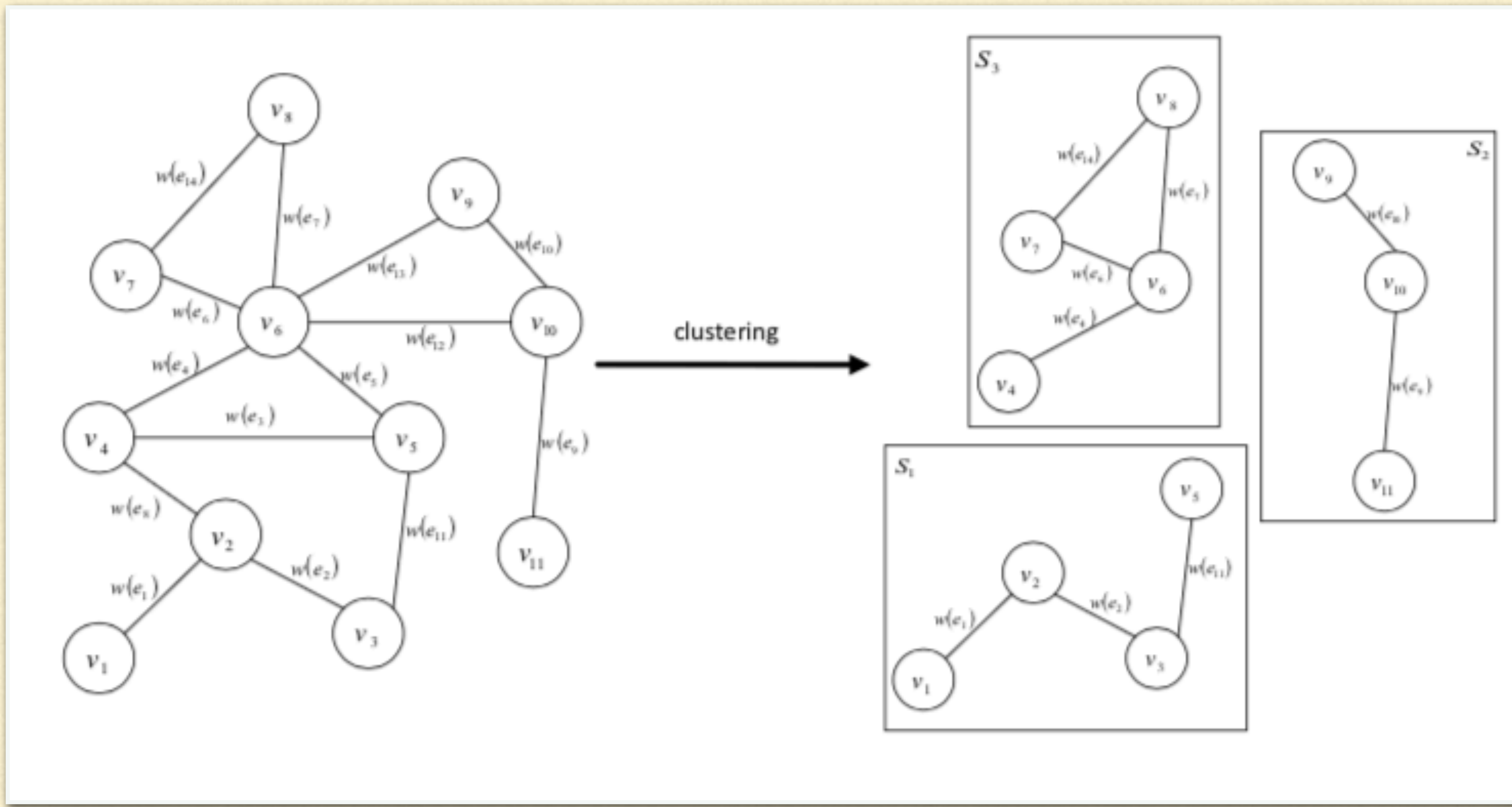


LOCATING REFACTORING OPPORTUNITIES?

- MVC-based [Levcovitz2016]
- resource based [Levcovitz2016, Mazlami2017, Gysel2016]
- metrics-based, source code analysis (k-clustering) [Mazlami2017, Gysel2016, Selmadji2018]
- team structure [Mazlami2017]
- interface analysis (semantic) [Baresi2017]



LOCATING REFACTORING OPPORTUNITIES



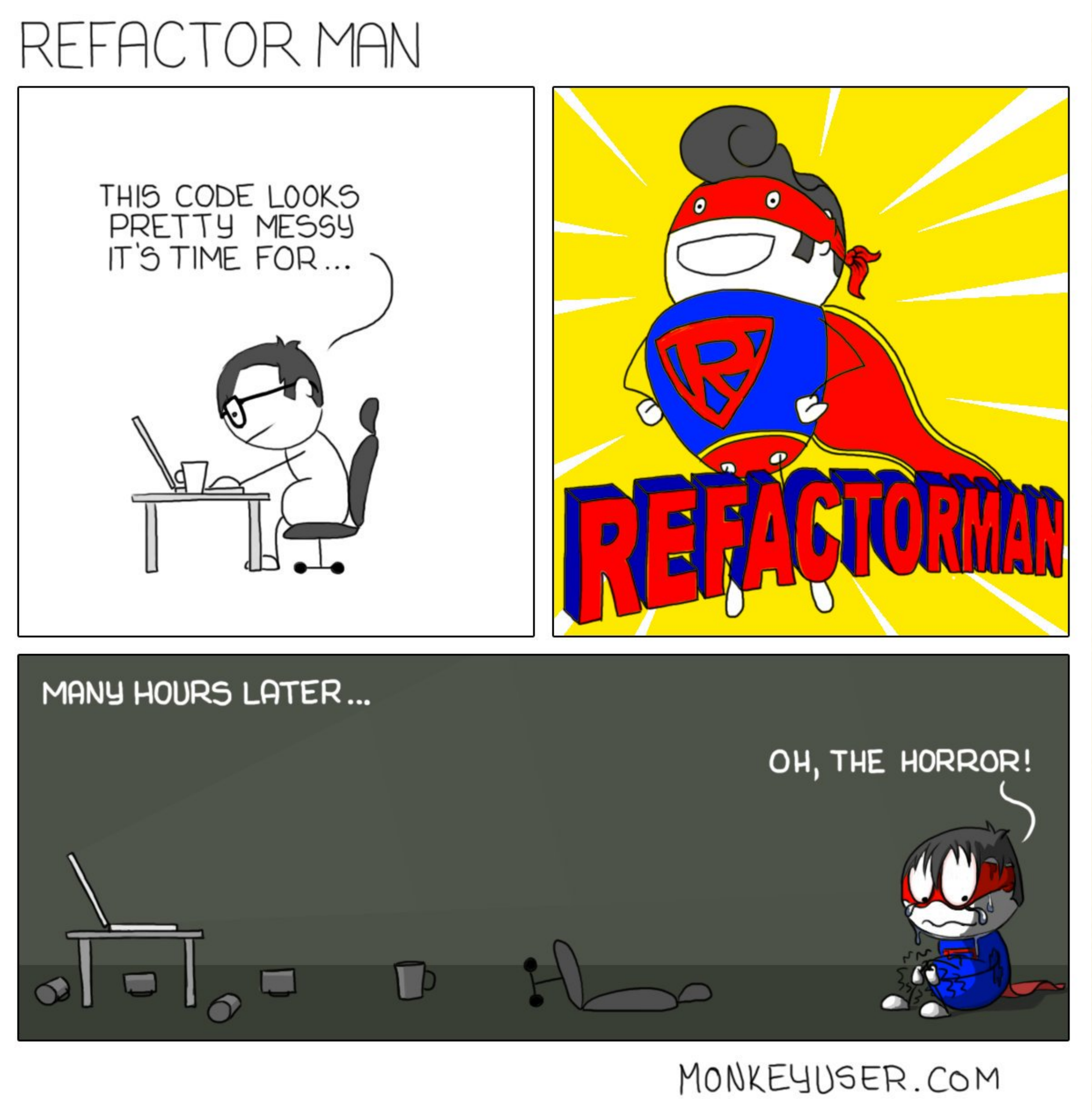
Anna Maria Eilertsen

University of Bergen, Norway

https://annaei.github.io/

@sowhow

anna.eilertsen@uib.no



REFERENCES

[Gysel2016]
Service Cutter: A Systematic Approach to Service Decomposition
Michael Gysel1, Lukas Kölbener1, Wolfgang Giersche2
and Olaf Zimmermann1
IFIP International Federation for Information Processing 2016

[Mazlami2017]
Extraction of Microservices from Monolithic Software Architectures
Genc Mazlami, Ju¨rgen Cito, Philipp Leitner
2017 IEEE 24th International Conference on Web Services

[Selmadji2018[
Re-architecting OO Software into Microservices A Quality-Centred Approach
Anfel Selmadji(B), Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, Christophe Dony, and Rahina Oumarou Mahamane
IFIP International Federation for Information Processing 2018

[Baresi2017]
Microservices Identification through Interface Analysis
Luciano Baresi1, Martin Garriga1, and Alan De Renzis2
Conference Paper · September 2017

[Levcovitz2016]
Levcovitz, A., Terra, R., Valente, M.T.: Towards a technique for extracting microservices from monolithic enterprise systems. arXiv preprint (2016)

[Yarygina2018]
Yarygina, Tetiana. "Exploring Microservice Security." (2018).