Identifying Code Smells

Selenium Conference 2023

About me

- Benjamin Bischoff
- Test Automation Engineer @ trivago N.V.
- 23 years in IT
- Last 8 years in testing

Disclaimer

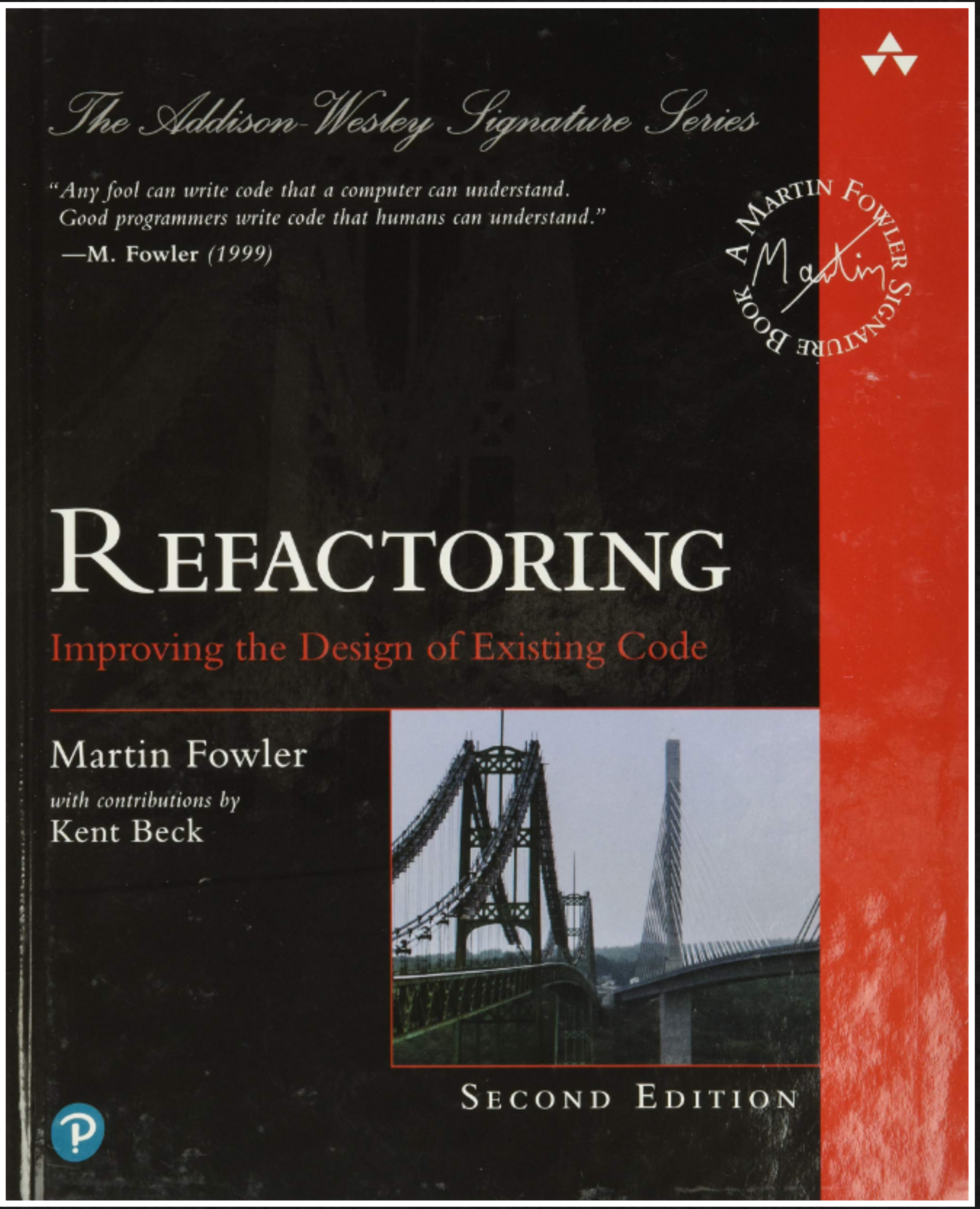
- This is only about identification
- Lots of code
- All Java

Smelly Code

Smelly code, smelly code
How are they treating you?

Smelly code, smelly code
It's not your fault.

The term


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

„A code smell is a surface indication that usually corresponds to a deeper problem in the system.“

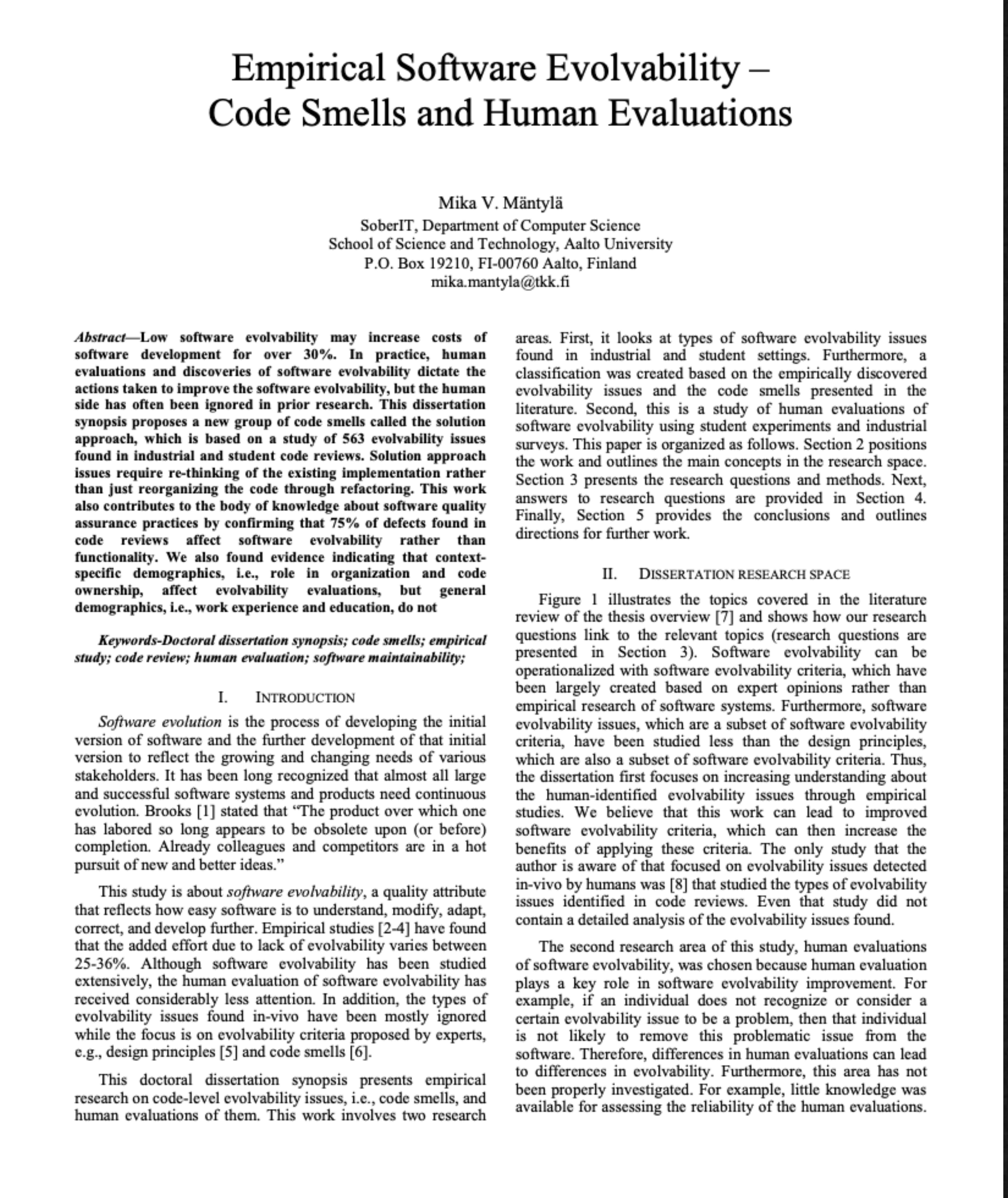
–Martin Fowler

Why should you care?

- Communication
  Speak a common language
- Better code
  Clean code principles and design patterns
- Testability
  Testable code is maintainable code!

TAXONOMY OF CODE SMELLS

Mika V. Mäntylä & Casper Lassenius, Helsinki University of Technology



Empirical Software Evolvability – Code Smells and Human Evaluations

Mika V. Mäntylä
SoberIT, Department of Computer Science
School of Science and Technology, Aalto University
P.O. Box 19210, FI-00760 Aalto, Finland
mika.mantyla@tkk.fi

***Abstract*—Low software evolvability may increase costs of software development for over 30%. In practice, human evaluations and discoveries of software evolvability dictate the actions taken to improve the software evolvability, but the human side has often been ignored in prior research. This dissertation synopsis proposes a new group of code smells called the solution approach, which is based on a study of 563 evolvability issues found in industrial and student code reviews. Solution approach issues require re-thinking of the existing implementation rather than just reorganizing the code through refactoring. This work also contributes to the body of knowledge about software quality assurance practices by confirming that 75% of defects found in code reviews affect software evolvability rather than functionality. We also found evidence indicating that context-specific demographics, i.e., role in organization and code ownership, affect evolvability evaluations, but general demographics, i.e., work experience and education, do not**

Keywords-Doctoral dissertation synopsis; code smells; empirical study; code review; human evaluation; software maintainability;

I. Introduction

Software evolution is the process of developing the initial version of software and the further development of that initial version to reflect the growing and changing needs of various stakeholders. It has been long recognized that almost all large and successful software systems and products need continuous evolution. Brooks [1] stated that "The product over which one has labored so long appears to be obsolete upon (or before) completion. Already colleagues and competitors are in a hot pursuit of new and better ideas."

This study is about *software evolvability*, a quality attribute that reflects how easy software is to understand, modify, adapt, correct, and develop further. Empirical studies [2-4] have found that the added effort due to lack of evolvability varies between 25-36%. Although software evolvability has been studied extensively, the human evaluation of software evolvability has received considerably less attention. In addition, the types of evolvability issues found in-vivo have been mostly ignored while the focus is on evolvability criteria proposed by experts, e.g., design principles [5] and code smells [6].

This doctoral dissertation synopsis presents empirical research on code-level evolvability issues, i.e., code smells, and human evaluations of them. This work involves two research areas. First, it looks at types of software evolvability issues found in industrial and student settings. Furthermore, a classification was created based on the empirically discovered evolvability issues and the code smells presented in the literature. Second, this is a study of human evaluations of software evolvability using student experiments and industrial surveys. This paper is organized as follows. Section 2 positions the work and outlines the main concepts in the research space. Section 3 presents the research questions and methods. Next, answers to research questions are provided in Section 4. Finally, Section 5 provides the conclusions and outlines directions for further work.

II. Dissertation Research Space

Figure 1 illustrates the topics covered in the literature review of the thesis overview [7] and shows how our research questions link to the relevant topics (research questions are presented in Section 3). Software evolvability can be operationalized with software evolvability criteria, which have been largely created based on expert opinions rather than empirical research of software systems. Furthermore, software evolvability issues, which are a subset of software evolvability criteria, have been studied less than the design principles, which are also a subset of software evolvability criteria. Thus, the dissertation first focuses on increasing understanding about the human-identified evolvability issues through empirical studies. We believe that this work can lead to improved software evolvability criteria, which can then increase the benefits of applying these criteria. The only study that the author is aware of that focused on evolvability issues detected in-vivo by humans was [8] that studied the types of evolvability issues identified in code reviews. Even that study did not contain a detailed analysis of the evolvability issues found.

The second research area of this study, human evaluations of software evolvability, was chosen because human evaluation plays a key role in software evolvability improvement. For example, if an individual does not recognize or consider a certain evolvability issue to be a problem, then that individual is not likely to remove this problematic issue from the software. Therefore, differences in human evaluations can lead to differences in evolvability. Furthermore, this area has not been properly investigated. For example, little knowledge was available for assessing the reliability of the human evaluations.

Classification

Bloaters	OOP Abusers	Change Preventers	Dispensables	Couplers
Long Method	Switch Statements	Divergent Change	Lazy Class	Feature Envy
Large Class	Temporary Field	Shotgun Surgery	Data Class	Inappropriate Intimacy
Primitive Obsession	Refused Bequest	Parallel Inheritance Hierarchy	Duplicate Code	Message Chains
Long Parameter List	Alternative classes with different Interfaces		Dead Code	Middleman
Data Clumps			Speculative Generality	

Bloaters

Too large to handle.

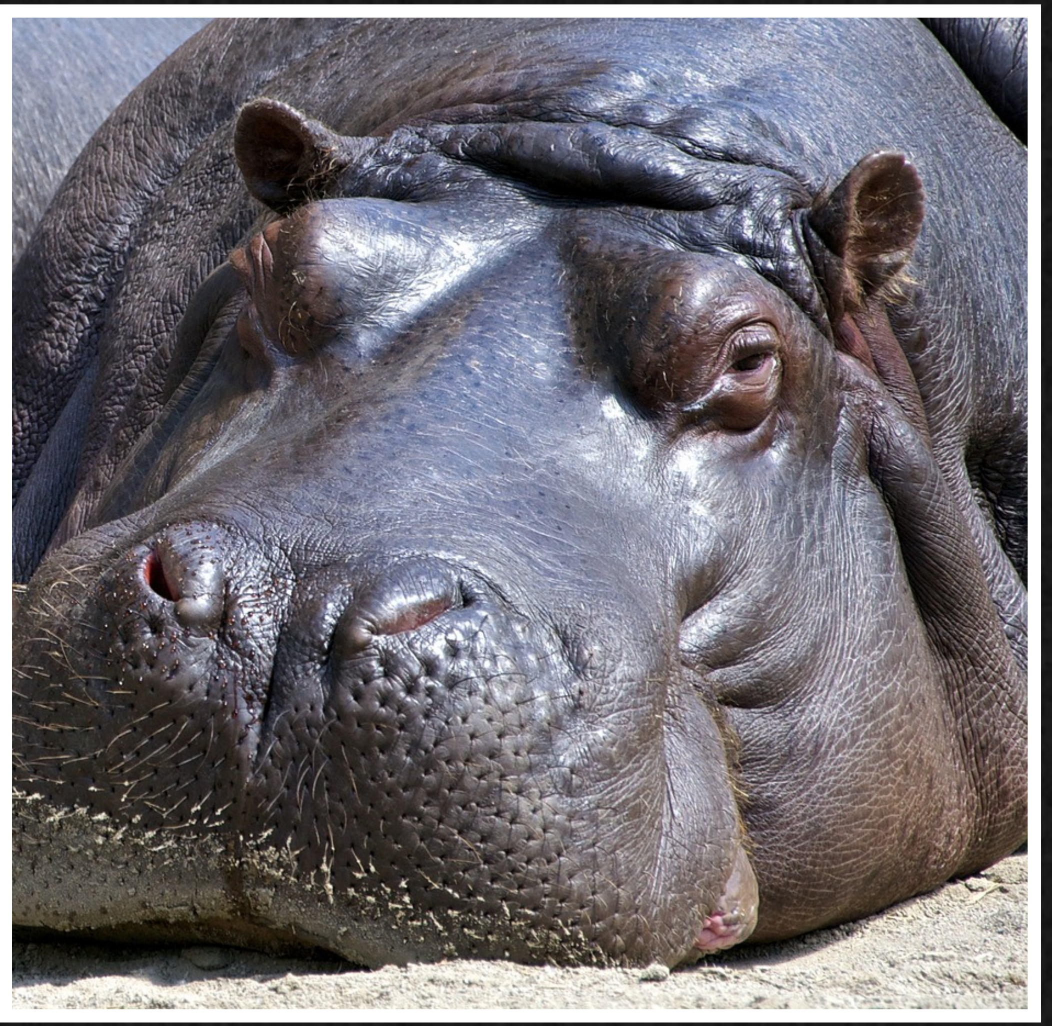
Bloaters Long Method

```
public class WebShop {
    private final List<Customer> customers = new ArrayList<>();

    public String saveCustomer(final Customer customer) {
        customers.add(customer);

        return String.format(
            "We have a new customer called %",
            customer.name());
    }
}
```

A method that does too much

Bloaters Large Class

```java
public class ShoppingCart {
    public void addProduct(final Product product) {}
    public void removeProduct(final Product product) {}
    public void checkOut() {}
    public void enterVoucherCode() {}
    public void contactSupport() {}
}
```



A class that does too much

Bloaters Primitive Obsession

```
record Customer(
        String name,
        int age,
        String street,
        String city,
        int zipCode,
        String country
) {
}
```

Favouring primitive data types

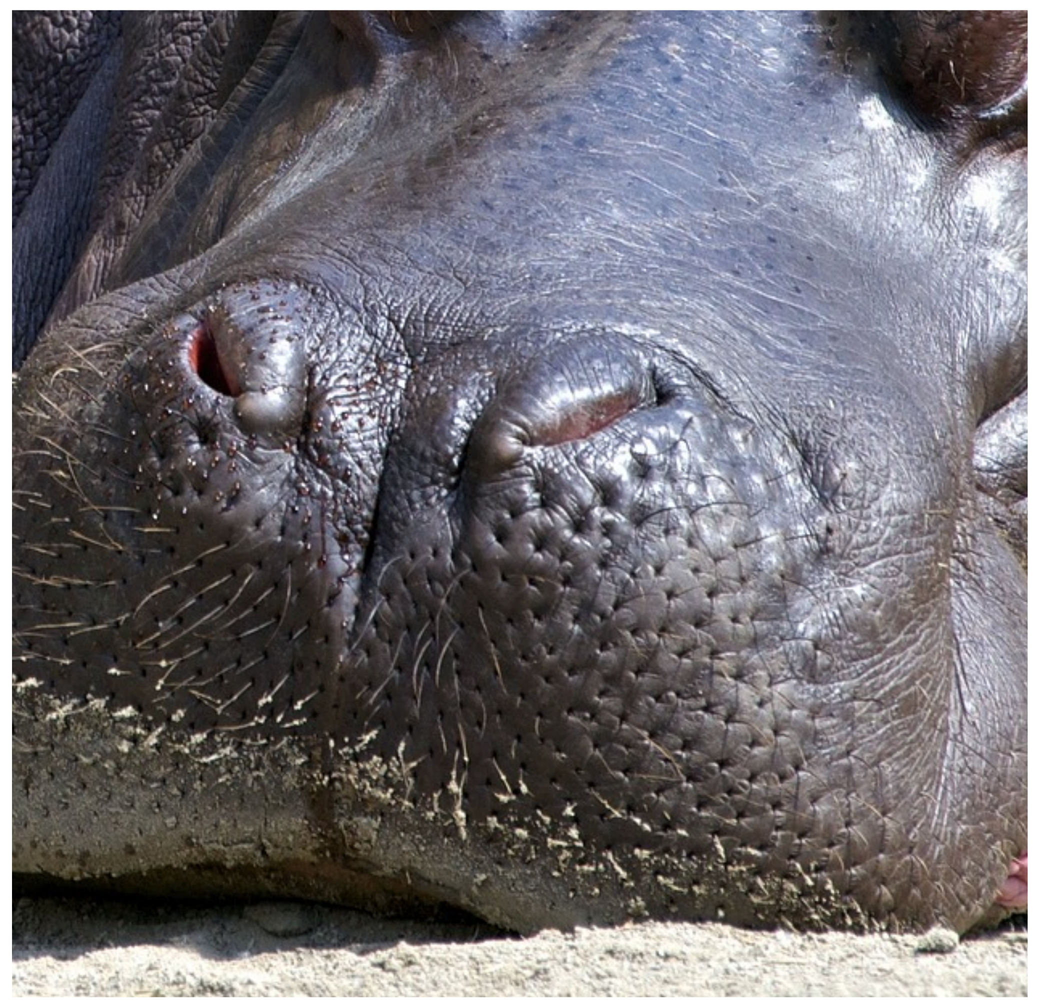
Bloaters Long Parameter List

```
public class CadTool {
    public static void main(String[] args) {
        int result =
            CadTool.calculateResult(13, false, true, -1, null);
    }

    public static int calculateResult(final int baseValue,
                                      final boolean isMetric,
                                      final boolean is2D,
                                      final int offset,
                                      final Integer height) {
        return 0; // some calculation
    }
}
```

Too many method parameters

Bloaters Data Clumps

```
record Customer(
        String lastName,
        String firstName,
        String middleName,
        String salutation,
        String streetAddress,
        String city,
        String state,
        String country,
        boolean isEmployed,
        boolean isHomeOwner
) {
}
```



Grouping unrelated data

OOP Abusers

Missing object-oriented design possibilities.

OOP Abusers Switch Statements

```java
public class Vehicles {
    public int numberOfWheels(final String vehicle)
        throws Exception {

        return switch (vehicle) {
            case "car" -> 4;
            case "boat" -> 0;
            case "bike" -> 2;
            case "bicycle" -> 2;
            default -> throw new Exception("Unknown");
        };
    }
}
```

Branching out too much

OOP Abusers Temporary Field

```java
public class Rectangle {
    private float sideA;
    private float sideB;

    public void setSideA(float sideA) {
        this.sideA = sideA;
    }

    public void setSideB(float sideB) {
        this.sideB = sideB;
    }

    public double getAreaSize() {
        return sideA * sideB;
    }
}
```

Fields that are only used once

OOP Abusers Refused Bequest

```
public class Animals {
    interface Animal { void speak(); }

    static class Dog implements Animal {
        @Override
        public void speak() {
            System.out.println("Woof");
        }
    }

    static class Fish implements Animal {
        @Override
        public void speak() {
        }
    }
}
```

Passing unneeded behaviour to classes

OOP Abusers Different Interfaces

```
public class Shapes {
    record Circle(float radius) {
        public double getAreaSize() {
            return radius * radius * Math.PI;
        }
    }

    record Rectangle(float a, float b) {
        public double getSurfaceSize() {
            return a * b;
        }
    }
}
```

Confusing naming of similar functions

Change Preventers

Hindering further development.

Change Preventers Divergent Change

```java
public class AnimalInformation {
    public static String getLatinName(final String animal) {
        if (animal.equalsIgnoreCase("horse"))
            return "Equus caballus";
        return "";
    }

    public static int getNumberOfLegs(final String animal) {
        if (animal.equalsIgnoreCase("horse"))
            return 4;
        return -1;
    }
}
```

Changes across methods

Change Preventers Shotgun Surgery

```java
public class Classification {
    public static String getLatinName(final String animal) {
        if (animal.equalsIgnoreCase("horse"))
            return "Equus caballus";
        return "";
    }
}

public class BodyParts {
    public static int getNumberOfLegs(final String animal) {
        if (animal.equalsIgnoreCase("horse"))
            return 4;
        return -1;
    }
}
```

Changes across classes

Change Preventers Parallel Hierarchy

```java
public class Birds {
    private static class Bird {}
    private static class Egg {}

    private static class Sparrow extends Bird {
        private SparrowEgg layEgg() {
            return new SparrowEgg();
        }
    }

    private static class SparrowEgg extends Egg {
    }
}
```

Implementing parallel interfaces

Dispensables

Unnecessary things that can be removed.

Dispensables Lazy Class

```
public class WebShopCheckOut {
    public static void checkOut(final ShoppingCart cart) {
        // Some implementation
    }
}

record ShoppingCart(Product... products) {
}

record Product(String id) {
}
```

A class with a single method

Dispensables Data Class

```java
public class DataClass {
    record Rectangle(int sideA, int sideB) {
    }

    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(5, 20);
        int rectangleArea = rectangle.sideA * rectangle.sideB;
    }
}
```

Class holding data but not its own logic

Dispensables Duplicate Code

```java
public class Customers {
    private List<Customer> customers;

    public void addCustomer(Customer customer) {
        customers.add(customer);
        customers.forEach(System.out::println);
    }

    public void removeCustomer(Customer customer) {
        customers.remove(customer);
        customers.forEach(System.out::println);
    }
}
```

The same code multiple times

Dispensables Dead Code

```
public class DeadCode {
    public static void main(String[] args) {
        System.out.println("Pi = " + Math.PI);
    }

    private double getPi() {
        return 3.141592;
        System.out.println("Returning Pi");
    }
}
```

Code that cannot be reached

Dispensables Speculative Generality

```java
public class LaserPrinter implements Printer {
    @Override
    public void print(final String textToPrint) {
        // Implementation
    }

    @Override
    public void turnOnOrOff() {
        //Implementation
    }
}

interface Printer {
    void print(String textToPrint);
    void turnOnOrOff();
}
```

Premature future-proofing

Couplers

Too much or too little coupling.

Couplers Feature Envy

```java
public class UsersAndAddresses {
    record User(Address address) {
        public String getAddressString() {
            return
                address.street() + ", "
                + address.city() + ", "
                + address.country();
        }
    }

    record Address(String street, String city, String country) { }
}
```

Class implementing features of another

Couplers Inappropriate Intimacy

```java
public class Book {
    private final String title;
    private Author author;

    public Book(String title) { this.title = title;}
    public void setAuthor(Author author) { this.author = author; }
}

public class Author {
    private final String name;
    private Book book;

    public Author(String name) { this.name = name; }
    public void setBook(Book book) { this.book = book; }
}
```

Classes knowing each other too well

Couplers Message Chains

```
public class Storehouse {
    public static void main(String[] args) {

        List<Product> products = List.of(
            new Product("Cheese", "Tasty cheese"));
        List<Shelf> shelves = List.of(new Shelf(products));

        String description =
            shelves.get(0).products().get(0).description();

    }

    record Shelf(List<Product> products) { }
    record Product(String name, String description) { }
}
```

Returning objects that return objects…

Dispensables Middle Man

```
public record Customer(String name, Address address) {

    public String getCity() {
        return address.city();
    }

    public String getStreet() {
        return address.street();
    }

    record Address(String street, String city) { }
}
```

Class accessing another one on its behalf

Classification

Bloaters	OOP Abusers	Change Preventers	Dispensables	Couplers
Long Method Large Class Primitive Obsession Long Parameter List Data Clumps	Switch Statements Temporary Field Refused Bequest Alternative classes with different Interfaces	Divergent Change Shotgun Surgery Parallel Inheritance Hierarchy	Lazy Class Data Class Duplicate Code Dead Code Speculative Generality	Feature Envy Inappropriate Intimacy Message Chains Middleman

Example Code

github.com/bischoffdev/code-smells

When to tackle code smells?

Broken window

Long-term effects of unfixed code.

YAGNI

Wasting time with future requirements.

Campground Rule

Later never comes…

Code Reviews

Point out smells before it is too late!

Smelly Code

Smelly code, smelly code
How are they treating you?

Smelly code, smelly code
It's not your fault.

Thank you!

softwaretester.blog | Benjamin Bischoff