How Reusability Mechanisms and Built-in Functions Impact Code Quality Over Time

Curriculum: Internet of Things and Smart Technologies

Ph.D. Candidate: Giammaria Giordano

Advisors: Prof. Fabio Palomba, Prof. Filomena Ferrucci

University of Salerno, Italy
Department of Computer Science
Software Engineering (SeSa) Lab

giagiordano@unisa.it

@GiammariaGiord1

https://giammariagiordano.github.io/giammaria-giordano/

ARIANE 5: Flight 501 Failure

The Ariane 5 **reused** the code from the inertial reference platform from the Ariane 4, but the early part of the Ariane 5's flight path differed from the Ariane 4 in having higher horizontal velocity values. This caused an internal value BH (Horizontal Bias) calculated in the alignment function to be unexpectedly high.

https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html

Write Sign up Sign in

ARIANE 5: Flight 501 Failure

The Ariane 5 **reused** the code from the inertial reference platform from the Ariane 4, but the early part of the Ariane 5's flight path differed from the Ariane 4 in having higher horizontal velocity values. This caused an internal value BH (Horizontal Bias) calculated in the alignment function to be unexpectedly high.

On 4 June 1996, the first flight of the **Ariane 5** launcher **failed**, only about **40 seconds** after initiation of the flight sequence

https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html

$370 million

https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html

1,179,038

African children fed for a year

https://www.wfpusa.org/articles/how-much-would-it-cost-to-end-world-hunger/ - World Food Program USA

Write Sign up Sign in

ARIANE 5: Flight 501 Failure

The Ariane 5 **reused** the code from the inertial reference platform from the Ariane 4, but the early part of the Ariane 5's flight path differed from the Ariane 4 in having higher horizontal velocity values. This caused an internal value BH (Horizontal Bias) calculated in the alignment function to be unexpectedly high.

On 4 June 1996, the maiden flight of the **Ariane 5** launcher **failed**, only about **40 seconds** after initiation of the flight sequence

The analysis showed that the software was **prone** to **defects** by presenting **code quality issues**

https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html

Write Sign up Sign in

On 4 June 1996, the maiden flight of the **Ariane 5** launcher **failed**, only about **40 seconds** after

The analysis showed that the software was **prone** to **defects** by presenting **code quality issues**

The Ariane 5 **reused** the code from the inertial reference platform from the Ariane 4, but the early part of the Ariane 5's flight path differed from the Ariane 4 in having higher horizontal velocity values. This caused an internal value BH (Horizontal Bias) calculated in the alignment function to be unexpectedly high.

The analysis showed that the software was **prone** to **defects** by presenting **code quality issues**

https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html

Write Sign up Sign in

On 4 June 1996, the maiden flight of the **Ariane 5** launcher **failed**, only about **40 seconds** after

The analysis showed that the software was **prone** to **defects** by presenting **code quality issues**

The analysis showed that the software was **prone** to **defects** by presenting **code quality issues**

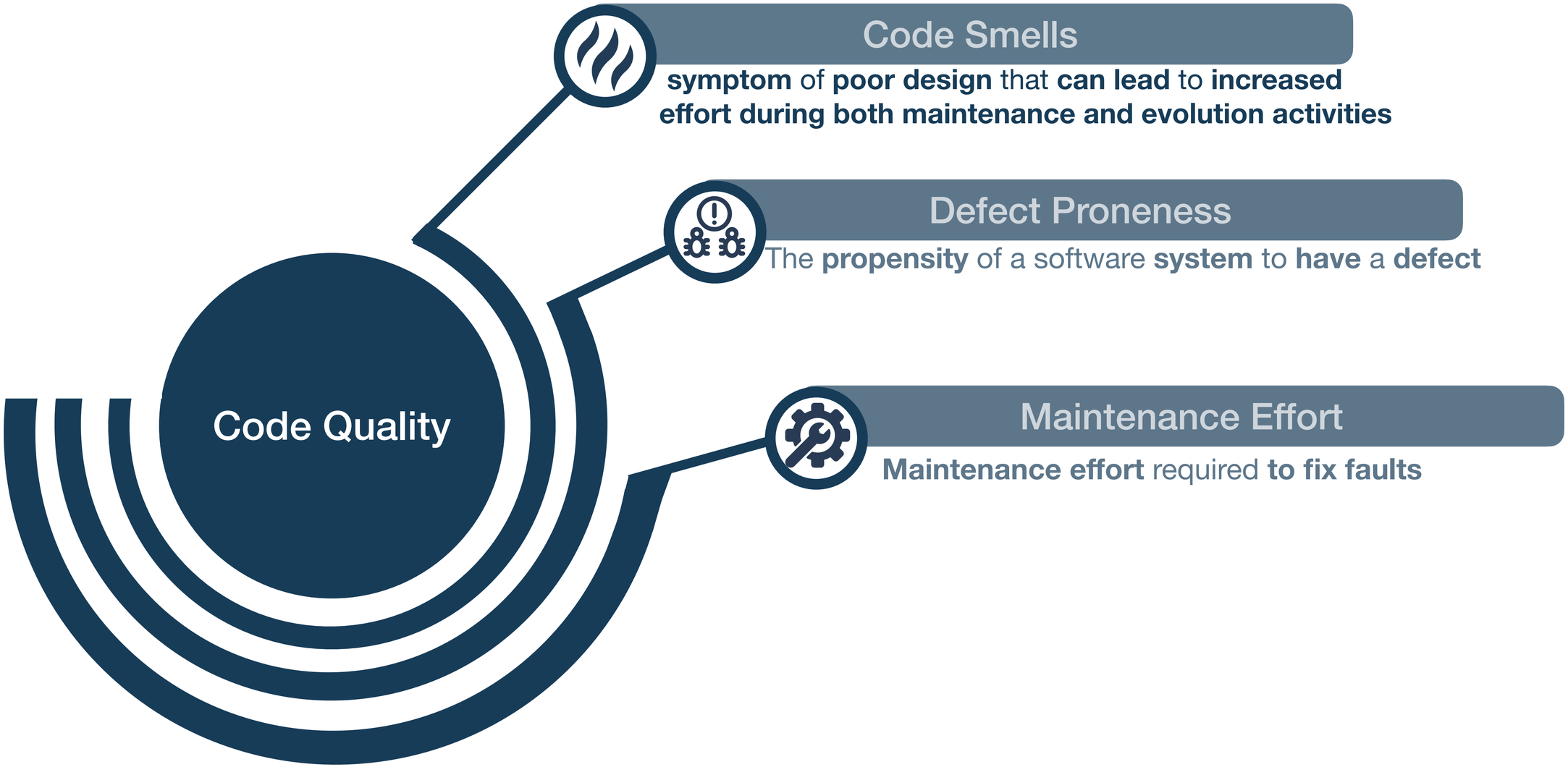
The company had **reused** code from the previous rocket but did not do the necessary **maintenance activities**

https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html

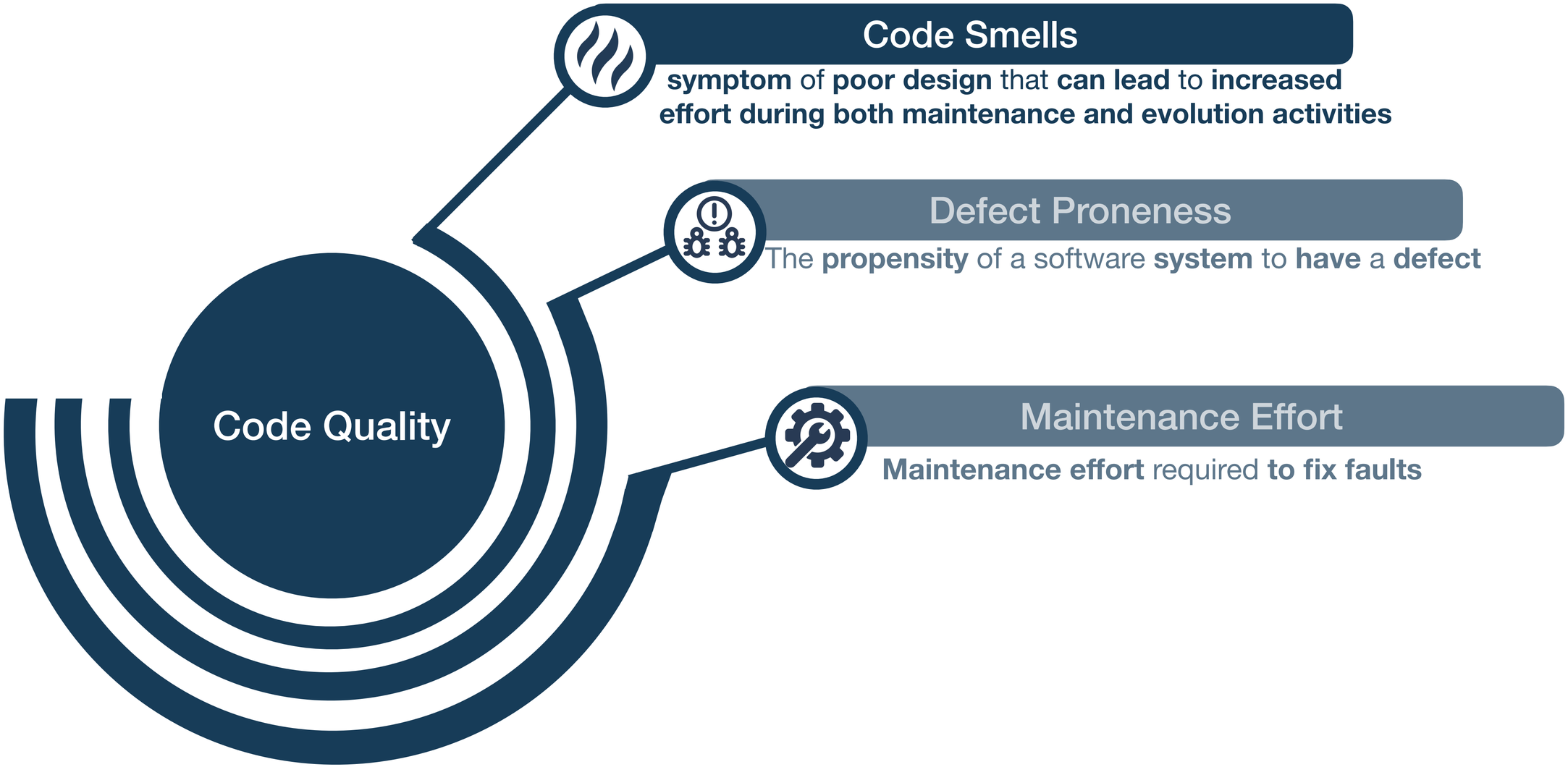
Could it have been prevented?

If the company had used tools to measure code quality variation over time, this accident probably would not have occurred

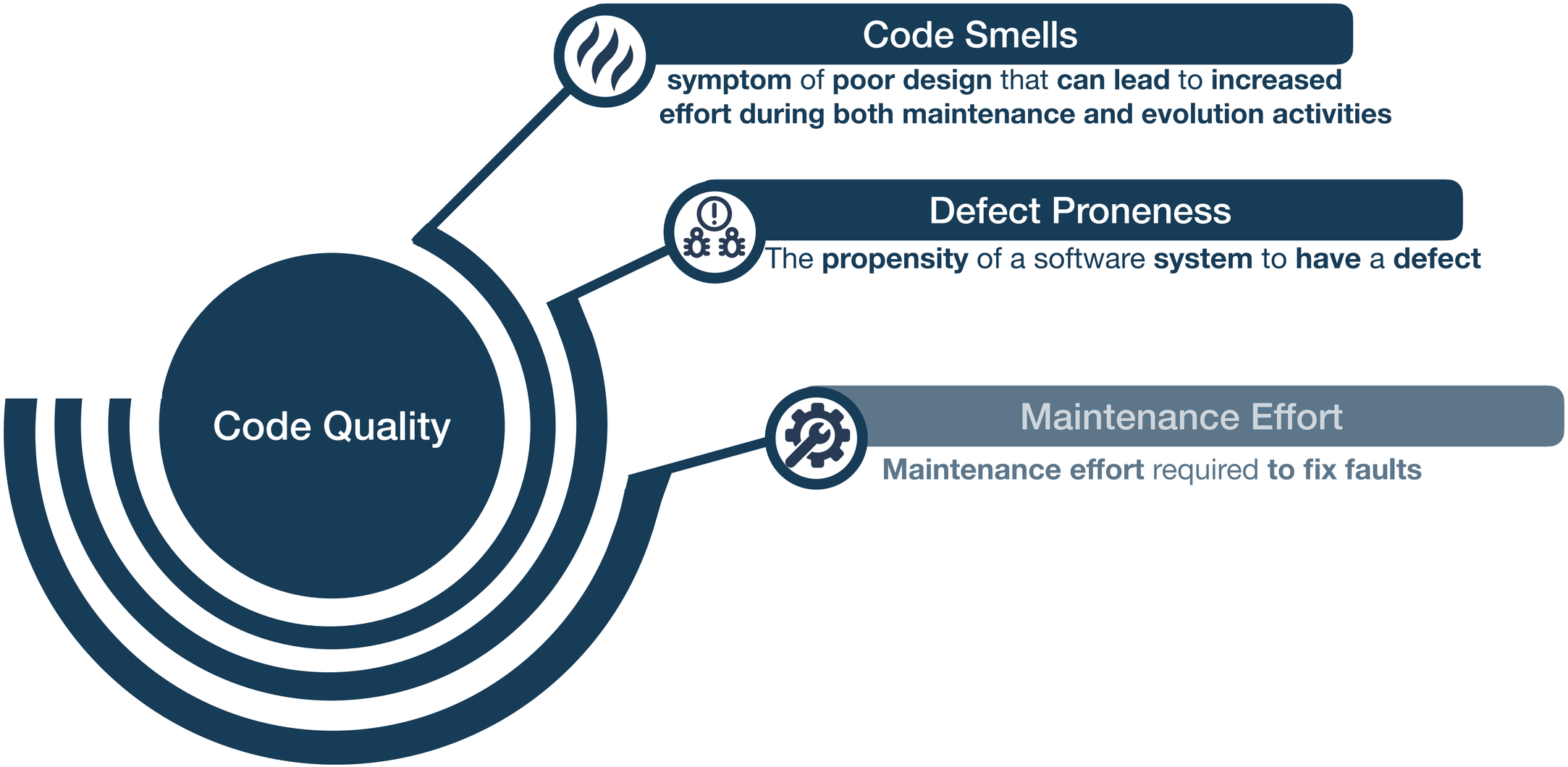
Code Quality Attributes



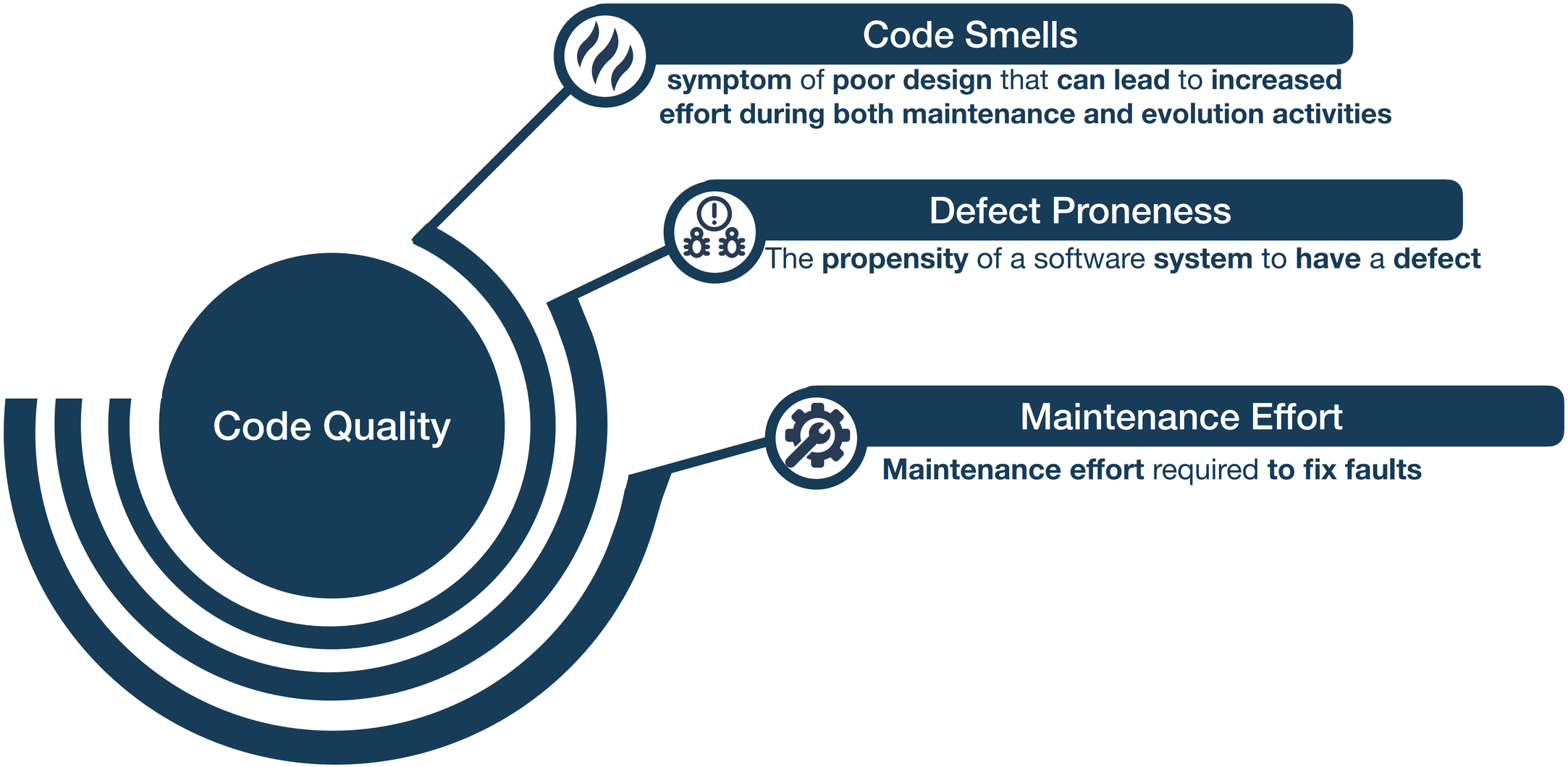
Code Quality Attributes



Code Quality Attributes



Code Quality Attributes



“Software design tends to erode over time because of continuous changes and increasing complexity”

“Evolution is an essential property of real-world software. As your needs change, your criteria for satisfaction change.”

Manny Lehman

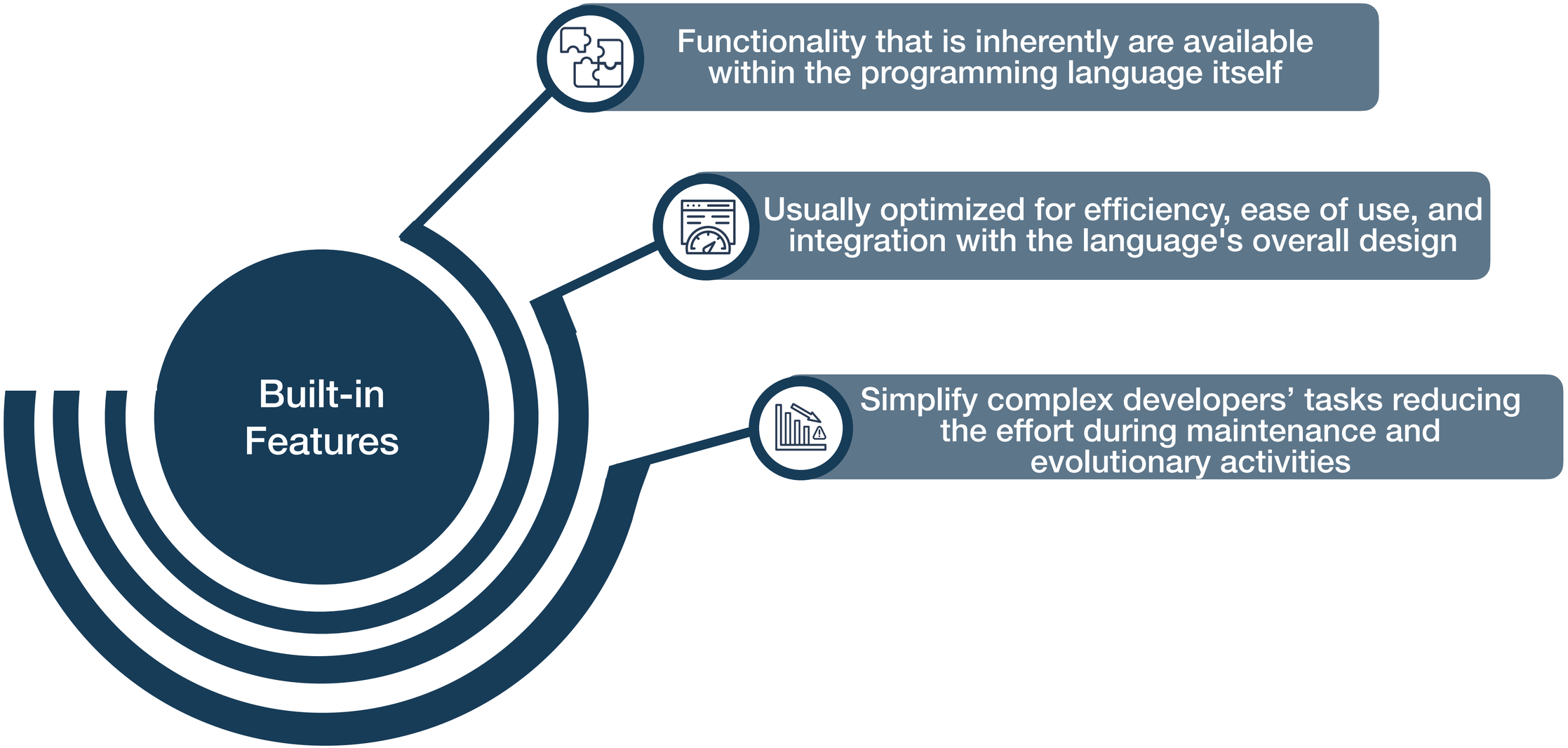
“Software design tends to erode over time because of continuous changes and increasing complexity”

“Evolution is an essential property of real-world software. As your needs change, your criteria for satisfaction change.”

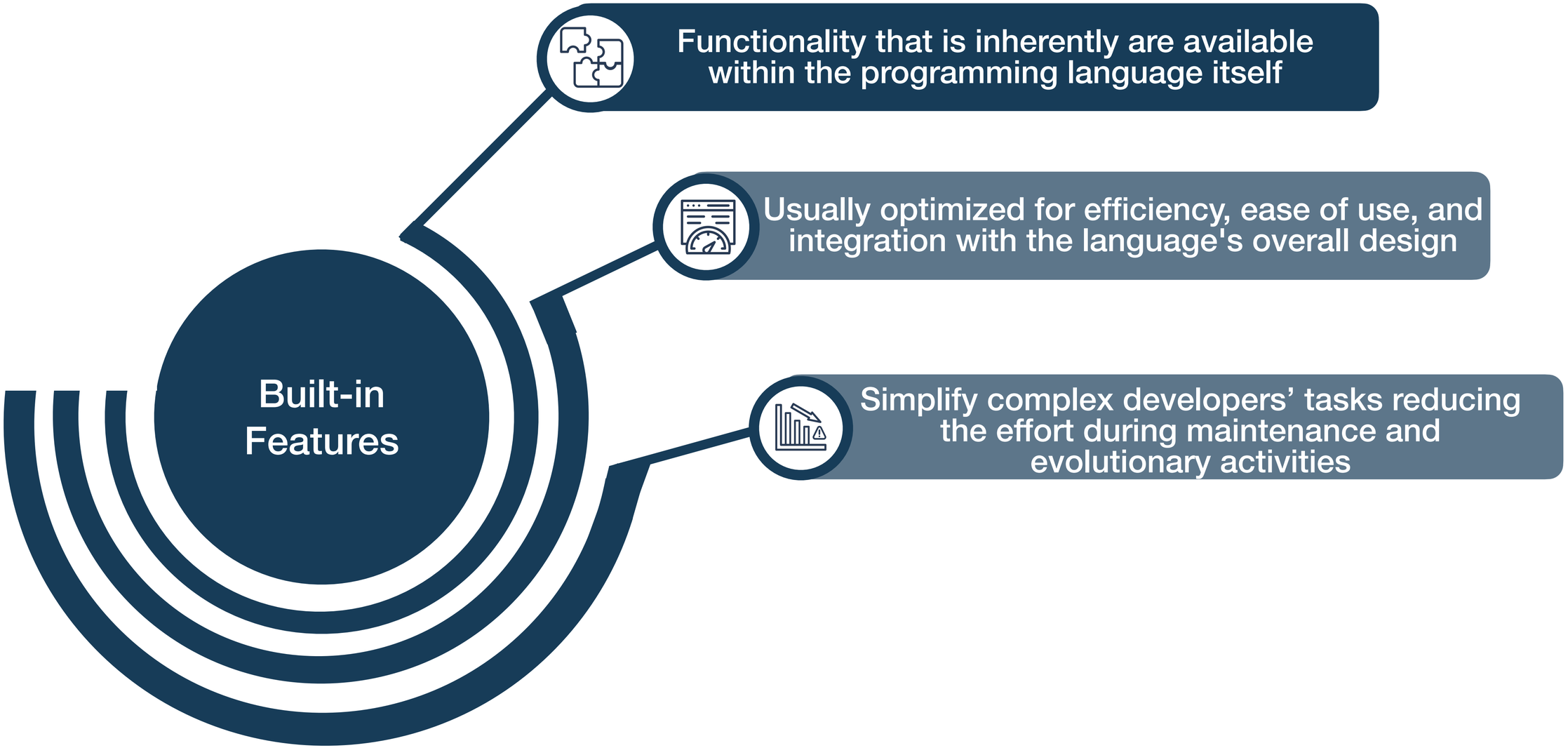
Manny Lehman

…But in which manners do developers perform maintenance and evolutionary activities?

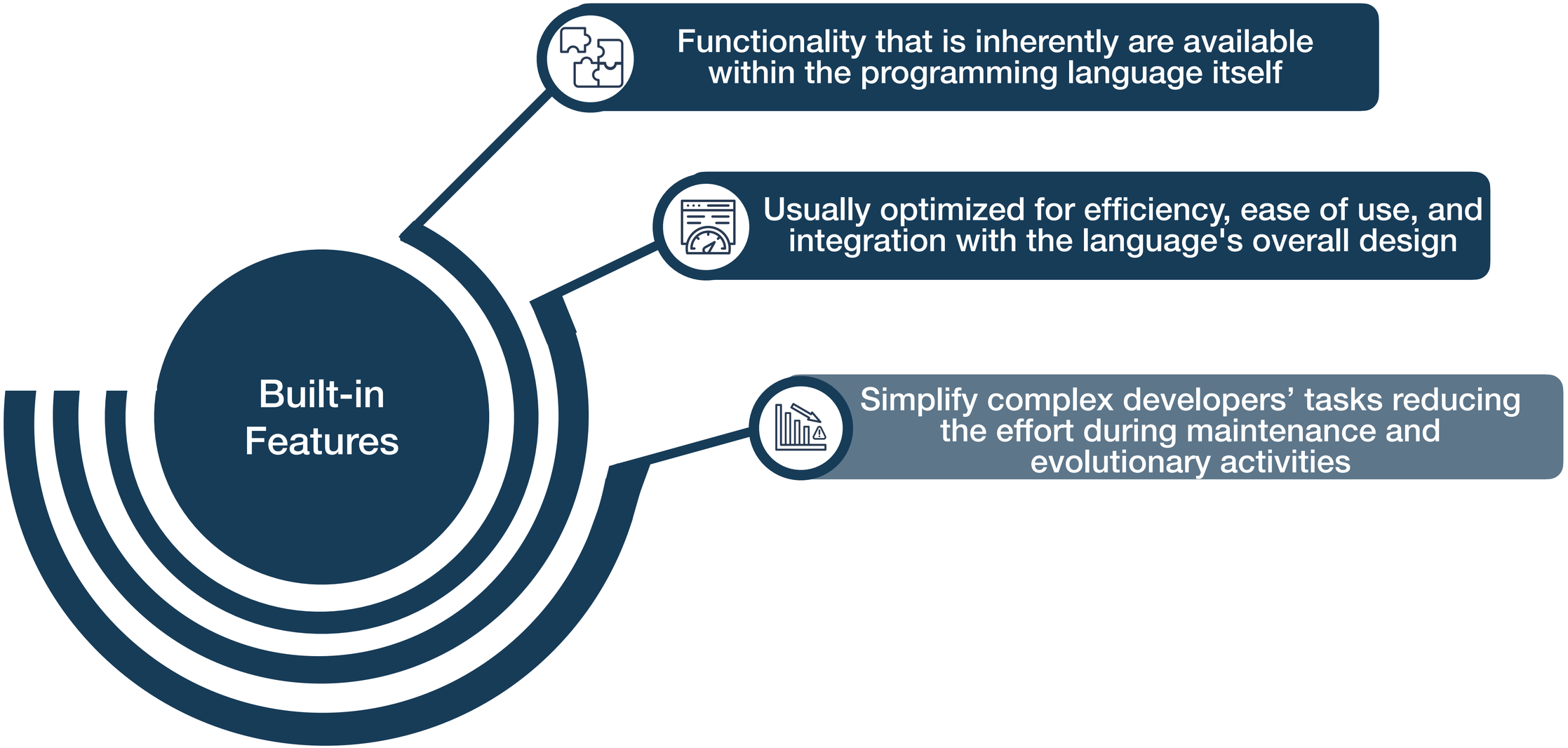
Built-in Features



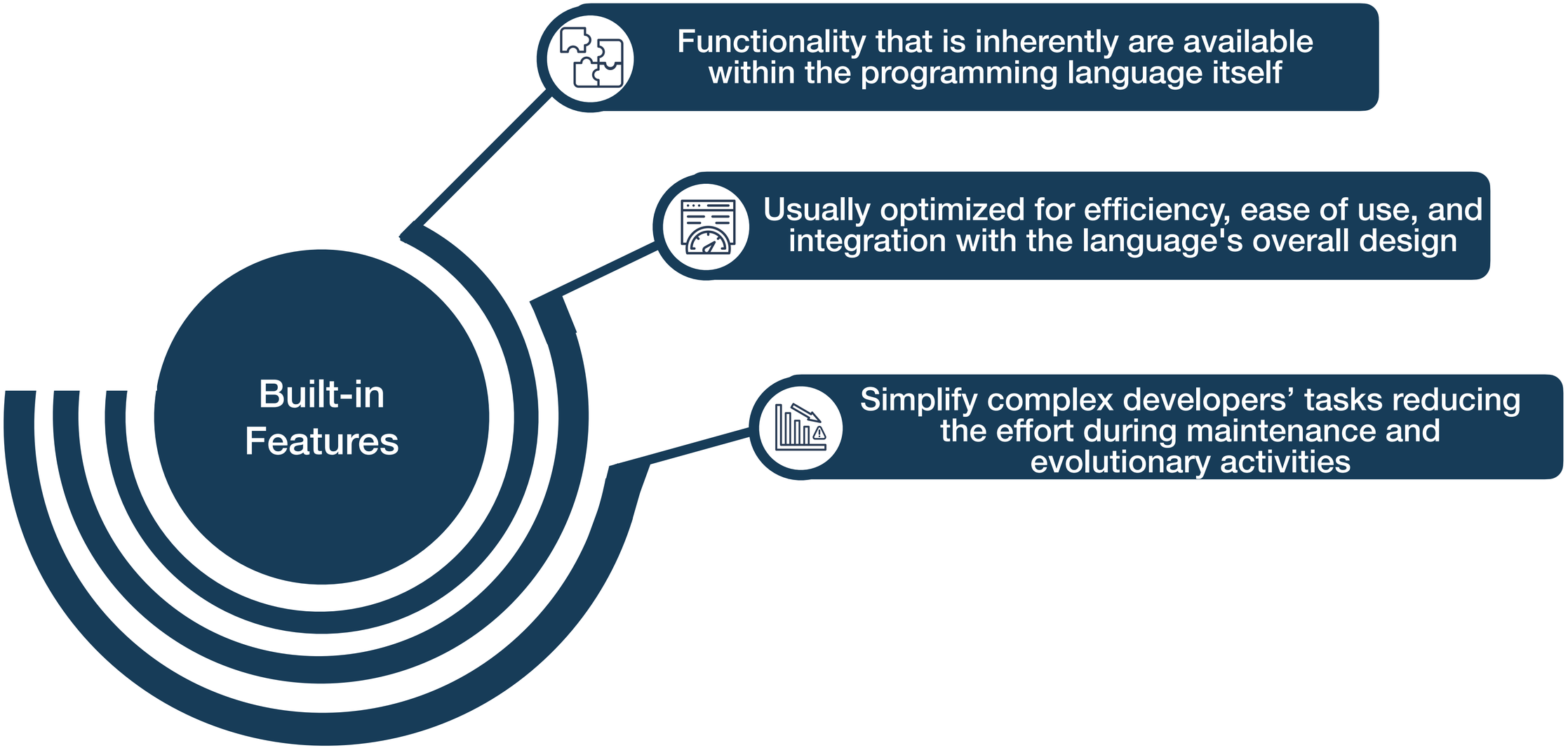
Built-in Features



Built-in Features



Built-in Features



Built-in Features

```
list(): Converts an iterable to a list

tuple(): Converts an iterable to a tuple

set(): Converts an iterable to a set

dict(): Creates a dictionary

all(): Returns True if all elements of an iterable are true

any(): Returns True if any element of an iterable is true

filter(): Filters elements from an iterable

map(): Applies a function to all items in an input iterable

reduce(): Applies a rolling computation to sequential pairs of values in an
iterable
```

Reusability Mechanisms

Plugins and Extensions

Modularization

Inheritance and Delegation

Design Patterns

Frameworks

Parameterization

Code Generation

Legacy System Wrapping

Service Oriented Systems

Reusability Mechanisms

Plugins and Extensions

Modularization

Inheritance and Delegation

Frameworks

Design Patterns

Parameterization

Code Generation

Legacy System Wrapping

Service Oriented Systems

Reusability Mechanisms

Plugins and Extensions

Modularization

Inheritance and Delegation

Frameworks

Design Patterns

Parameterization

Code Generation

Legacy System Wrapping

Service Oriented Systems

Code Quality: State-of-the-art

Toward a Smell-Aware Bug Prediction Model

Fabio Palomba, *Member, IEEE*, Marco Zanoni, Francesca Arcelli Fontana, *Member, IEEE*, Andrea De Lucia, *Senior Member, IEEE*, and Rocco Oliveto

Abstract—Code smells are symptoms of poor design and implementation choices. Previous studies empirically assessed the impact of smells on code quality and clearly indicate their negative impact on maintainability, including a higher bug-proneness of components affected by code smells. In this paper, we capture previous findings on bug-proneness to build a specialized bug prediction model for smelly classes. Specifically, we evaluate the contribution of a measure of the severity of code smells (i.e., code smell intensity) by adding it to existing bug prediction models based on both product and process metrics, and comparing the results of the new model against the baseline models. Results indicate that the accuracy of a bug prediction model increases by adding the code smell intensity as predictor. We also compare the results achieved by the proposed model with the ones of an alternative technique which considers metrics about the history of code smells in files, finding that our model works generally better. However, we observed interesting complementarities between the set of *buggy and smelly* classes correctly classified by the two models. By evaluating the actual information gain provided by the intensity index with respect to the other metrics in the model, we found that the intensity index is a relevant feature for both product and process metrics-based models. At the same time, the metric counting the average number of code smells in previous versions of a class considered by the alternative model is also able to reduce the entropy of the model. On the basis of this result, we devise and evaluate a *smell-aware* combined bug prediction model that included product, process, and smell-related features. We demonstrate how such model classifies bug-prone code components with an F-Measure at least 13 percent higher than the existing state-of-the-art models.

Index Terms—Code smells, bug prediction, empirical study, mining software repositories

A Developer Centered Bug Prediction Model

Dario Di Nucci, *Student Member, IEEE*, Fabio Palomba, *Member, IEEE*, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia, *Senior Member, IEEE*

Abstract—Several techniques have been proposed to accurately predict software defects. These techniques generally exploit characteristics of the code artefacts (e.g., size, complexity, etc.) and/or of the process adopted during their development and maintenance (e.g., the number of developers working on a component) to spot out components likely containing bugs. While these bug prediction models achieve good levels of accuracy, they mostly ignore the major role played by human-related factors in the introduction of bugs. Previous studies have demonstrated that focused developers are less prone to introduce defects than non-focused developers. According to this observation, software components changed by focused developers should also be less error prone than components changed by less focused developers. We capture this observation by measuring the scattering of changes performed by developers working on a component and use this information to build a bug prediction model. Such a model has been evaluated on 26 systems and compared with four competitive techniques. The achieved results show the superiority of our model, and its high complementarity with respect to predictors commonly used in the literature. Based on this result, we also show the results of a "hybrid" prediction model combining our predictors with the existing ones.

Index Terms—Scattering metrics, bug prediction, empirical study, mining software repositories

1 Introduction

Bug prediction techniques are used to identify areas of software systems that are more likely to contain bugs. These prediction models represent an important aid when the resources available for testing are scarce, since they can indicate *where* to invest such resources. The scientific community has developed several bug prediction models that can be roughly classified into two families, based on the information they exploit to discriminate between "buggy" and "clean" code components. The first set of techniques exploits *product metrics* (i.e., metrics capturing intrinsic characteristics of the code components, like their size and complexity) [1], [2], [3], [4], [5], while the second one focuses on *process metrics* (i.e., metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) [6], [7], [8], [9], [10], [11], [12]. While some studies highlighted the superiority of these latter with respect to the *product metric based* techniques [7], [11], [13] there is a general consensus on the fact that no technique is the best in all contexts [14], [15]. For this reason, the research community is still spending effort in investigating under which circumstances and during which coding activities developers tend to introduce bugs (see, e.g., [16], [17], [18], [19], [20], [21], [22]).

Some of these studies have highlighted the central role played by developer-related factors in the introduction of bugs.

In particular, Eyolfson et al. [17] showed that more experienced developers tend to introduce less faults in software systems. Rahman and Devanbu [18] partly contradicted the study by Eyolfson et al. by showing that the experience of a developer has no clear link with the bug introduction. Bird et al. [20] found that high levels of ownership are associated with fewer bugs. Finally, Posnett et al. [22] showed that focused developers (i.e., developers focusing their attention on a specific part of the system) introduce fewer bugs than unfocused developers.

Although such studies showed the potential of human-related factors in bug prediction, this information is not captured in state-of-the-art bug prediction models based on process metrics extracted from version history. Indeed, previous bug prediction models exploit predictors based on (i)

Empirical validation of object-oriented metrics for predicting fault proneness models

Yogesh Singh · Arvinder Kaur · Ruchika Malhotra

Published online: 1 July 2009

Abstract Empirical validation of software metrics used to predict software quality attributes is important to ensure their practical relevance in software organizations. The aim of this work is to find the relation of object-oriented (OO) metrics with fault proneness

A Validation of Object-Oriented Design Metrics as Quality Indicators

Victor R. Basili, *Fellow, IEEE*, Lionel C. Briand, and Walcélio L. Melo, *Member, IEEE Computer Society*

Abstract—This paper presents the results of a study in which we empirically investigated the suite of object-oriented (OO) design metrics introduced in [13]. More specifically, our goal is to assess these metrics as predictors of fault-prone classes and, therefore, determine whether they can be used as early quality indicators. This study is complementary to the work described in [30] where the same suite of metrics had been used to assess frequencies of maintenance changes to classes. To perform our validation accurately, we collected data on the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method and the C++ programming language. Based on empirical and quantitative analysis, the advantages and drawbacks of these OO metrics are discussed. Several of Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle. Also, on our data set, they are better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes.

Index Terms—Object-oriented design metrics, error prediction model, object-oriented software development, C++ programming language.

1 Introduction

1.1 Motivation

The development of a large software system is a time- and resource-consuming activity. Even with the increasing automation of software development activities, resources are still scarce. Therefore, we need to be able to provide accurate information and guidelines to managers to help them make decisions, plan and schedule activities, and allocate resources for the different software activities that take place during software development. Software metrics are, thus, necessary to identify where the resources are needed; they are a crucial source of information for decision-making [22].

Testing of large systems is an example of a resource- and time-consuming activity. Applying equal testing and verification effort to all parts of a software system has become cost-prohibitive. Therefore, one needs to be able to identify fault-prone modules so that testing/verification effort can be concentrated on these modules [21]. The availability of adequate product design metrics for characterizing error-

Many product metrics have been proposed [16], [26], used, and, sometimes, empirically validated [3], [4], [19], [30], e.g., number of lines of code, McCabe complexity metric, etc. In fact, many companies have built their own cost, quality, and resource prediction models based on product metrics. TRW [7], the Software Engineering Laboratory (SEL) [31], and Hewlett Packard [20] are examples of software organizations that have been using product metrics to build their cost, resource, defect, and productivity models.

1.2 Issues

In the last decade, many companies have started to introduce object-oriented (OO) technology into their software development environments. OO analysis/design methods, OO languages, and OO development environments are currently popular worldwide in both small and large software organizations. The insertion of OO technology in the software industry, however, has created new challenges for

Predicting Fault-Proneness using OO Metrics An Industrial Case Study

Ping Yu
Network Service Management, Alcatel Canada Inc,
400-4190 Still Creek Dr.
Burnaby, BC, V5C 6C6, Canada
p.yu@alcatel.com

Tarja Systä
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
tsysta@cs.tut.fi

Hausi Müller
Department of Computer Science
University of Victoria
P.O. Box 3055, Victoria, BC, V8W 3P6, Canada
hausi@csr.uvic.ca

Abstract

Software quality is an important external software at-

project managers in decision making. In software forward engineering, software metrics are traditionally used to revise an improper design in an early phase of the software

Empirical Software Engineering (2020) 25:49–95
https://doi.org/10.1007/s10664-019-09739-0

Improving change prediction models with code smell-related information

Gemma Catolino[1] · Fabio Palomba[2] · Francesca Arcelli Fontana[3] · Andrea De Lucia[1] · Andy Zaidman[4] · Filomena Ferrucci[1]

Published online: 2 August 2019

Abstract
Code smells are sub-optimal implementation choices applied by developers that have the effect of negatively impacting, among others, the change-proneness of the affected classes. Based on this consideration, in this paper we conjecture that code smell-related information can be effectively exploited to improve the performance of change prediction models, i.e., models having the goal of indicating which classes are more likely to change in the future. We exploit the so-called *intensity index*—a previously defined metric that captures the severity of a code smell—and evaluate its contribution when added as additional feature in the context of three state of the art change prediction models based on product, process, and developer-based features. We also compare the performance achieved by the proposed model with a model based on previously defined antipattern metrics, a set of indicators computed considering the history of code smells in files. Our results report that (i) the prediction performance of the intensity-including models is statistically better than the baselines and, (ii) the intensity is a better predictor than antipattern metrics. We observed some orthogonality between the set of change-prone and non-change-prone classes correctly classified by the models relying on intensity and antipattern metrics: for this reason, we also devise and evaluate a smell-aware combined change prediction model including product, process, developer-based, and smell-related features. We show that the F-Measure of this model is notably higher than other models.

Keywords Change prediction · Code smells · Empirical study

Code Quality: State-of-the-art

Toward a Smell-Aware Bug Prediction Model

Fabio Palomba, *Member, IEEE*, Marco Zanoni, Francesca Arcelli Fontana, *Member, IEEE*, Andrea De Lucia, *Senior Member, IEEE*, and Rocco Oliveto

Abstract—Code smells are symptoms of poor design and implementation choices. Previous studies empirically assessed the impact of smells on code quality and clearly indicate their negative impact on maintainability, including a higher bug-proneness of components

A Developer Centered Bug Prediction Model

Dario Di Nucci, *Student Member, IEEE*, Fabio Palomba, *Member, IEEE*, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia, *Senior Member, IEEE*

Abstract—Several techniques have been proposed to accurately predict software defects. These techniques generally exploit characteristics of the code artefacts (e.g., size, complexity, etc.) and/or of the process adopted during their development and maintenance (e.g., the number of developers working on a component) to spot out components likely containing bugs. While these bug prediction models achieve good levels of accuracy, they mostly ignore the major role played by human-related factors in the introduction of bugs. Previous studies have demonstrated that focused developers are less prone to introduce defects than non-focused developers. According to this observation, software components changed by focused developers should also be less error prone than components changed by less focused developers. We capture this observation by measuring the scattering of changes performed by developers working on a component and use this information to build a bug prediction model. Such a model has been evaluated on 26 systems and compared with four competitive techniques. The achieved results show the superiority of our model, and its high complementarity with respect to predictors commonly used in the literature. Based on this result, we also show the results of a "hybrid" prediction model combining our predictors with the existing ones.

Index Terms—Scattering metrics, bug prediction, empirical study, mining software repositories

1 Introduction

Bug prediction techniques are used to identify areas of software systems that are more likely to contain bugs. These prediction models represent an important aid when the resources available for testing are scarce, since they can indicate *where* to invest such resources. The scientific community has developed several bug prediction models that can be roughly classified into two families, based on the information they exploit to discriminate between "buggy" and "clean" code components. The first set of techniques exploits *product metrics* (i.e., metrics capturing intrinsic characteristics of the code components, like their size and complexity) [1], [2], [3], [4], [5], while the second one focuses on *process metrics* (i.e., metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) [6], [7], [8], [9], [10], [11], [12]. While some studies highlighted the superiority of these latter with respect to the *product metric based* techniques [7], [11], [13] there is a general consensus on the fact that no technique is the best in all contexts [14], [15]. For this reason, the research community is still spending effort in investigating under which circumstances and during which coding activities developers tend to introduce bugs (see, e.g., [16], [17], [18], [19], [20], [21], [22]).

Some of these studies have highlighted the central role played by developer-related factors in the introduction of bugs.

In particular, Eyolfson et al. [17] showed that more experienced developers tend to introduce less faults in software systems. Rahman and Devanbu [18] partly contradicted the study by Eyolfson et al. by showing that the experience of a developer has no clear link with the bug introduction. Bird et al. [20] found that high levels of ownership are associated with fewer bugs. Finally, Posnett et al. [22] showed that focused developers (i.e., developers focusing their attention on a specific part of the system) introduce fewer bugs than unfocused developers.

Although such studies showed the potential of human-related factors in bug prediction, this information is not captured in state-of-the-art bug prediction models based on process metrics extracted from version history. Indeed, previous bug prediction models exploit predictors based on (i) the number of developers working on a code component [9], [10]; (ii) the analysis of change-proneness [11], [12], [13]; and (iii) the entropy of changes [8]. Thus, despite the previ-

• D. Di Nucci, G. De Rosa, and A. De Lucia are with the University of Salerno, Fisciano, SA 84084, Italy.

Empirical validation of object-oriented metrics for predicting fault proneness models

Yogesh Singh · Arvinder Kaur · Ruchika Malhotra

July 2009
+Business Media, LLC 2009

rical validation of software metrics used to predict software quality ortant to ensure their practical relevance in software organizations. The is to find the relation of object-oriented (OO) metrics with fault proneness

A Validation of Object-Oriented Design Metrics as Quality Indicators

Victor R. Basili, *Fellow, IEEE*, Lionel C. Briand, and Walcélio L. Melo, *Member, IEEE Computer Society*

t—This paper presents the results of a study in which we empirically investigated the suite of object-oriented (OO) design ntroduced in [13]. More specifically, our goal is to assess these metrics as predictors of fault-prone classes and, therefore, e whether they can be used as early quality indicators. This study is complementary to the work described in [30] where the ite of metrics had been used to assess frequencies of maintenance changes to classes. To perform our validation ly, we collected data on the development of eight medium-sized information management systems based on identical ents. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method and programming language. Based on empirical and quantitative analysis, the advantages and drawbacks of these OO metrics ssed. Several of Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the ses of the life-cycle. Also, on our data set, they are better predictors than "traditional" code metrics, which can only be at a later phase of the software development processes.

erms—Object-oriented design metrics, error prediction model, object-oriented software development, C++ programming .

UCTION

ion

lopment of a large software system is a time-ource-consuming activity. Even with the in-omation of software development activities, still scarce. Therefore, we need to be able to rate information and guidelines to managers make decisions, plan and schedule activities, resources for the different software activities ace during software development. Software thus, necessary to identify where the resources hey are a crucial source of information for de-g [22].

large systems is an example of a resource- and ing activity. Applying equal testing and verifi-to all parts of a software system has become ive. Therefore, one needs to be able to identify modules so that testing/verification effort can ted on these modules [21]. The availability of oduct design metrics for characterizing error-

Many product metrics have been proposed [16], [26], used, and, sometimes, empirically validated [3], [4], [19], [30], e.g., number of lines of code, McCabe complexity metric, etc. In fact, many companies have built their own cost, quality, and resource prediction models based on product metrics. TRW [7], the Software Engineering Laboratory (SEL) [31], and Hewlett Packard [20] are examples of software organizations that have been using product metrics to build their cost, resource, defect, and productivity models.

1.2 Issues

In the last decade, many companies have started to introduce object-oriented (OO) technology into their software development environments. OO analysis/design methods, OO languages, and OO development environments are currently popular worldwide in both small and large software organizations. The insertion of OO technology in the software industry, however, has created new challenges for

Predicting Fault-Proneness using OO Metrics An Industrial Case Study

Ping Yu
Network Service Management, Alcatel Canada Inc,
400-4190 Still Creek Dr.
Burnaby, BC, V5C 6C6, Canada
p.yu@alcatel.com

Tarja Systä
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
tsysta@cs.tut.fi

Hausi Müller
Department of Computer Science
University of Victoria
P.O. Box 3055, Victoria, BC, V8W 3P6, Canada
hausi@csr.uvic.ca

Abstract

Software quality is an important external software at-

project managers in decision making. In software forward engineering, software metrics are traditionally used to revise an improper design in an early phase of the software

Empirical Software Engineering (2020) 25:49–95
https://doi.org/10.1007/s10664-019-09739-0

Check for updates

Improving change prediction models with code smell-related information

Gemma Catolino[1] · Fabio Palomba[2] · Francesca Arcelli Fontana[3] · Andrea De Lucia[1] · Andy Zaidman[4] · Filomena Ferrucci[1]

Published online: 2 August 2019

Abstract
Code smells are sub-optimal implementation choices applied by developers that have the effect of negatively impacting, among others, the change-proneness of the affected classes. Based on this consideration, in this paper we conjecture that code smell-related information can be effectively exploited to improve the performance of change prediction models, i.e., models having the goal of indicating which classes are more likely to change in the future. We exploit the so-called *intensity index*—a previously defined metric that captures the severity of a code smell—and evaluate its contribution when added as additional feature in the context of three state of the art change prediction models based on product, process, and developer-based features. We also compare the performance achieved by the proposed model with a model based on previously defined antipattern metrics, a set of indicators computed considering the history of code smells in files. Our results report that (i) the prediction performance of the intensity-including models is statistically better than the baselines and, (ii) the intensity is a better predictor than antipattern metrics. We observed some orthogonality between the set of change-prone and non-change-prone classes correctly classified by the models relying on intensity and antipattern metrics: for this reason, we also devise and evaluate a smell-aware combined change prediction model including product, process, developer-based, and smell-related features. We show that the F-Measure of this model is notably higher than other models.

Keywords Change prediction · Code smells · Empirical study

Code Quality: State-of-the-art

Toward a Smell-Aware Bug Prediction Model

Fabio Palomba, *Member, IEEE*, Marco Zanoni, Francesca Arcelli Fontana, *Member, IEEE*, Andrea De Lucia, *Senior Member, IEEE*, and Rocco Oliveto

Abstract—Code smells are symptoms of poor design and implementation choices. Previous studies empirically assessed the impact of smells on code quality and clearly indicate their negative impact on maintainability, including a higher bug-proneness of components affected by code smells. In this paper, we capture previous findings on bug-proneness to build a specialized bug prediction model for smelly classes. Specifically, we evaluate the contribution of a measure of the severity of code smells (i.e., code smell intensity) by adding it to existing bug prediction models based on both product and process metrics, and comparing the results of the new model against the baseline models. Results indicate that the accuracy of a bug prediction model increases by adding the code smell intensity as predictor. We also compare the results achieved by the proposed model with the ones of an alternative technique which considers metrics about the history of code smells in files, finding that our model works generally better. However, we observed interesting complementarities between the set of *buggy and smelly* classes correctly classified by the two models. By evaluating the actual information gain provided by the intensity index with respect to the other metrics in the model, we found that the intensity index is a relevant feature for both product and process metrics-based models. At the same time, the metric counting the average number of code smells in previous versions of a class considered by the alternative model is also able to reduce the entropy of the model. On the basis of this result, we devise and evaluate a *smell-aware* combined bug prediction model that included product, process, and smell-related features. We demonstrate how such model classifies bug-prone code components with an F-Measure at least 13 percent higher than the existing state-of-the-art models.

Index Terms—Code smells, bug prediction, empirical study, mining software repositories

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 1, JANUARY 2018

A Developer Centered Bug Prediction Model

Dario Di Nucci, *Student Member, IEEE*, Fabio Palomba, *Member, IEEE*, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia, *Senior Member, IEEE*

Abstract—Several techniques have been proposed to accurately predict software defects. These techniques generally exploit characteristics of the code artefacts (e.g., size, complexity, etc.) and/or of the process adopted during their development and maintenance (e.g., the number of developers working on a component) to spot out components likely containing bugs. While these bug prediction models achieve good levels of accuracy, they mostly ignore the major role played by human-related factors in the introduction of bugs. Previous studies have demonstrated that focused developers are less prone to introduce defects than non-focused developers. According to this observation, software components changed by focused developers should also be less error prone than components changed by less focused developers. We capture this observation by measuring the scattering of changes performed by developers working on a component and use this information to build a bug prediction model. Such a model has been evaluated on 26 systems and compared with four competitive techniques. The achieved results show the superiority of our model, and its high complementarity with respect to predictors commonly used in the literature. Based on this result, we also show the results of a "hybrid" prediction model combining our predictors with the existing ones.

Index Terms—Scattering metrics, bug prediction, empirical study, mining software repositories

1 Introduction

Bug prediction techniques are used to identify areas of software systems that are more likely to contain bugs. These prediction models represent an important aid when the resources available for testing are scarce, since they can indicate *where* to invest such resources. The scientific community has developed several bug prediction models that can be roughly classified into two families, based on the information they exploit to discriminate between "buggy" and "clean" code components. The first set of techniques exploits *product metrics* (i.e., metrics capturing intrinsic characteristics of the code components, like their size and complexity) [1], [2], [3], [4], [5], while the second one focuses on *process metrics* (i.e., metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) [6], [7], [8], [9], [10], [11], [12]. While some studies highlighted the superiority of these latter with respect to the *product metric based* techniques [7], [11], [13] there is a general consensus on the fact that no technique is the best in all contexts [14], [15]. For this reason, the research community is still spending effort in investigating under which circumstances and during which coding activities developers tend to introduce bu (see, e.g., [16], [17], [18], [19], [20], [21], [22]).

Some of these studies have highlighted the central ro played by developer-related factors in the introduction bugs.

In particular, Eyolfson et al. [17] showed that more exp rienced developers tend to introduce less faults in softwa systems. Rahman and Devanbu [18] partly contradicted t study by Eyolfson et al. by showing that the experience o developer has no clear link with the bug introduction. Bi et al. [20] found that high levels of ownership are associate with fewer bugs. Finally, Posnett et al. [22] showed th focused developers (i.e., developers focusing their attenti on a specific part of the system) introduce fewer bugs tha unfocused developers.

Although such studies showed the potential of huma related factors in bug prediction, this information is not ca tured in state-of-the-art bug prediction models based process metrics extracted from version history. Indeed, pr vious bug prediction models exploit predictors based on

Empirical validation of object-oriented metrics for predicting fault proneness models

Yogesh Singh · Arvinder Kaur · Ruchika Malhotra

A Validation of Object-Oriented Design Metrics as Quality Indicators

Victor R. Basili, *Fellow, IEEE*, Lionel C. Briand, and Walcélio L. Melo, *Member, IEEE Computer Society*

Abstract—This paper presents the results of a study in which we empirically investigated the suite of object-oriented (OO) design metrics introduced in [13]. More specifically, our goal is to assess these metrics as predictors of fault-prone classes and, therefore, determine whether they can be used as early quality indicators. This study is complementary to the work described in [30] where the same suite of metrics had been used to assess frequencies of maintenance changes to classes. To perform our validation accurately, we collected data on the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method and the C++ programming language. Based on empirical and quantitative analysis, the advantages and drawbacks of these OO metrics are discussed. Several of Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle. Also, on our data set, they are better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes.

Index Terms—Object-oriented design metrics, error prediction model, object-oriented software development, C++ programming language.

1 Introduction

1.1 Motivation

The development of a large software system is a time- and resource-consuming activity. Even with the increasing automation of software development activities, resources are still scarce. Therefore, we need to be able to provide accurate information and guidelines to managers to help them make decisions, plan and schedule activities, and allocate resources for the different software activities that take place during software development. Software metrics are, thus, necessary to identify where the resources are needed; they are a crucial source of information for decision-making [22].

Testing of large systems is an example of a resource- and time-consuming activity. Applying equal testing and verification effort to all parts of a software system has become cost-prohibitive. Therefore, one needs to be able to identify fault-prone modules so that testing/verification effort can be concentrated on these modules [21]. The availability of adequate product design metrics for characterizing error-prone modules is, thus, vital.

Many product metrics have been proposed [16], [26], used, and, sometimes, empirically validated [3], [4], [19], [30], e.g., number of lines of code, McCabe complexity metric, etc. In fact, many companies have built their own cost, quality, and resource prediction models based on product metrics. TRW [7], the Software Engineering Laboratory (SEL) [31], and Hewlett Packard [20] are examples of software organizations that have been using product metrics to build their cost, resource, defect, and productivity models.

1.2 Issues

In the last decade, many companies have started to introduce object-oriented (OO) technology into their software development environments. OO analysis/design methods, OO languages, and OO development environments are currently popular worldwide in both small and large software organizations. The insertion of OO technology in the software industry, however, has created new challenges for companies which use product metrics as a tool for monitoring, controlling, and improving the way they develop and maintain software. Therefore, metrics which reflect the specificities of the OO paradigm must be defined and vali-

Predicting Fault-Proneness using OO Metrics An Industrial Case Study

Ping Yu
Network Service Management, Alcatel Canada Inc,
400-4190 Still Creek Dr.
Burnaby, BC, V5C 6C6, Canada
p.yu@alcatel.com

Tarja Systä
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
tsysta@cs.tut.fi

Hausi Müller
Department of Computer Science
University of Victoria
P.O. Box 3055, Victoria, BC, V8W 3P6, Canada
hausi@csr.uvic.ca

Abstract

re quality is an important external software at-

project managers in decision making. In software forward engineering, software metrics are traditionally used to revise an improper design in an early phase of the software

Empirical Software Engineering (2020) 25:49–95
https://doi.org/10.1007/s10664-019-09739-0

Improving change prediction models with code smell-related information

Gemma Catolino[1] · Fabio Palomba[2] · Francesca Arcelli Fontana[3] · Andrea De Lucia[1] · Andy Zaidman[4] · Filomena Ferrucci[1]

Published online: 2 August 2019

Abstract
Code smells are sub-optimal implementation choices applied by developers that have the effect of negatively impacting, among others, the change-proneness of the affected classes. Based on this consideration, in this paper we conjecture that code smell-related information can be effectively exploited to improve the performance of change prediction models, i.e., models having the goal of indicating which classes are more likely to change in the future. We exploit the so-called *intensity index*—a previously defined metric that captures the severity of a code smell—and evaluate its contribution when added as additional feature in the context of three state of the art change prediction models based on product, process, and developer-based features. We also compare the performance achieved by the proposed model with a model based on previously defined antipattern metrics, a set of indicators computed considering the history of code smells in files. Our results report that (i) the prediction performance of the intensity-including models is statistically better than the baselines and, (ii) the intensity is a better predictor than antipattern metrics. We observed some orthogonality between the set of change-prone and non-change-prone classes correctly classified by the models relying on intensity and antipattern metrics: for this reason, we also devise and evaluate a smell-aware combined change prediction model including product, process, developer-based, and smell-related features. We show that the F-Measure of this model is notably higher than other models.

Keywords Change prediction · Code smells · Empirical study

Code Quality: State-of-the-art

Toward a Smell-Aware Bug Prediction Model

Fabio Palomba, *Member, IEEE*, Marco Zanoni, Francesca Arcelli Fontana, *Member, IEEE*, Andrea De Lucia, *Senior Member, IEEE*, and Rocco Oliveto

Abstract—Code smells are symptoms of poor design and implementation choices. Previous studies empirically assessed the impact of smells on code quality and clearly indicate their negative impact on maintainability, including a higher bug-proneness of components affected by code smells. In this paper, we capture previous findings on bug-proneness to build a specialized bug prediction model for smelly classes. Specifically, we evaluate the contribution of a measure of the severity of code smells (i.e., code smell intensity) by adding it to existing bug prediction models based on both product and process metrics, and comparing the results of the new model against the baseline models. Results indicate that the accuracy of a bug prediction model increases by adding the code smell intensity as predictor. We also compare the results achieved by the proposed model with the ones of an alternative technique which considers metrics about the history of code smells in files, finding that our model works generally better. However, we observed interesting complementarities between the set of *buggy and smelly* classes correctly classified by the two models. By evaluating the actual information gain provided by the intensity index with respect to the other metrics in the model, we found that the intensity index is a relevant feature for both product and process metrics-based models. At the same time, the metric counting the average number of code smells in previous versions of a class considered by the alternative model is also able to reduce the entropy of the model. On the basis of this result, we devise and evaluate a *smell-aware* combined bug prediction model that included product, process, and smell-related features. We demonstrate how such model classifies bug-prone code components with an F-Measure at least 13 percent higher than the existing state-of-the-art models.

Index Terms—Code smells, bug prediction, empirical study, mining software repositories

A Developer Centered Bug Prediction Model

Dario Di Nucci, *Student Member, IEEE*, Fabio Palomba, *Member, IEEE*, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia, *Senior Member, IEEE*

Abstract—Several techniques have been proposed to accurately predict software defects. These techniques generally exploit characteristics of the code artefacts (e.g., size, complexity, etc.) and/or of the process adopted during their development and maintenance (e.g., the number of developers working on a component) to spot out components likely containing bugs. While these bug prediction models achieve good levels of accuracy, they mostly ignore the major role played by human-related factors in the introduction of bugs. Previous studies have demonstrated that focused developers are less prone to introduce defects than non-focused developers. According to this observation, software components changed by focused developers should also be less error prone than components changed by less focused developers. We capture this observation by measuring the scattering of changes performed by developers working on a component and use this information to build a bug prediction model. Such a model has been evaluated on 26 systems and compared with four competitive techniques. The achieved results show the superiority of our model, and its high complementarity with respect to predictors commonly used in the literature. Based on this result, we also show the results of a "hybrid" prediction model combining our predictors with the existing ones.

Index Terms—Scattering metrics, bug prediction, empirical study, mining software repositories

1 Introduction

Bug prediction techniques are used to identify areas of software systems that are more likely to contain bugs. These prediction models represent an important aid when the resources available for testing are scarce, since they can indicate *where* to invest such resources. The scientific community has developed several bug prediction models that can be roughly classified into two families, based on the information they exploit to discriminate between "buggy" and "clean" code components. The first set of techniques exploits *product metrics* (i.e., metrics capturing intrinsic characteristics of the code components, like their size and complexity) [1], [2], [3], [4], [5], while the second one focuses on *process metrics* (i.e., metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) [6], [7], [8], [9], [10], [11], [12]. While some studies highlighted the superiority of these latter with respect to the *product metric based* techniques [7], [11], [13] there is a general consensus on the fact that no technique is the best in all contexts [14], [15]. For this reason, the research community is still spending effort in investigating under which circumstances and during which coding activities developers tend to introduce bugs (see, e.g., [16], [17], [18], [19], [20], [21], [22]).

Some of these studies have highlighted the central role played by developer-related factors in the introduction of bugs.

In particular, Eyolfson et al. [17] showed that more experienced developers tend to introduce less faults in software systems. Rahman and Devanbu [18] partly contradicted the study by Eyolfson et al. by showing that the experience of a developer has no clear link with the bug introduction. Bird et al. [20] found that high levels of ownership are associated with fewer bugs. Finally, Posnett et al. [22] showed that focused developers (i.e., developers focusing their attention on a specific part of the system) introduce fewer bugs than unfocused developers.

Although such studies showed the potential of human-related factors in bug prediction, this information is not captured in state-of-the-art bug prediction models based on process metrics extracted from version history. Indeed, previous bug prediction models exploit predictors based on (i)

Empirical validation of object-oriented metrics for predicting fault proneness models

Yogesh Singh · Arvinder Kaur · Ruchika Malhotra

Published online: 1 July 2009

Abstract Empirical validation of software metrics used to predict software qu attributes is important to ensure their practical relevance in software organizations. aim of this work is to find the relation of object-oriented (OO) metrics with fault prone

A Validation of Object-Oriented Design Metrics as Quality Indicators

Victor R. Basili, *Fellow, IEEE*, Lionel C. Briand, and Walcélio L. Melo, *Member, IEEE Computer Society*

Abstract—This paper presents the results of a study in which we empirically investigated the suite of object-oriented (OO) des metrics introduced in [13]. More specifically, our goal is to assess these metrics as predictors of fault-prone classes and, theref determine whether they can be used as early quality indicators. This study is complementary to the work described in [30] whe same suite of metrics had been used to assess frequencies of maintenance changes to classes. To perform our validation accurately, we collected data on the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method the C++ programming language. Based on empirical and quantitative analysis, the advantages and drawbacks of these OO m are discussed. Several of Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during th early phases of the life-cycle. Also, on our data set, they are better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes.

Index Terms—Object-oriented design metrics, error prediction model, object-oriented software development, C++ programmin language.

1 Introduction

1.1 Motivation

The development of a large software system is a time- and resource-consuming activity. Even with the increasing automation of software development activities, resources are still scarce. Therefore, we need to be able to provide accurate information and guidelines to managers to help them make decisions, plan and schedule activities, and allocate resources for the different software activities that take place during software development. Software metrics are, thus, necessary to identify where the resources are needed; they are a crucial source of information for decision-making [22].

Testing of large systems is an example of a resource- and time-consuming activity. Applying equal testing and verification effort to all parts of a software system has become cost-prohibitive. Therefore, one needs to be able to identify fault-prone modules so that testing/verification effort can be concentrated on these modules [21]. The availability of adequate product design metrics for characterizing error-

Many product metrics have been proposed used, and, sometimes, empirically validated [3 [30], e.g., number of lines of code, McCabe comp ric, etc. In fact, many companies have built their quality, and resource prediction models based metrics. TRW [7], the Software Engineering (SEL) [31], and Hewlett Packard [20] are examp ware organizations that have been using product build their cost, resource, defect, and productivity

1.2 Issues

In the last decade, many companies have starte duce object-oriented (OO) technology into thei development environments. OO analysis/design OO languages, and OO development environ currently popular worldwide in both small and ware organizations. The insertion of OO techno software industry, however, has created new cha

Empirical Software Engineering (2020) 25:49–95
https://doi.org/10.1007/s10664-019-09739-0

Improving change prediction models with code smell-related information

Gemma Catolino[1] · Fabio Palomba[2] · Francesca Arcelli Fontana[3] · Andrea De Lucia[1] · Andy Zaidman[4] · Filomena Ferrucci[1]

Published online: 2 August 2019

Abstract
Code smells are sub-optimal implementation choices applied by developers that have the effect of negatively impacting, among others, the change-proneness of the affected classes. Based on this consideration, in this paper we conjecture that code smell-related information can be effectively exploited to improve the performance of change prediction models, i.e., models having the goal of indicating which classes are more likely to change in the future. We exploit the so-called *intensity index*—a previously defined metric that captures the severity of a code smell—and evaluate its contribution when added as additional feature in the context of three state of the art change prediction models based on product, process, and developer-based features. We also compare the performance achieved by the proposed model with a model based on previously defined antipattern metrics, a set of indicators computed considering the history of code smells in files. Our results report that (i) the prediction performance of the intensity-including models is statistically better than the baselines and, (ii) the intensity is a better predictor than antipattern metrics. We observed some orthogonality between the set of change-prone and non-change-prone classes correctly classified by the models relying on intensity and antipattern metrics: for this reason, we also devise and evaluate a smell-aware combined change prediction model including product, process, developer-based, and smell-related features. We show that the F-Measure of this model is notably higher than other models.

Keywords Change prediction · Code smells · Empirical study

1 Introduction

Code Quality: State-of-the-art

Toward a Smell-Aware Bug Prediction Model

Empirical validation of object-oriented metrics for predicting fault proneness models

Predicting Fault-Proneness using OO Metrics
An Industrial Case Study

The **code quality variation** over time is **rarely considered**

A Developer Centered Bug Prediction Model

Dario Di Nucci, *Student Member, IEEE*, Fabio Palomba, *Member, IEEE*, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia, *Senior Member, IEEE*

Abstract—Several techniques have been proposed to accurately predict software defects. These techniques generally exploit characteristics of the code artefacts (e.g., size, complexity, etc.) and/or of the process adopted during their development and maintenance (e.g., the number of developers working on a component) to spot out components likely containing bugs. While these bug prediction models achieve good levels of accuracy, they mostly ignore the major role played by human-related factors in the introduction of bugs. Previous studies have demonstrated that focused developers are less prone to introduce defects than non-focused developers. According to this observation, software components changed by focused developers should also be less error prone than components changed by less focused developers. We capture this observation by measuring the scattering of changes performed by developers working on a component and use this information to build a bug prediction model. Such a model has been evaluated on 26 systems and compared with four competitive techniques. The achieved results show the superiority of our model, and its high complementarity with respect to predictors commonly used in the literature. Based on this result, we also show the results of a "hybrid" prediction model combining our predictors with the existing ones.

Index Terms—Scattering metrics, bug prediction, empirical study, mining software repositories

1 Introduction

Bug prediction techniques are used to identify areas of software systems that are more likely to contain bugs. These prediction models represent an important aid when the resources available for testing are scarce, since they can indicate *where* to invest such resources. The scientific community has developed several bug prediction models that can be roughly classified into two families, based on the information they exploit to discriminate between "buggy" and "clean" code components. The first set of techniques exploits *product metrics* (i.e., metrics capturing intrinsic characteristics of the code components, like their size and complexity) [1], [2], [3], [4], [5], while the second one focuses on *process metrics* (i.e., metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) [6], [7], [8], [9], [10], [11], [12]. While some studies highlighted the superiority of these latter with respect to the *product metric based* techniques [7], [11], [13] there is a general consensus on the fact that no technique is the best in all contexts [14], [15]. For this reason, the research community is still spending effort in investigating under which circumstances and during which coding activities developers tend to introduce bugs (see, e.g., [16], [17], [18], [19], [20], [21], [22]).

Some of these studies have highlighted the central role played by developer-related factors in the introduction of bugs.

In particular, Eyolfson et al. [17] showed that more experienced developers tend to introduce less faults in software systems. Rahman and Devanbu [18] partly contradicted the study by Eyolfson et al. by showing that the experience of a developer has no clear link with the bug introduction. Bird et al. [20] found that high levels of ownership are associated with fewer bugs. Finally, Posnett et al. [22] showed that focused developers (i.e., developers focusing their attention on a specific part of the system) introduce fewer bugs than unfocused developers.

Although such studies showed the potential of human-related factors in bug prediction, this information is not captured in state-of-the-art bug prediction models based on process metrics extracted from version history. Indeed, previous bug prediction models exploit predictors based on (i)

A Validation of Object-Oriented Design Metrics as Quality Indicators

Victor R. Basili, *Fellow, IEEE*, Lionel C. Briand, and Walcélio L. Melo, *Member, IEEE Computer Society*

Abstract—This paper presents the results of a study in which we empirically investigated the suite of object-oriented (OO) design metrics introduced in [13]. More specifically, our goal is to assess these metrics as predictors of fault-prone classes and, therefore, determine whether they can be used as early quality indicators. This study is complementary to the work described in [30] where the same suite of metrics had been used to assess frequencies of maintenance changes to classes. To perform our validation accurately, we collected data on the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method and the C++ programming language. Based on empirical and quantitative analysis, the advantages and drawbacks of these OO metrics are discussed. Several of Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle. Also, on our data set, they are better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes.

Index Terms—Object-oriented design metrics, error prediction model, object-oriented software development, C++ programming language.

1 Introduction

1.1 Motivation

The development of a large software system is a time- and resource-consuming activity. Even with the increasing automation of software development activities, resources are still scarce. Therefore, we need to be able to provide accurate information and guidelines to managers to help them make decisions, plan and schedule activities, and allocate resources for the different software activities that take place during software development. Software metrics are, thus, necessary to identify where the resources are needed; they are a crucial source of information for decision-making [22].

Testing of large systems is an example of a resource- and time-consuming activity. Applying equal testing and verification effort to all parts of a software system has become cost-prohibitive. Therefore, one needs to be able to identify fault-prone modules so that testing/verification effort can be concentrated on these modules [21]. The availability of adequate product design metrics for characterizing error-

Many product metrics have been proposed [16], [26], used, and, sometimes, empirically validated [3], [4], [19], [30], e.g., number of lines of code, McCabe complexity metric, etc. In fact, many companies have built their own cost, quality, and resource prediction models based on product metrics. TRW [7], the Software Engineering Laboratory (SEL) [31], and Hewlett Packard [20] are examples of software organizations that have been using product metrics to build their cost, resource, defect, and productivity models.

1.2 Issues

In the last decade, many companies have started to introduce object-oriented (OO) technology into their software development environments. OO analysis/design methods, OO languages, and OO development environments are currently popular worldwide in both small and large software organizations. The insertion of OO technology in the software industry, however, has created new challenges for

Empirical Software Engineering (2020) 25:49–95
https://doi.org/10.1007/s10664-019-09739-0

Check for updates

Improving change prediction models with code smell-related information

Gemma Catolino[1] · Fabio Palomba[2] · Francesca Arcelli Fontana[3] · Andrea De Lucia[1] · Andy Zaidman[4] · Filomena Ferrucci[1]

Published online: 2 August 2019

Abstract
Code smells are sub-optimal implementation choices applied by developers that have the effect of negatively impacting, among others, the change-proneness of the affected classes. Based on this consideration, in this paper we conjecture that code smell-related information can be effectively exploited to improve the performance of change prediction models, i.e., models having the goal of indicating which classes are more likely to change in the future. We exploit the so-called *intensity index*—a previously defined metric that captures the severity of a code smell—and evaluate its contribution when added as additional feature in the context of three state of the art change prediction models based on product, process, and developer-based features. We also compare the performance achieved by the proposed model with a model based on previously defined antipattern metrics, a set of indicators computed considering the history of code smells in files. Our results report that (i) the prediction performance of the intensity-including models is statistically better than the baselines and, (ii) the intensity is a better predictor than antipattern metrics. We observed some orthogonality between the set of change-prone and non-change-prone classes correctly classified by the models relying on intensity and antipattern metrics: for this reason, we also devise and evaluate a smell-aware combined change prediction model including product, process, developer-based, and smell-related features. We show that the F-Measure of this model is notably higher than other models.

Keywords Change prediction · Code smells · Empirical study

Code Quality: State-of-the-art

The **code quality variation** over time is **rarely considered**

There is a lack of coverage of how **code instruments impact code quality**

194 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 45, NO. 2, FEBRUARY 2019

Toward a Smell-Aware Bug Prediction Model

Fabio Palomba, Member, IEEE, Marco Zanoni, Francesca Arcelli Fontana, Member, IEEE

Empirical validation of object-oriented metrics for predicting fault proneness models

Predicting Fault-Proneness using OO Metrics
An Industrial Case Study

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 1, JANUARY 2018 5

A Developer Centered Bug Prediction Model

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 22, NO. 10, OCTOBER 1996 751

Empirical Software Engineering (2020) 25:49–95
https://doi.org/10.1007/s10664-019-09739-0

1 Introduction

Bug prediction techniques are used to identify areas of software systems that are more likely to contain bugs. These prediction models represent an important aid when the resources available for testing are scarce, since they can indicate *where* to invest such resources. The scientific community has developed several bug prediction models that can be roughly classified into two families, based on the information they exploit to discriminate between "buggy" and "clean" code components. The first set of techniques exploits *product metrics* (i.e., metrics capturing intrinsic characteristics of the code components, like their size and complexity) [1], [2], [3], [4], [5], while the second one focuses on *process metrics* (i.e., metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) [6], [7], [8], [9], [10], [11], [12]. While some studies highlighted the superiority of these latter with respect to the *product metric based* techniques [7], [11], [13] there is a general consensus on the fact that no technique is the best in all contexts [14], [15]. For this reason, the research community is still spending effort in investigating under which circumstances and during which coding activities developers tend to introduce bugs (see, e.g., [16], [17], [18], [19], [20], [21], [22]).

Some of these studies have highlighted the central role played by developer-related factors in the introduction of bugs.

In particular, Eyolfson et al. [17] showed that more experienced developers tend to introduce less faults in software systems. Rahman and Devanbu [18] partly contradicted the study by Eyolfson et al. by showing that the experience of a developer has no clear link with the bug introduction. Bird et al. [20] found that high levels of ownership are associated with fewer bugs. Finally, Posnett et al. [22] showed that focused developers (i.e., developers focusing their attention on a specific part of the system) introduce fewer bugs than unfocused developers.

Although such studies showed the potential of human-related factors in bug prediction, this information is not captured in state-of-the-art bug prediction models based on process metrics extracted from version history. Indeed, previous bug prediction models exploit predictors based on (i)

1 Introduction

1.1 Motivation

The development of a large software system is a time- and resource-consuming activity. Even with the increasing automation of software development activities, resources are still scarce. Therefore, we need to be able to provide accurate information and guidelines to managers to help them make decisions, plan and schedule activities, and allocate resources for the different software activities that take place during software development. Software metrics are, thus, necessary to identify where the resources are needed; they are a crucial source of information for decision-making [22].

Testing of large systems is an example of a resource- and time-consuming activity. Applying equal testing and verification effort to all parts of a software system has become cost-prohibitive. Therefore, one needs to be able to identify fault-prone modules so that testing/verification effort can be concentrated on these modules [21]. The availability of adequate product design metrics for characterizing error-

Many product metrics have been proposed [16], [26], used, and, sometimes, empirically validated [3], [4], [19], [30], e.g., number of lines of code, McCabe complexity metric, etc. In fact, many companies have built their own cost, quality, and resource prediction models based on product metrics. TRW [7], the Software Engineering Laboratory (SEL) [31], and Hewlett Packard [20] are examples of software organizations that have been using product metrics to build their cost, resource, defect, and productivity models.

1.2 Issues

In the last decade, many companies have started to introduce object-oriented (OO) technology into their software development environments. OO analysis/design methods, OO languages, and OO development environments are currently popular worldwide in both small and large software organizations. The insertion of OO technology in the software industry, however, has created new challenges for

tion can be effectively exploited to improve the performance of change prediction models, i.e., models having the goal of indicating which classes are more likely to change in the future. We exploit the so-called *intensity index*—a previously defined metric that captures the severity of a code smell—and evaluate its contribution when added as additional feature in the context of three state of the art change prediction models based on product, process, and developer-based features. We also compare the performance achieved by the proposed model with a model based on previously defined antipattern metrics, a set of indicators computed considering the history of code smells in files. Our results report that (i) the prediction performance of the intensity-including models is statistically better than the baselines and, (ii) the intensity is a better predictor than antipattern metrics. We observed some orthogonality between the set of change-prone and non-change-prone classes correctly classified by the models relying on intensity and antipattern metrics: for this reason, we also devise and evaluate a smell-aware combined change prediction model including product, process, developer-based, and smell-related features. We show that the F-Measure of this model is notably higher than other models.

Keywords Change prediction · Code smells · Empirical study

Ultimate Goal

Ultimate Goal

The *goal* of this Ph.D. project was to perform a step toward ***Evolutionary Code Quality*** to provide a framework that practitioners can use to perform quality assurance over time

Research Goal

Research Goal

Understanding How Reusability Mechanisms and Built-in Features Affect Code Quality **Over Time**

Large-Scale Mining Software Repository Techniques

Why MSR Techniques?

MSR techniques allow us to **analyze** code quality by considering how the r**epository evolves over time**

Challenges

Challenges

Challenges

Different Code Quality Attributes

Multiple Code Instruments

Multiple Families of Software Systems

Challenges

Different Code Quality Attributes

Multiple Code Instruments

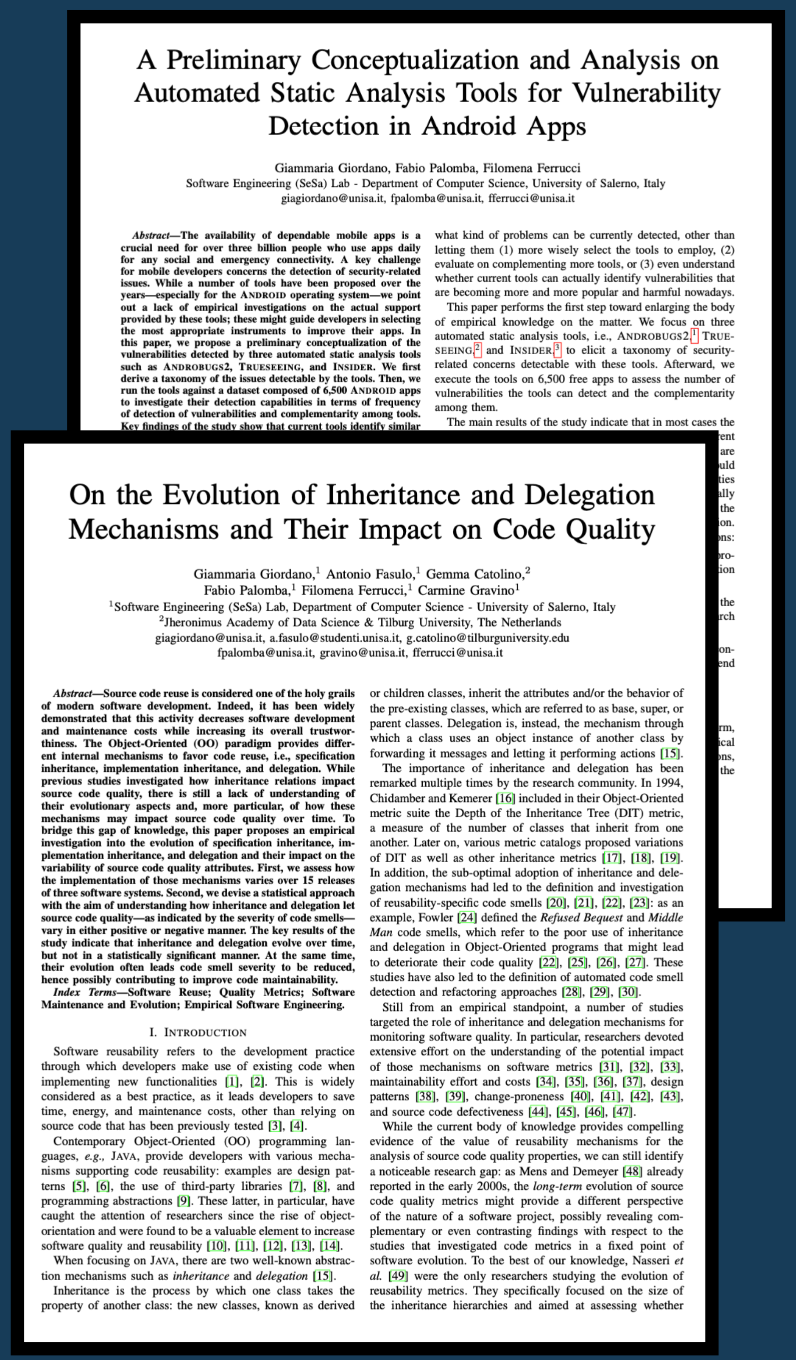
Multiple Families of Software Systems

Challenges

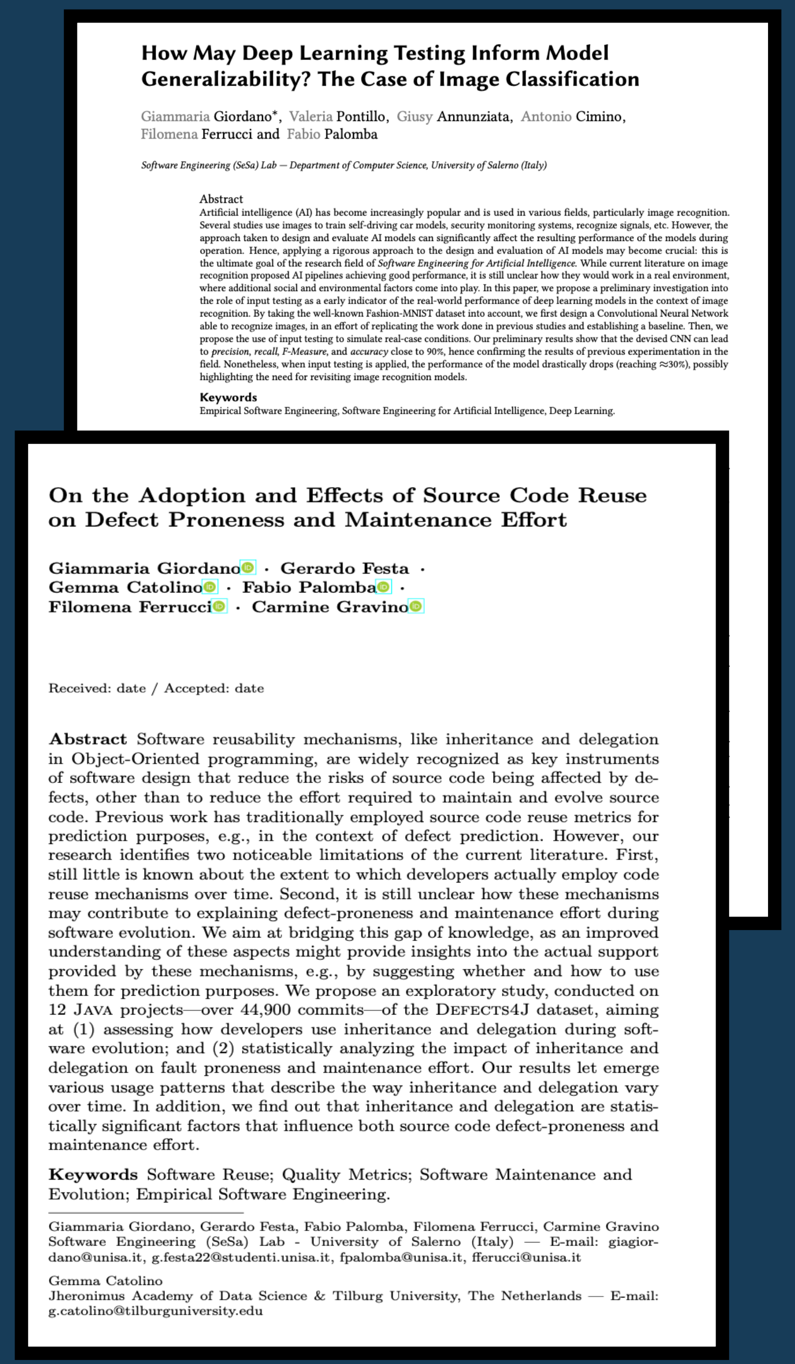
Multiple Code Instruments

Multiple Families of Software Systems

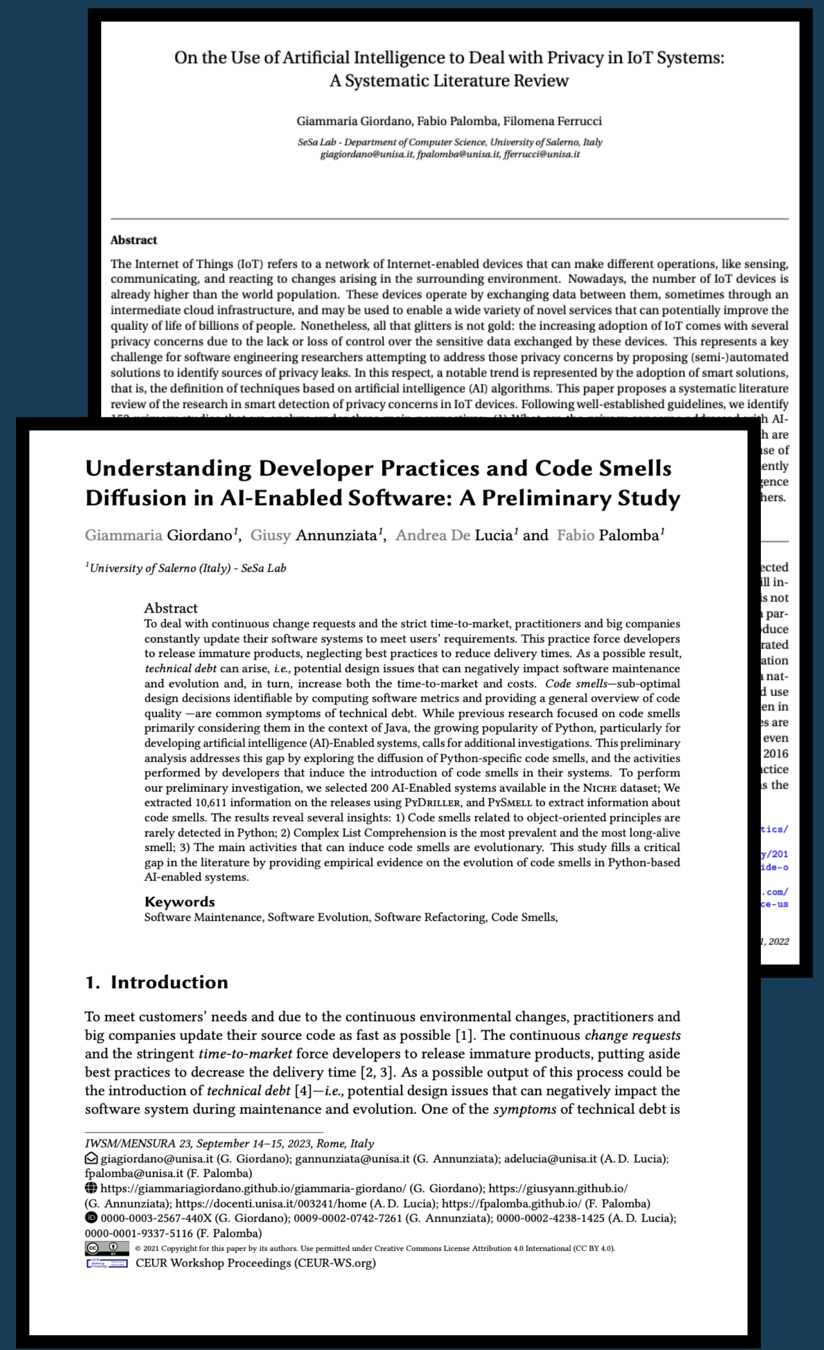
Challenges



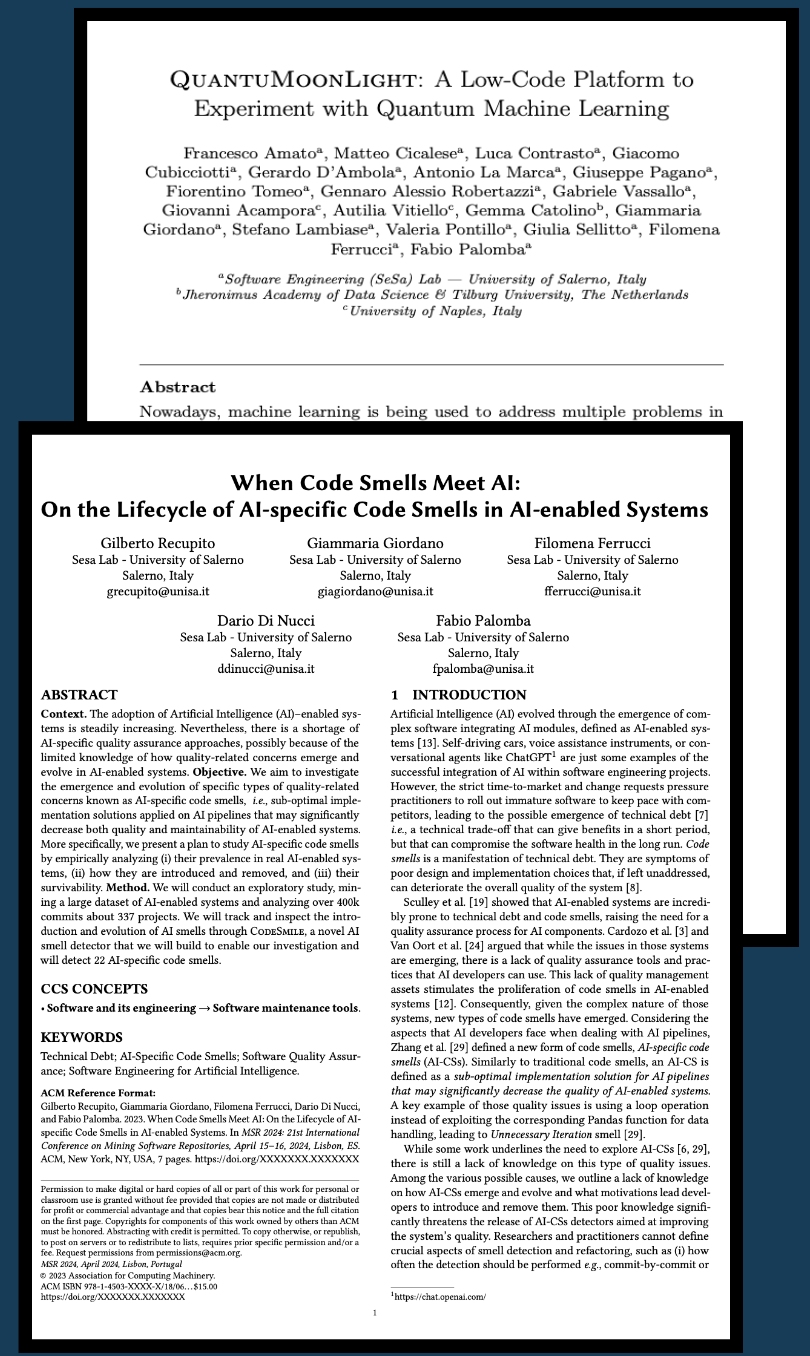
Software analysis, evolution, and Reengineering (SANER)



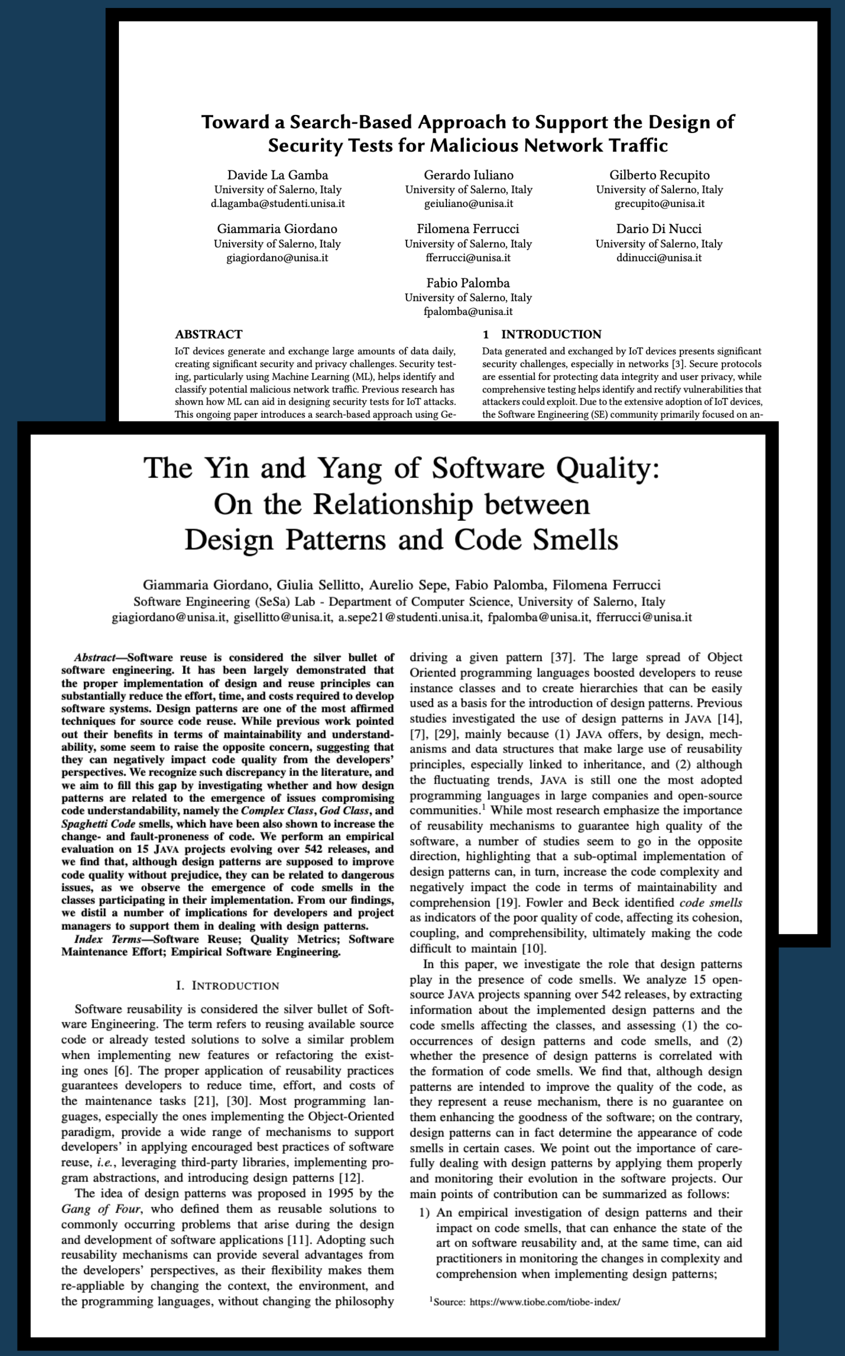
Empirical Software Engineering (EMSE)



International Conference on Software Process and Product Measurement (MENSURA)

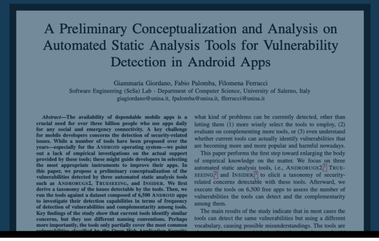


Mining Software Repository (MSR) - (Registered-Report)



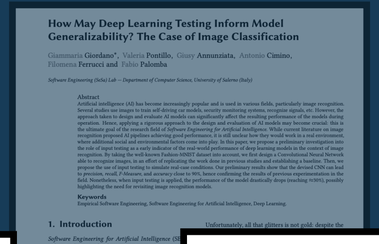
Euromicro SEAA 2023

Challenges



A Preliminary Conceptualization and Analysis on Automated Static Analysis Tools for Vulnerability Detection in Android Apps

Giammaria Giordano, Fabio Palomba, Filomena Ferrucci
Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy
giagiordano@unisa.it, fpalomba@unisa.it, fferrucci@unisa.it



How May Deep Learning Testing Inform Model Generalizability? The Case of Image Classification

Giammaria Giordano*, Valeria Pontillo, Giusy Annunziata, Antonio Cimino, Filomena Ferrucci and Fabio Palomba

Software Engineering (SeSa) Lab — Department of Computer Science, University of Salerno (Italy)

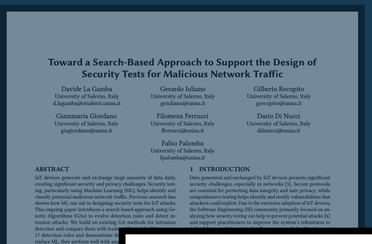
On the Use of Artificial Intelligence to Deal with Privacy in IoT Systems: A Systematic Literature Review

Giammaria Giordano, Fabio Palomba, Filomena Ferrucci

SeSa Lab - Department of Computer Science, University of Salerno, Italy
giagiordano@unisa.it, fpalomba@unisa.it, fferrucci@unisa.it

QuantuMoonLight: A Low-Code Platform to Experiment with Quantum Machine Learning

Francesco Amato*, Matteo Cicalese*, Luca Contrasto*, Giacomo Cubicciotti*, Gerardo D'Ambola*, Antonio La Marca*, Giuseppe Pagano*, Fiorentino Tonno*, Gennaro Alessio Robertazzi*, Gabriele Vassallo*, Giovanni Acamporа*, Autilia Vitiello*, Gemma Catolino*, Giammaria Giordano*, Stefano Lambiase*, Valeria Pontillo*, Giulia Sellitto*, Filomena Ferrucci*, Fabio Palomba*



Toward a Search-Based Approach to Support the Design of Security Tests for Malicious Network Traffic

Davide La Gamba, Gerardo Iuliano, Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, Fabio Palomba

On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Giammaria Giordano,[1] Antonio Fasulo,[1] Gemma Catolino,[2]
Fabio Palomba,[1] Filomena Ferrucci,[1] Carmine Gravino[1]
[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
giagiordano@unisa.it, a.fasulo@studenti.unisa.it, g.catolino@tilburguniversity.edu
fpalomba@unisa.it, gravino@unisa.it, fferrucci@unisa.it

***Abstract*—Source code reuse is considered one of the holy grails of modern software development. Indeed, it has been widely demonstrated that this activity decreases software development and maintenance costs while increasing its overall trustworthiness. The Object-Oriented (OO) paradigm provides different internal mechanisms to favor code reuse, i.e., specification inheritance, implementation inheritance, and delegation. While previous studies investigated how inheritance relations impact source code quality, there is still a lack of understanding of their evolutionary aspects and, more particular, of how these mechanisms may impact source code quality over time. To bridge this gap of knowledge, this paper proposes an empirical investigation into the evolution of specification inheritance, implementation inheritance, and delegation and their impact on the variability of source code quality attributes. First, we assess how the implementation of those mechanisms varies over 15 releases of three software systems. Second, we devise a statistical approach with the aim of understanding how inheritance and delegation let source code quality—as indicated by the severity of code smells—vary in either positive or negative manner. The key results of the study indicate that inheritance and delegation evolve over time, but not in a statistically significant manner. At the same time, their evolution often leads code smell severity to be reduced, hence possibly contributing to improve code maintainability.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.**

I. Introduction

Software reusability refers to the development practice through which developers make use of existing code when implementing new functionalities [1], [2]. This is widely considered as a best practice, as it leads developers to save time, energy, and maintenance costs, other than relying on source code that has been previously tested [3], [4].

Contemporary Object-Oriented (OO) programming languages, *e.g.*, Java, provide developers with various mechanisms supporting code reusability: examples are design patterns [5], [6], the use of third-party libraries [7], [8], and programming abstractions [9]. These latter, in particular, have caught the attention of researchers since the rise of object-orientation and were found to be a valuable element to increase software quality and reusability [10], [11], [12], [13], [14].

When focusing on Java, there are two well-known abstraction mechanisms such as *inheritance* and *delegation* [15].

Inheritance is the process by which one class takes the property of another class: the new classes, known as derived or children classes, inherit the attributes and/or the behavior of the pre-existing classes, which are referred to as base, super, or parent classes. Delegation is, instead, the mechanism through which a class uses an object instance of another class by forwarding it messages and letting it performing actions [15].

The importance of inheritance and delegation has been remarked multiple times by the research community. In 1994, Chidamber and Kemerer [16] included in their Object-Oriented metric suite the Depth of the Inheritance Tree (DIT) metric, a measure of the number of classes that inherit from one another. Later on, various metric catalogs proposed variations of DIT as well as other inheritance metrics [17], [18], [19]. In addition, the sub-optimal adoption of inheritance and delegation mechanisms had led to the definition and investigation of reusability-specific code smells [20], [21], [22], [23]: as an example, Fowler [24] defined the *Refused Bequest* and *Middle Man* code smells, which refer to the poor use of inheritance and delegation in Object-Oriented programs that might lead to deteriorate their code quality [22], [25], [26], [27]. These studies have also led to the definition of automated code smell detection and refactoring approaches [28], [29], [30].

Still from an empirical standpoint, a number of studies targeted the role of inheritance and delegation mechanisms for monitoring software quality. In particular, researchers devoted extensive effort on the understanding of the potential impact of those mechanisms on software metrics [31], [32], [33], maintainability effort and costs [34], [35], [36], [37], design patterns [38], [39], change-proneness [40], [41], [42], [43], and source code defectiveness [44], [45], [46], [47].

While the current body of knowledge provides compelling evidence of the value of reusability mechanisms for the analysis of source code quality properties, we can still identify a noticeable research gap: as Mens and Demeyer [48] already reported in the early 2000s, the *long-term* evolution of source code quality metrics might provide a different perspective of the nature of a software project, possibly revealing complementary or even contrasting findings with respect to the studies that investigated code metrics in a fixed point of software evolution. To the best of our knowledge, Nasseri *et al.* [49] were the only researchers studying the evolution of reusability metrics. They specifically focused on the size of the inheritance hierarchies and aimed at assessing whether

On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

Giammaria Giordano · Gerardo Festa · Gemma Catolino · Fabio Palomba · Filomena Ferrucci · Carmine Gravino

Received: date / Accepted: date

Abstract Software reusability mechanisms, like inheritance and delegation in Object-Oriented programming, are widely recognized as key instruments of software design that reduce the risks of source code being affected by defects, other than to reduce the effort required to maintain and evolve source code. Previous work has traditionally employed source code reuse metrics for prediction purposes, e.g., in the context of defect prediction. However, our research identifies two noticeable limitations of the current literature. First, still little is known about the extent to which developers actually employ code reuse mechanisms over time. Second, it is still unclear how these mechanisms may contribute to explaining defect-proneness and maintenance effort during software evolution. We aim at bridging this gap of knowledge, as an improved understanding of these aspects might provide insights into the actual support provided by these mechanisms, e.g., by suggesting whether and how to use them for prediction purposes. We propose an exploratory study, conducted on 12 Java projects—over 44,900 commits—of the Defects4J dataset, aiming at (1) assessing how developers use inheritance and delegation during software evolution; and (2) statistically analyzing the impact of inheritance and delegation on fault proneness and maintenance effort. Our results let emerge various usage patterns that describe the way inheritance and delegation vary over time. In addition, we find out that inheritance and delegation are statistically significant factors that influence both source code defect-proneness and maintenance effort.

Keywords Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.

Giammaria Giordano, Gerardo Festa, Fabio Palomba, Filomena Ferrucci, Carmine Gravino
Software Engineering (SeSa) Lab - University of Salerno (Italy) — E-mail: giagiordano@unisa.it, g.festa22@studenti.unisa.it, fpalomba@unisa.it, fferrucci@unisa.it

Gemma Catolino
Jheronimus Academy of Data Science & Tilburg University, The Netherlands — E-mail: g.catolino@tilburguniversity.edu

Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study

Giammaria Giordano[1], Giusy Annunziata[1], Andrea De Lucia[1] and Fabio Palomba[1]

[1]*University of Salerno (Italy) - SeSa Lab*

Abstract
To deal with continuous change requests and the strict time-to-market, practitioners and big companies constantly update their software systems to meet users' requirements. This practice force developers to release immature products, neglecting best practices to reduce delivery times. As a possible result, *technical debt* can arise, *i.e.*, potential design issues that can negatively impact software maintenance and evolution and, in turn, increase both the time-to-market and costs. *Code smells*—sub-optimal design decisions identifiable by computing software metrics and providing a general overview of code quality —are common symptoms of technical debt. While previous research focused on code smells primarily considering them in the context of Java, the growing popularity of Python, particularly for developing artificial intelligence (AI)-Enabled systems, calls for additional investigations. This preliminary analysis addresses this gap by exploring the diffusion of Python-specific code smells, and the activities performed by developers that induce the introduction of code smells in their systems. To perform our preliminary investigation, we selected 200 AI-Enabled systems available in the Niche dataset; We extracted 10,611 information on the releases using PyDriller, and PySmell to extract information about code smells. The results reveal several insights: 1) Code smells related to object-oriented principles are rarely detected in Python; 2) Complex List Comprehension is the most prevalent and the most long-alive smell; 3) The main activities that can induce code smells are evolutionary. This study fills a critical gap in the literature by providing empirical evidence on the evolution of code smells in Python-based AI-enabled systems.

Keywords
Software Maintenance, Software Evolution, Software Refactoring, Code Smells,

1. Introduction

To meet customers' needs and due to the continuous environmental changes, practitioners and big companies update their source code as fast as possible [1]. The continuous *change requests* and the stringent *time-to-market* force developers to release immature products, putting aside best practices to decrease the delivery time [2, 3]. As a possible output of this process could be the introduction of *technical debt* [4]—*i.e.*, potential design issues that can negatively impact the software system during maintenance and evolution. One of the *symptoms* of technical debt is

IWSM/MENSURA 23, September 14–15, 2023, Rome, Italy
giagiordano@unisa.it (G. Giordano); gannunziata@unisa.it (G. Annunziata); adelucia@unisa.it (A. D. Lucia); fpalomba@unisa.it (F. Palomba)
https://giammariagiordano.github.io/giammaria-giordano/ (G. Giordano); https://giusyann.github.io/ (G. Annunziata); https://docenti.unisa.it/003241/home (A. D. Lucia); https://fpalomba.github.io/ (F. Palomba)
0000-0003-2567-440X (G. Giordano); 0009-0002-0742-7261 (G. Annunziata); 0000-0002-4238-1425 (A. D. Lucia); 0000-0001-9337-5116 (F. Palomba)

When Code Smells Meet AI: On the Lifecycle of AI-specific Code Smells in AI-e

Gilberto Recupito
Sesa Lab - University of Salerno
Salerno, Italy
grecupito@unisa.it

Giammaria Giordano
Sesa Lab - University of Salerno
Salerno, Italy
giagiordano@unisa.it

Fil
Sesa Lab
Salerno,
ff

Dario Di Nucci
Sesa Lab - University of Salerno
Salerno, Italy
ddinucci@unisa.it

Fabio Palomba
Sesa Lab - University of Salerno
Salerno, Italy
fpalomba@unisa.it

ABSTRACT

Context. The adoption of Artificial Intelligence (AI)-enabled systems is steadily increasing. Nevertheless, there is a shortage of AI-specific quality assurance approaches, possibly because of the limited knowledge of how quality-related concerns emerge and evolve in AI-enabled systems. **Objective.** We aim to investigate the emergence and evolution of specific types of quality-related concerns known as AI-specific code smells, *i.e.*, sub-optimal implementation solutions applied on AI pipelines that may significantly decrease both quality and maintainability of AI-enabled systems. More specifically, we present a plan to study AI-specific code smells by empirically analyzing (i) their prevalence in real AI-enabled systems, (ii) how they are introduced and removed, and (iii) their survivability. **Method.** We will conduct an exploratory study, mining a large dataset of AI-enabled systems and analyzing over 400k commits about 337 projects. We will track and inspect the introduction and evolution of AI smells through CodeSmile, a novel AI smell detector that we will build to enable our investigation and will detect 22 AI-specific code smells.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Technical Debt; AI-Specific Code Smells; Software Quality Assurance; Software Engineering for Artificial Intelligence.

ACM Reference Format:
Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. 2023. When Code Smells Meet AI: On the Lifecycle of AI-specific Code Smells in AI-enabled Systems. In *MSR 2024: 21st International Conference on Mining Software Repositories, April 15–16, 2024, Lisbon, ES.* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

MSR 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Artificial Intelligence (AI) evolves
plex software integrating AI mo
tems [13]. Self-driving cars, voic
versational agents like ChatGPT
successful integration of AI with
However, the strict time-to-mark
practitioners to roll out immature
petitors, leading to the possible
i.e., a technical trade-off that o
but that can compromise the soft
smells is a manifestation of techn
poor design and implementation
can deteriorate the overall quality

Sculley et al. [19] showed that
bly prone to technical debt and c
quality assurance process for AI c
Van Oort et al. [24] argued that v
are emerging, there is a lack of q
tices that AI developers can use.
assets stimulates the proliferatio
systems [12]. Consequently, gen
systems, new types of code smell
smells (AI-CSs). Similarly to trad
defined as a *sub-optimal implem*
that may significantly decrease t
A key example of those quality
instead of exploiting the corresp
handling, leading to *Unnecessary*

While some work underlines th
there is still a lack of knowledg
Among the various possible cause
on how AI-CSs emerge and evolve
opers to introduce and remove th
cantly threatens the release of AI-
the system's quality. Researchers
crucial aspects of smell detection
often the detection should be perf

[1] https://chat.openai.com/

Software analysis, evolution, and Reengineering (SANER)

Empirical Software Engineering (EMSE)

International Conference on Software Process and Product Measurement (MENSURA)

Mining Software Repository (Registered-Report)

Challenges

tware Engineering (EMSE)

International Conference on Software Process and Product Measurement (MENSURA)

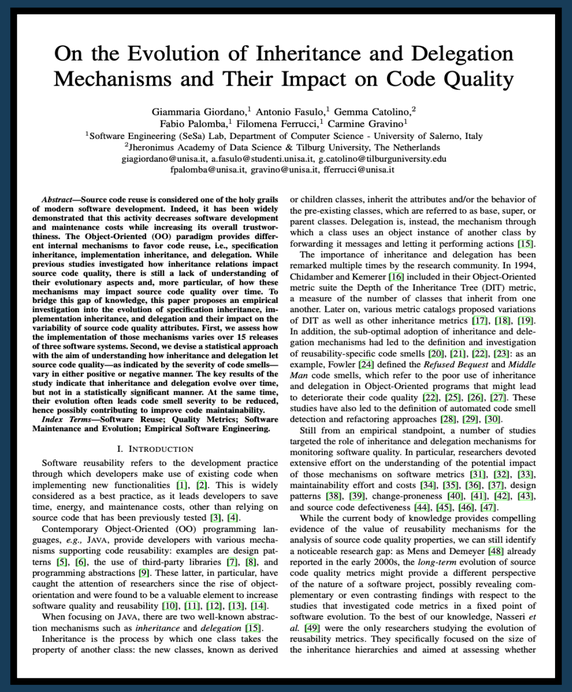
Mining Software Repository (MSR) - (Registered-Report)

Euromicro SEAA 2023

Investigate Multiple Code Quality Attributes Over Time

Investigate Multiple Code Quality Attributes Over Time

Code Smells



On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Giammaria Giordano,[1] Antonio Fasulo,[1] Gemma Catolino,[2]
Fabio Palomba,[1] Filomena Ferrucci,[1] Carmine Gravino[1]
[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
giagiordano@unisa.it, a.fasulo@studenti.unisa.it, g.catolino@tilburguniversity.edu
fpalomba@unisa.it, gravino@unisa.it, fferrucci@unisa.it

***Abstract*—Source code reuse is considered one of the holy grails of modern software development. Indeed, it has been widely demonstrated that this activity decreases software development and maintenance costs while increasing its overall trustworthiness. The Object-Oriented (OO) paradigm provides different internal mechanisms to favor code reuse, i.e., specification inheritance, implementation inheritance, and delegation. While previous studies investigated how inheritance relations impact source code quality, there is still a lack of understanding of their evolutionary aspects and, more particular, of how these mechanisms may impact source code quality over time. To bridge this gap of knowledge, this paper proposes an empirical investigation into the evolution of specification inheritance, implementation inheritance, and delegation and their impact on the variability of source code quality attributes. First, we assess how the implementation of those mechanisms varies over 15 releases of three software systems. Second, we devise a statistical approach with the aim of understanding how inheritance and delegation let source code quality—as indicated by the severity of code smells—vary in either positive or negative manner. The key results of the study indicate that inheritance and delegation evolve over time, but not in a statistically significant manner. At the same time, their evolution often leads code smell severity to be reduced, hence possibly contributing to improve code maintainability.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.**

I. Introduction

Software reusability refers to the development practice through which developers make use of existing code when implementing new functionalities [1], [2]. This is widely considered as a best practice, as it leads developers to save time, energy, and maintenance costs, other than relying on source code that has been previously tested [3], [4].

Contemporary Object-Oriented (OO) programming languages, *e.g.,* Java, provide developers with various mechanisms supporting code reusability: examples are design patterns [5], [6], the use of third-party libraries [7], [8], and programming abstractions [9]. The latter, in particular, have caught the attention of researchers since the rise of object-orientation and were found to be a valuable element to increase software quality and reusability [10], [11], [12], [13], [14].

When focusing on Java, there are two well-known abstraction mechanisms such as *inheritance* and *delegation* [15].

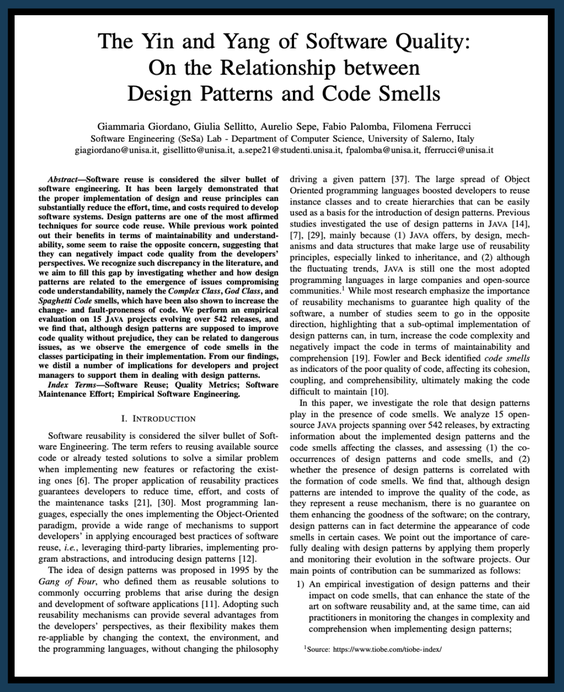
Inheritance is the process by which one class takes the property of another class: the new classes, known as derived or children classes, inherit the attributes and/or the behavior of the pre-existing classes, which are referred to as base, super, or parent classes. Delegation is, instead, the mechanism through which a class uses an object instance of another class by forwarding it messages and letting it performing actions [15].

The importance of inheritance and delegation has been remarked multiple times by the research community. In 1994, Chidamber and Kemerer [16] included in their Object-Oriented metric suite the Depth of the Inheritance Tree (DIT) metric, a measure of the number of classes that inherit from one another. Later on, various metric catalogs proposed variations of DIT as well as other inheritance metrics [17], [18], [19]. In addition, the sub-optimal adoption of inheritance and delegation mechanisms had led to the definition and investigation of reusability-specific code smells [20], [21], [22], [23]: as an example, Fowler [24] defined the *Refused Request* and *Middle Man* code smells, which refer to the poor use of inheritance and delegation in Object-Oriented programs that might lead to deteriorate their code quality [22], [25], [26], [27]. These studies have also led to the definition of automated code smell detection and refactoring approaches [28], [29], [30].

Still from an empirical standpoint, a number of studies targeted the role of inheritance and delegation mechanisms for monitoring software quality. In particular, researchers devoted extensive effort on the understanding of the potential impact of those mechanisms on software metrics [31], [32], [33], maintainability effort and costs [34], [35], [36], [37], design patterns [38], [39], change-proneness [40], [41], [42], [43], and source code defectiveness [44], [45], [46], [47].

While the current body of knowledge provides compelling evidence of the value of reusability mechanisms for the analysis of source code quality properties, we can still identify a noticeable research gap: as Mens and Demeyer [48] already reported in the early 2000s, the *long-term* evolution of source code quality metrics might provide a different perspective of the nature of a software project, possibly revealing complementary or even contrasting findings with respect to the studies that investigated code metrics in a fixed point of software evolution. To the best of our knowledge, Nasseri *et al.* [49] were the only researchers studying the evolution of reusability metrics. They specifically focused on the size of the inheritance hierarchies and aimed at assessing whether

Software analysis, evolution, and Reengineering (SANER)



The Yin and Yang of Software Quality: On the Relationship between Design Patterns and Code Smells

Giammaria Giordano, Giulia Sellitto, Aurelio Sepe, Fabio Palomba, Filomena Ferrucci
Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy
giagiordano@unisa.it, gisellitto@unisa.it, a.sepe21@studenti.unisa.it, fpalomba@unisa.it, fferrucci@unisa.it

***Abstract*—Software reuse is considered the silver bullet of software engineering. It has been largely demonstrated that the proper implementation of design and reuse principles can substantially reduce the effort, time, and costs required to develop software systems. Design patterns are one of the most affirmed techniques for source code reuse. While previous work pointed out their benefits in terms of maintainability and understandability, some seem to raise the opposite concern, suggesting that they can negatively impact code quality from the developers' perspectives. We recognize such discrepancy in the literature, and we aim to fill this gap by investigating whether and how design patterns are related to the emergence of issues compromising code understandability, namely the *Complex Class*, *God Class*, and *Spaghetti Code* smells, which have been also shown to increase the change- and fault-proneness of code. We perform an empirical evaluation on 15 Java projects evolving over 542 releases, and we find that, although design patterns are supposed to improve code quality without prejudice, they can be related to dangerous issues, as we observe the emergence of code smells in the classes participating in their implementation. From our findings, we distil a number of implications for developers and project managers to support them in dealing with design patterns.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance Effort; Empirical Software Engineering.**

I. Introduction

Software reusability is considered the silver bullet of Software Engineering. The term refers to reusing available source code or already tested solutions to solve a similar problem when implementing new features or refactoring the existing ones [6]. The proper application of reusability practices guarantees developers to reduce time, effort, and costs of the maintenance tasks [21], [30]. Most programming languages, especially the ones implementing the Object-Oriented paradigm, provide a wide range of mechanisms to support developers' in applying encouraged best practices of software reuse, *i.e.*, leveraging third-party libraries, implementing program abstractions, and introducing design patterns [12].

The idea of design patterns was proposed in 1995 by the *Gang of Four*, who defined them as reusable solutions to commonly occurring problems that arise during the design and development of software applications [11]. Adopting such reusability mechanisms can provide several advantages from the developers' perspectives, as their flexibility makes them re-appliable by changing the context, the environment, and the programming languages, without changing the philosophy driving a given pattern [37]. The large spread of Object Oriented programming languages boosted developers to reuse instance classes and to create hierarchies that can be easily used as a basis for the introduction of design patterns. Previous studies investigated the use of design patterns in Java [14], [7], [29], mainly because (1) Java offers, by design, mechanisms and data structures that make large use of reusability principles, especially linked to inheritance, and (2) although the fluctuating trends, Java is still one the most adopted programming languages in large companies and open-source communities.[1] While most research emphasize the importance of reusability mechanisms to guarantee high quality of the software, a number of studies seem to go in the opposite direction, highlighting that a sub-optimal implementation of design patterns can, in turn, increase the code complexity and negatively impact the code in terms of maintainability and comprehension [19]. Fowler and Beck identified *code smells* as indicators of the poor quality of code, affecting its cohesion, coupling, and comprehensibility, ultimately making the code difficult to maintain [10].

In this paper, we investigate the role that design patterns play in the presence of code smells. We analyze 15 open-source Java projects spanning over 542 releases, by extracting information about the implemented design patterns and the code smells affecting the classes, and assessing (1) the co-occurrences of design patterns and code smells, and (2) whether the presence of design patterns is correlated with the formation of code smells. We find that, although design patterns are intended to improve the quality of the code, as they represent a reuse mechanism, there is no guarantee on them enhancing the goodness of the software; on the contrary, design patterns can in fact determine the appearance of code smells in certain cases. We point out the importance of carefully dealing with design patterns by applying them properly and monitoring their evolution in the software projects. Our main points of contribution can be summarized as follows:

1) An empirical investigation of design patterns and their impact on code smells, that can enhance the state of the art on software reusability and, at the same time, can aid practitioners in monitoring the changes in complexity and comprehension when implementing design patterns;

[1]Source: https://www.tiobe.com/tiobe-index/

Euromicro SEAA 2023

Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study

Giammaria Giordano[1], Giusy Annunziata[1], Andrea De Lucia[1] and Fabio Palomba[1]
[1]University of Salerno (Italy) - SeSa Lab

Abstract
To deal with continuous change requests and the strict time-to-market, practitioners and big companies constantly update their software systems to meet users' requirements. This practice force developers to release immature products, neglecting best practices to reduce delivery times. As a possible result, *technical debt* can arise, *i.e.*, potential design issues that can negatively impact software maintenance and evolution and, in turn, increase both the time-to-market and costs. *Code smells*—sub-optimal design decisions identifiable by computing software metrics and providing a general overview of code quality —are common symptoms of technical debt. While previous research focused on code smells primarily considering them in the context of Java, the growing popularity of Python, particularly for developing artificial intelligence (AI)-Enabled systems, calls for additional investigations. This preliminary analysis addresses this gap by exploring the diffusion of Python-specific code smells, and the activities performed by developers that induce the introduction of code smells in their systems. To perform our preliminary investigation, we selected 200 AI-Enabled systems available in the Niche dataset; We extracted 10,611 information on the releases using PyDriller, and PySmell to extract information about code smells. The results reveal several insights: 1) Code smells related to object-oriented principles are rarely detected in Python; 2) Complex List Comprehension is the most prevalent and the most long-alive smell; 3) The main activities that can induce code smells are evolutionary. This study fills a critical gap in the literature by providing empirical evidence on the evolution of code smells in Python-based AI-enabled systems.

Keywords
Software Maintenance, Software Evolution, Software Refactoring, Code Smells,

1. Introduction

To meet customers' needs and due to the continuous environmental changes, practitioners and big companies update their source code as fast as possible [1]. The continuous *change requests* and the stringent *time-to-market* force developers to release immature products, putting aside best practices to decrease the delivery time [2, 3]. As a possible output of this process could be the introduction of *technical debt* [4]—*i.e.*, potential design issues that can negatively impact the software system during maintenance and evolution. One of the *symptoms* of technical debt is

IWSM/MENSURA 23, September 14–15, 2023, Rome, Italy
giagiordano@unisa.it (G. Giordano); gannunziata@unisa.it (G. Annunziata); adelucia@unisa.it (A. D. Lucia); fpalomba@unisa.it (F. Palomba)
https://giammariagiordano.github.io/giammaria-giordano/ (G. Giordano); https://giusyann.github.io/ (G. Annunziata); https://docenti.unisa.it/003241/home (A. D. Lucia); https://fpalomba.github.io/ (F. Palomba)
0000-0003-2567-440X (G. Giordano); 0009-0002-0742-7261 (G. Annunziata); 0000-0002-4238-1425 (A. D. Lucia); 0000-0001-9337-5116 (F. Palomba)

International Conference on Software Process and Product Measurement (MENSURA)

When Code Smells Meet AI: On the Lifecycle of AI-specific Code Smells in AI-enabled Systems

Gilberto Recupito
Sesa Lab - University of Salerno
Salerno, Italy
grecupito@unisa.it

Giammaria Giordano
Sesa Lab - University of Salerno
Salerno, Italy
giagiordano@unisa.it

Filomena Ferrucci
Sesa Lab - University of Salerno
Salerno, Italy
fferrucci@unisa.it

Dario Di Nucci
Sesa Lab - University of Salerno
Salerno, Italy
ddinucci@unisa.it

Fabio Palomba
Sesa Lab - University of Salerno
Salerno, Italy
fpalomba@unisa.it

ABSTRACT
Context. The adoption of Artificial Intelligence (AI)–enabled systems is steadily increasing. Nevertheless, there is a shortage of AI-specific quality assurance approaches, possibly because of the limited knowledge of how quality-related concerns emerge and evolve in AI-enabled systems. **Objective.** We aim to investigate the emergence and evolution of specific types of quality-related concerns known as AI-specific code smells, *i.e.*, sub-optimal implementation solutions applied on AI pipelines that may significantly decrease both quality and maintainability of AI-enabled systems. More specifically, we present a plan to study AI-specific code smells by empirically analyzing (i) their prevalence in real AI-enabled systems, (ii) how they are introduced and removed, and (iii) their survivability. **Method.** We will conduct an exploratory study, mining a large dataset of AI-enabled systems and analyzing over 400k commits about 337 projects. We will track and inspect the introduction and evolution of AI smells through CodeSmile, a novel AI smell detector that we will build to enable our investigation and will detect 22 AI-specific code smells.

CCS CONCEPTS
• **Software and its engineering → Software maintenance tools.**

KEYWORDS
Technical Debt; AI-Specific Code Smells; Software Quality Assurance; Software Engineering for Artificial Intelligence.

ACM Reference Format:
Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. 2023. When Code Smells Meet AI: On the Lifecycle of AI-specific Code Smells in AI-enabled Systems. In *MSR 2024: 21st International Conference on Mining Software Repositories, April 15–16, 2024, Lisbon, ES.* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

MSR 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Artificial Intelligence (AI) evolved through the emergence of complex software integrating AI modules, defined as AI-enabled systems [13]. Self-driving cars, voice assistance instruments, or conversational agents like ChatGPT[1] are just some examples of the successful integration of AI within software engineering projects. However, the strict time-to-market and change requests pressure practitioners to roll out immature software to keep pace with competitors, leading to the possible emergence of technical debt [7] *i.e.*, a technical trade-off that can give benefits in a short period, but that can compromise the software health in the long run. *Code smells* is a manifestation of technical debt. They are symptoms of poor design and implementation choices that, if left unaddressed, can deteriorate the overall quality of the system [8].

Sculley et al. [19] showed that AI-enabled systems are incredibly prone to technical debt and code smells, raising the need for a quality assurance process for AI components. Cardoso et al. [5] and Van Oort et al. [24] argued that while the issues in those systems are emerging, there is a lack of quality assurance tools and practices that AI developers can use. This lack of quality management assets stimulates the proliferation of code smells in AI-enabled systems [12]. Consequently, given the complex nature of those systems, new types of code smells have emerged. Considering the aspects that AI developers face when dealing with AI pipelines, Zhang et al. [29] defined a new form of code smells, *AI-specific code smells* (AI-CSs). Similarly to traditional code smells, an AI-CS is defined as a *sub-optimal implementation solution for AI pipelines that may significantly decrease the quality of AI-enabled systems.* A key example of those quality issues is using a loop operation instead of exploiting the corresponding Pandas function for data handling, leading to *Unnecessary Iteration* smell [29].

While some work underlines the need to explore AI-CSs [6, 29], there is still a lack of knowledge on this type of quality issues. Among the various possible causes, we outline a lack of knowledge on how AI-CSs emerge and evolve and what motivations lead developers to introduce and remove them. This poor knowledge significantly threatens the release of AI-CSs detectors aimed at improving the system's quality. Researchers and practitioners cannot define crucial aspects of smell detection and refactoring, such as (i) how often the detection should be performed *e.g.*, commit-by-commit or

[1]https://chat.openai.com/

Mining Software Repository (MSR) - (Registered-Report)

Investigate Multiple Code Quality Attributes Over Time

Defect Proneness and Maintenance Effort

On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

Giammaria Giordano · Gerardo Festa · Gemma Catolino · Fabio Palomba · Filomena Ferrucci · Carmine Gravino

Received: date / Accepted: date

Abstract Software reusability mechanisms, like inheritance and delegation in Object-Oriented programming, are widely recognized as key instruments of software design that reduce the risks of source code being affected by defects, other than to reduce the effort required to maintain and evolve source code. Previous work has traditionally employed source code reuse metrics for prediction purposes, e.g., in the context of defect prediction. However, our research identifies two noticeable limitations of the current literature. First, still little is known about the extent to which developers actually employ code reuse mechanisms over time. Second, it is still unclear how these mechanisms may contribute to explaining defect-proneness and maintenance effort during software evolution. We aim at bridging this gap of knowledge, as an improved understanding of these aspects might provide insights into the actual support provided by these mechanisms, e.g., by suggesting whether and how to use them for prediction purposes. We propose an exploratory study, conducted on 12 Java projects—over 44,900 commits—of the Defects4J dataset, aiming at (1) assessing how developers use inheritance and delegation during software evolution; and (2) statistically analyzing the impact of inheritance and delegation on fault proneness and maintenance effort. Our results let emerge various usage patterns that describe the way inheritance and delegation vary over time. In addition, we find out that inheritance and delegation are statistically significant factors that influence both source code defect-proneness and maintenance effort.

Keywords Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.

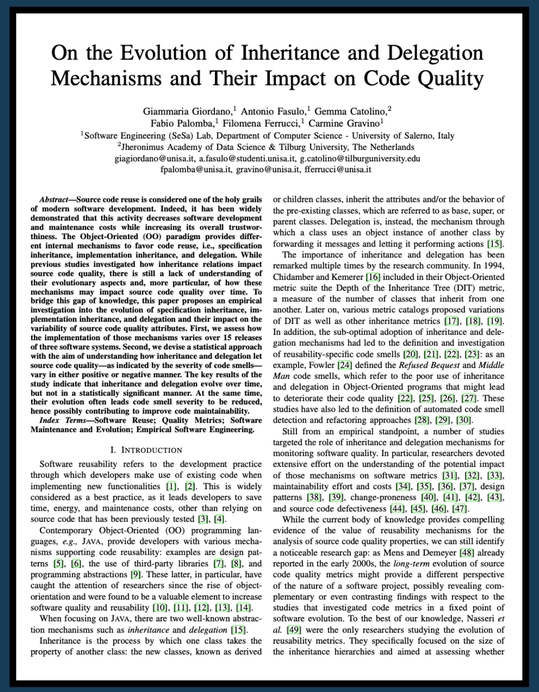
Giammaria Giordano, Gerardo Festa, Fabio Palomba, Filomena Ferrucci, Carmine Gravino
Software Engineering (SeSa) Lab - University of Salerno (Italy) — E-mail: giagiordano@unisa.it, g.festa22@studenti.unisa.it, fpalomba@unisa.it, fferucci@unisa.it

Gemma Catolino
Jheronimus Academy of Data Science & Tilburg University, The Netherlands — E-mail: g.catolino@tilburguniversity.edu

Empirical Software Engineering (EMSE)

Multiple Code Instruments

Inheritance and Delegation



On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Giammaria Giordano,[1] Antonio Fasulo,[1] Gemma Catolino,[2]
Fabio Palomba,[1] Filomena Ferrucci,[1] Carmine Gravino[1]
[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
giagiordano@unisa.it, a.fasulo@studenti.unisa.it, g.catolino@tilburguniversity.edu
fpalomba@unisa.it, gravino@unisa.it, fferrucci@unisa.it

***Abstract*—Source code reuse is considered one of the holy grails of modern software development. Indeed, it has been widely demonstrated that this activity decreases software development and maintenance costs while increasing its overall trustworthiness. The Object-Oriented (OO) paradigm provides different internal mechanisms to favor code reuse, i.e., specification inheritance, implementation inheritance, and delegation. While previous studies investigated how inheritance relations impact source code quality, there is still a lack of understanding of their evolutionary aspects and, more particular, of how these mechanisms may impact source code quality over time. To bridge this gap of knowledge, this paper proposes an empirical investigation into the evolution of specification inheritance, implementation inheritance, and delegation and their impact on the variability of source code quality attributes. First, we assess how the implementation of those mechanisms varies over 15 releases of three software systems. Second, we devise a statistical approach with the aim of understanding how inheritance and delegation let source code quality—as indicated by the severity of code smells—vary in either positive or negative manner. The key results of the study indicate that inheritance and delegation evolve over time, but not in a statistically significant manner. At the same time, their evolution often leads code smell severity to be reduced, hence possibly contributing to improve code maintainability.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.**

I. Introduction

Software reusability refers to the development practice through which developers make use of existing code when implementing new functionalities [1], [2]. This is widely considered as a best practice, as it leads developers to save time, energy, and maintenance costs, other than relying on source code that has been previously tested [3], [4].

Contemporary Object-Oriented (OO) programming languages, *e.g.*, Java, provide developers with various mechanisms supporting code reusability: examples are design patterns [5], [6], the use of third-party libraries [7], [8], and programming abstractions [9]. These latter, in particular, have caught the attention of researchers since the rise of object-orientation and were found to be a valuable element to increase software quality and reusability [10], [11], [12], [13], [14].

When focusing on Java, there are two well-known abstraction mechanisms such as *inheritance* and *delegation* [15].

Inheritance is the process by which one class takes the property of another class: the new classes, known as derived or children classes, inherit the attributes and/or the behavior of the pre-existing classes, which are referred to as base, super, or parent classes. Delegation is, instead, the mechanism through which a class uses an object instance of another class by forwarding it messages and letting it performing actions [15].

The importance of inheritance and delegation has been remarked multiple times by the research community. In 1994, Chidamber and Kemerer [16] included in their Object-Oriented metric suite the Depth of the Inheritance Tree (DIT) metric, a measure of the number of classes that inherit from one another. Later on, various metric catalogs proposed variations of DIT as well as other inheritance metrics [17], [18], [19]. In addition, the sub-optimal adoption of inheritance and delegation mechanisms had led to the definition and investigation of reusability-specific code smells [20], [21], [22], [23]: as an example, Fowler [24] defined the *Refused Bequest* and *Middle Man* code smells, which refer to the poor use of inheritance and delegation in Object-Oriented programs that might lead to deteriorate their code quality [22], [25], [26], [27]. These studies have also led to the definition of automated code smell detection and refactoring approaches [28], [29], [30].

Still from an empirical standpoint, a number of studies targeted the role of inheritance and delegation mechanisms for monitoring software quality. In particular, researchers devoted extensive effort on the understanding of the potential impact of those mechanisms on software metrics [31], [32], [33], maintainability effort and costs [34], [35], [36], [37], design patterns [38], [39], change-proneness [40], [41], [42], [43], and source code defectiveness [44], [45], [46], [47].

While the current body of knowledge provides compelling evidence of the value of reusability mechanisms for the analysis of source code quality properties, we can still identify a noticeable research gap: as Mens and Demeyer [48] already reported in the early 2000s, the *long-term* evolution of source code quality metrics might provide a different perspective of the nature of a software project, possibly revealing complementary or even contrasting findings with respect to the studies that investigated code metrics in a fixed point of software evolution. To the best of our knowledge, Nasseri *et al.* [49] were the only researchers studying the evolution of reusability metrics. They specifically focused on the size of the inheritance hierarchies and aimed at assessing whether

Software analysis, evolution, and Reengineering (SANER)

On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

Giammaria Giordano · Gerardo Festa · Gemma Catolino · Fabio Palomba · Filomena Ferrucci · Carmine Gravino

Received: date / Accepted: date

Abstract Software reusability mechanisms, like inheritance and delegation in Object-Oriented programming, are widely recognized as key instruments of software design that reduce the risks of source code being affected by defects, other than to reduce the effort required to maintain and evolve source code. Previous work has traditionally employed source code reuse metrics for prediction purposes, e.g., in the context of defect prediction. However, our research identifies two noticeable limitations of the current literature. First, still little is known about the extent to which developers actually employ code reuse mechanisms over time. Second, it is still unclear how these mechanisms may contribute to explaining defect-proneness and maintenance effort during software evolution. We aim at bridging this gap of knowledge, as an improved understanding of these aspects might provide insights into the actual support provided by these mechanisms, e.g., by suggesting whether and how to use them for prediction purposes. We propose an exploratory study, conducted on 12 Java projects—over 44,900 commits—of the Defects4J dataset, aiming at (1) assessing how developers use inheritance and delegation during software evolution; and (2) statistically analyzing the impact of inheritance and delegation on fault proneness and maintenance effort. Our results let emerge various usage patterns that describe the way inheritance and delegation vary over time. In addition, we find out that inheritance and delegation are statistically significant factors that influence both source code defect-proneness and maintenance effort.

Keywords Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.

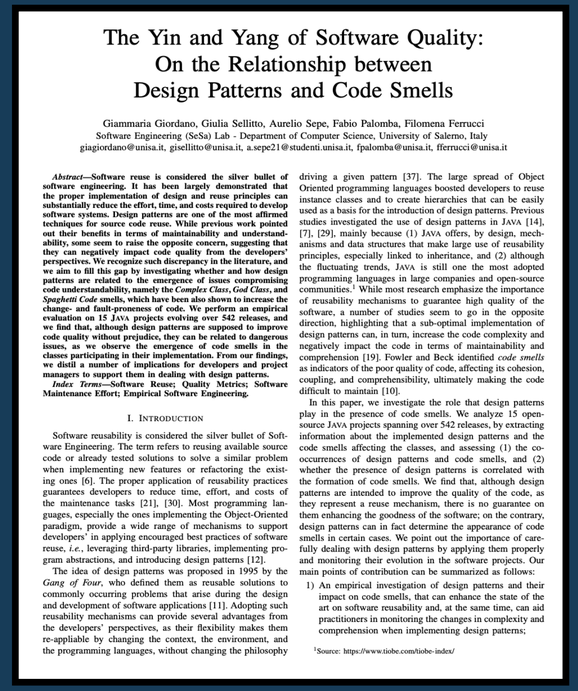
Giammaria Giordano, Gerardo Festa, Fabio Palomba, Filomena Ferrucci, Carmine Gravino
Software Engineering (SeSa) Lab - University of Salerno (Italy) — E-mail: giagiordano@unisa.it, g.festa22@studenti.unisa.it, fpalomba@unisa.it, fferucci@unisa.it

Gemma Catolino
Jheronimus Academy of Data Science & Tilburg University, The Netherlands — E-mail: g.catolino@tilburguniversity.edu

Empirical Software Engineering (EMSE)

Multiple Code Instruments

Design Patterns



The Yin and Yang of Software Quality:
On the Relationship between
Design Patterns and Code Smells

Giammaria Giordano, Giulia Sellitto, Aurelio Sepe, Fabio Palomba, Filomena Ferrucci
Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy
giagiordano@unisa.it, gisellitto@unisa.it, a.sepe21@studenti.unisa.it, fpalomba@unisa.it, fferrucci@unisa.it

***Abstract*—Software reuse is considered the silver bullet of software engineering. It has been largely demonstrated that the proper implementation of design and reuse principles can substantially reduce the effort, time, and costs required to develop software systems. Design patterns are one of the most affirmed techniques for source code reuse. While previous work pointed out their benefits in terms of maintainability and understandability, some seem to raise the opposite concern, suggesting that they can negatively impact code quality from the developers' perspectives. We recognize such discrepancy in the literature, and we aim to fill this gap by investigating whether and how design patterns are related to the emergence of issues compromising code understandability, namely the *Complex Class*, *God Class*, and *Spaghetti Code* smells, which have been also shown to increase the change- and fault-proneness of code. We perform an empirical evaluation on 15 JAVA projects evolving over 542 releases, and we find that, although design patterns are supposed to improve code quality without prejudice, they can be related to dangerous issues, as we observe the emergence of code smells in the classes participating in their implementation. From our findings, we distil a number of implications for developers and project managers to support them in dealing with design patterns.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance Effort; Empirical Software Engineering.**

I. INTRODUCTION

Software reusability is considered the silver bullet of Software Engineering. The term refers to reusing available source code or already tested solutions to solve a similar problem when implementing new features or refactoring the existing ones [6]. The proper application of reusability practices guarantees developers to reduce time, effort, and costs of the maintenance tasks [21], [30]. Most programming languages, especially the ones implementing the Object-Oriented paradigm, provide a wide range of mechanisms to support developers' in applying encouraged best practices of software reuse, *i.e.*, leveraging third-party libraries, implementing program abstractions, and introducing design patterns [12].

The idea of design patterns was proposed in 1995 by the *Gang of Four*, who defined them as reusable solutions to commonly occurring problems that arise during the design and development of software applications [11]. Adopting such reusability mechanisms can provide several advantages from the developers' perspectives, as their flexibility makes them re-appliable by changing the context, the environment, and the programming languages, without changing the philosophy driving a given pattern [37]. The large spread of Object Oriented programming languages boosted developers to reuse instance classes and to create hierarchies that can be easily used as a basis for the introduction of design patterns. Previous studies investigated the use of design patterns in JAVA [14], [7], [29], mainly because (1) JAVA offers, by design, mechanisms and data structures that make large use of reusability principles, especially linked to inheritance, and (2) although the fluctuating trends, JAVA is still one the most adopted programming languages in large companies and open-source communities.[1] While most research emphasize the importance of reusability mechanisms to guarantee high quality of the software, a number of studies seem to go in the opposite direction, highlighting that a sub-optimal implementation of design patterns can, in turn, increase the code complexity and negatively impact the code in terms of maintainability and comprehension [19]. Fowler and Beck identified *code smells* as indicators of the poor quality of code, affecting its cohesion, coupling, and comprehensibility, ultimately making the code difficult to maintain [10].

In this paper, we investigate the role that design patterns play in the presence of code smells. We analyze 15 open-source JAVA projects spanning over 542 releases, by extracting information about the implemented design patterns and the code smells affecting the classes, and assessing (1) the co-occurrences of design patterns and code smells, and (2) whether the presence of design patterns is correlated with the formation of code smells. We find that, although design patterns are intended to improve the quality of the code, as they represent a reuse mechanism, there is no guarantee on them enhancing the goodness of the software; on the contrary, design patterns can in fact determine the appearance of code smells in certain cases. We point out the importance of carefully dealing with design patterns by applying them properly and monitoring their evolution in the software projects. Our main points of contribution can be summarized as follows:

1) An empirical investigation of design patterns and their impact on code smells, that can enhance the state of the art on software reusability and, at the same time, can aid practitioners in monitoring the changes in complexity and comprehension when implementing design patterns;

[1]Source: https://www.tiobe.com/tiobe-index/

Euromicro SEAA 2023

Multiple Code Instruments

Built-in Features

Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study

Giammaria Giordano[1], Giusy Annunziata[1], Andrea De Lucia[1] and Fabio Palomba[1]

[1]*University of Salerno (Italy) - SeSa Lab*

Abstract

To deal with continuous change requests and the strict time-to-market, practitioners and big companies constantly update their software systems to meet users' requirements. This practice force developers to release immature products, neglecting best practices to reduce delivery times. As a possible result, *technical debt* can arise, *i.e.*, potential design issues that can negatively impact software maintenance and evolution and, in turn, increase both the time-to-market and costs. *Code smells*—sub-optimal design decisions identifiable by computing software metrics and providing a general overview of code quality —are common symptoms of technical debt. While previous research focused on code smells primarily considering them in the context of Java, the growing popularity of Python, particularly for developing artificial intelligence (AI)-Enabled systems, calls for additional investigations. This preliminary analysis addresses this gap by exploring the diffusion of Python-specific code smells, and the activities performed by developers that induce the introduction of code smells in their systems. To perform our preliminary investigation, we selected 200 AI-Enabled systems available in the Niche dataset; We extracted 10,611 information on the releases using PyDriller, and PySmell to extract information about code smells. The results reveal several insights: 1) Code smells related to object-oriented principles are rarely detected in Python; 2) Complex List Comprehension is the most prevalent and the most long-alive smell; 3) The main activities that can induce code smells are evolutionary. This study fills a critical gap in the literature by providing empirical evidence on the evolution of code smells in Python-based AI-enabled systems.

Keywords

Software Maintenance, Software Evolution, Software Refactoring, Code Smells,

1. Introduction

To meet customers' needs and due to the continuous environmental changes, practitioners and big companies update their source code as fast as possible [1]. The continuous *change requests* and the stringent *time-to-market* force developers to release immature products, putting aside best practices to decrease the delivery time [2, 3]. As a possible output of this process could be the introduction of *technical debt* [4]—*i.e.*, potential design issues that can negatively impact the software system during maintenance and evolution. One of the *symptoms* of technical debt is

IWSM/MENSURA 23, September 14–15, 2023, Rome, Italy

giagiordano@unisa.it (G. Giordano); gannunziata@unisa.it (G. Annunziata); adelucia@unisa.it (A. D. Lucia); fpalomba@unisa.it (F. Palomba)

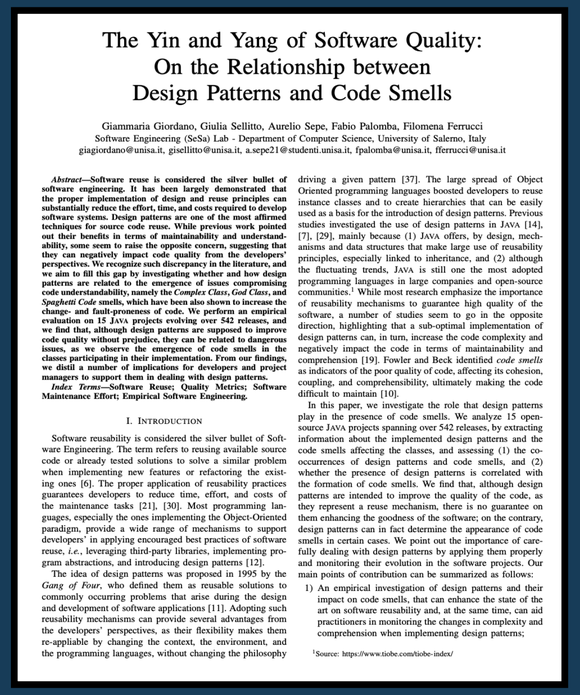
https://giammariagiordano.github.io/giammaria-giordano/ (G. Giordano); https://giusyann.github.io/ (G. Annunziata); https://docenti.unisa.it/003241/home (A. D. Lucia); https://fpalomba.github.io/ (F. Palomba)

0000-0003-2567-440X (G. Giordano); 0009-0002-0742-7261 (G. Annunziata); 0000-0002-4238-1425 (A. D. Lucia); 0000-0001-9337-5116 (F. Palomba)

International Conference on Software Process and Product Measurement (MENSURA)

Multiple Families of Software Systems

Traditional Systems



The Yin and Yang of Software Quality: On the Relationship between Design Patterns and Code Smells

Giammaria Giordano, Giulia Sellitto, Aurelio Sepe, Fabio Palomba, Filomena Ferrucci
Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy
giagiordano@unisa.it, gisellitto@unisa.it, a.sepe21@studenti.unisa.it, fpalomba@unisa.it, fferrucci@unisa.it

***Abstract*—Software reuse is considered the silver bullet of software engineering. It has been largely demonstrated that the proper implementation of design and reuse principles can substantially reduce the effort, time, and costs required to develop software systems. Design patterns are one of the most affirmed techniques for source code reuse. While previous work pointed out their benefits in terms of maintainability and understandability, some seem to raise the opposite concern, suggesting that they can negatively impact code quality from the developers' perspectives. We recognize such discrepancy in the literature, and we aim to fill this gap by investigating whether and how design patterns are related to the emergence of issues compromising code understandability, namely the *Complex Class*, *God Class*, and *Spaghetti Code* smells, which have been also shown to increase the change- and fault-proneness of code. We perform an empirical evaluation on 15 Java projects evolving over 542 releases, and we find that, although design patterns are supposed to improve code quality without prejudice, they can be related to dangerous issues, as we observe the emergence of code smells in the classes participating in their implementation. From our findings, we distill a number of implications for developers and project managers to support them in dealing with design patterns.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance Effort; Empirical Software Engineering.**

I. Introduction

Software reusability is considered the silver bullet of Software Engineering. The term refers to reusing available source code or already tested solutions to solve a similar problem when implementing new features or refactoring the existing ones [6]. The proper application of reusability practices guarantees developers to reduce time, effort, and costs of the maintenance tasks [21], [30]. Most programming languages, especially the ones implementing the Object-Oriented paradigm, provide a wide range of mechanisms to support developers' in applying encouraged best practices of software reuse, *i.e.*, leveraging third-party libraries, implementing program abstractions, and introducing design patterns [12].

The idea of design patterns was proposed in 1995 by the *Gang of Four*, who defined them as reusable solutions to commonly occurring problems that arise during the design and development of software applications [11]. Adopting such reusability mechanisms can provide several advantages from the developers' perspectives, as their flexibility makes them re-appliable by changing the context, the environment, and the programming languages, without changing the philosophy driving a given pattern [37]. The large spread of Object Oriented programming languages boosted developers to reuse instance classes and to create hierarchies that can be easily used as a basis for the introduction of design patterns. Previous studies investigated the use of design patterns in Java [14], [7], [29], mainly because (1) Java offers, by design, mechanisms and data structures that make large use of reusability principles, especially linked to inheritance, and (2) although the fluctuating trends, Java is still one the most adopted programming languages in large companies and open-source communities.[1] While most research emphasize the importance of reusability mechanisms to guarantee high quality of the software, a number of studies seem to go in the opposite direction, highlighting that a sub-optimal implementation of design patterns can, in turn, increase the code complexity and negatively impact the code in terms of maintainability and comprehension [19]. Fowler and Beck identified *code smells* as indicators of the poor quality of code, affecting its cohesion, coupling, and comprehensibility, ultimately making the code difficult to maintain [10].

In this paper, we investigate the role that design patterns play in the presence of code smells. We analyze 15 open-source Java projects spanning over 542 releases, by extracting information about the implemented design patterns and the code smells affecting the classes, and assessing (1) the co-occurrences of design patterns and code smells, and (2) whether the presence of design patterns is correlated with the formation of code smells. We find that, although design patterns are intended to improve the quality of the code, as they represent a reuse mechanism, there is no guarantee on them enhancing the goodness of the software; on the contrary, design patterns can in fact determine the appearance of code smells in certain cases. We point out the importance of carefully dealing with design patterns by applying them properly and monitoring their evolution in the software projects. Our main points of contribution can be summarized as follows:

1) An empirical investigation of design patterns and their impact on code smells, that can enhance the state of the art on software reusability and, at the same time, can aid practitioners in monitoring the changes in complexity and comprehension when implementing design patterns;

[1]Source: https://www.tiobe.com/tiobe-index/

Euromicro SEAA 2023

On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

Giammaria Giordano · Gerardo Festa · Gemma Catolino · Fabio Palomba · Filomena Ferrucci · Carmine Gravino

Received: date / Accepted: date

Abstract Software reusability mechanisms, like inheritance and delegation in Object-Oriented programming, are widely recognized as key instruments of software design that reduce the risks of source code being affected by defects, other than to reduce the effort required to maintain and evolve source code. Previous work has traditionally employed source code reuse metrics for prediction purposes, e.g., in the context of defect prediction. However, our research identifies two noticeable limitations of the current literature. First, still little is known about the extent to which developers actually employ code reuse mechanisms over time. Second, it is still unclear how these mechanisms may contribute to explaining defect-proneness and maintenance effort during software evolution. We aim at bridging this gap of knowledge, as an improved understanding of these aspects might provide insights into the actual support provided by these mechanisms, e.g., by suggesting whether and how to use them for prediction purposes. We propose an exploratory study, conducted on 12 Java projects—over 44,900 commits—of the Defects4J dataset, aiming at (1) assessing how developers use inheritance and delegation during software evolution; and (2) statistically analyzing the impact of inheritance and delegation on fault proneness and maintenance effort. Our results let emerge various usage patterns that describe the way inheritance and delegation vary over time. In addition, we find out that inheritance and delegation are statistically significant factors that influence both source code defect-proneness and maintenance effort.

Keywords Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.

Giammaria Giordano, Gerardo Festa, Fabio Palomba, Filomena Ferrucci, Carmine Gravino
Software Engineering (SeSa) Lab - University of Salerno (Italy) — E-mail: giagiordano@unisa.it, g.festa22@studenti.unisa.it, fpalomba@unisa.it, fferucci@unisa.it

Gemma Catolino
Jheronimus Academy of Data Science & Tilburg University, The Netherlands — E-mail: g.catolino@tilburguniversity.edu

Empirical Software Engineering (EMSE)

On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Giammaria Giordano,[1] Antonio Fasulo,[1] Gemma Catolino,[2]
Fabio Palomba,[1] Filomena Ferrucci,[1] Carmine Gravino[1]
[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
giagiordano@unisa.it, a.fasulo@studenti.unisa.it, g.catolino@tilburguniversity.edu
fpalomba@unisa.it, gravino@unisa.it, fferrucci@unisa.it

***Abstract*—Source code reuse is considered one of the holy grails of modern software development. Indeed, it has been widely demonstrated that this activity decreases software development and maintenance costs while increasing its overall trustworthiness. The Object-Oriented (OO) paradigm provides different internal mechanisms to favor code reuse, i.e., specification inheritance, implementation inheritance, and delegation. While previous studies investigated how inheritance relations impact source code quality, there is still a lack of understanding of their evolutionary aspects and, more particular, of how these mechanisms may impact source code quality over time. To bridge this gap of knowledge, this paper proposes an empirical investigation into the evolution of specification inheritance, implementation inheritance, and delegation and their impact on the variability of source code quality attributes. First, we assess how the implementation of those mechanisms varies over 15 releases of three software systems. Second, we devise a statistical approach with the aim of understanding how inheritance and delegation let source code quality—as indicated by the severity of code smells—vary in either positive or negative manner. The key results of the study indicate that inheritance and delegation evolve over time, but not in a statistically significant manner. At the same time, their evolution often leads code smell severity to be reduced, hence possibly contributing to improve code maintainability.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.**

I. Introduction

Software reusability refers to the development practice through which developers make use of existing code when implementing new functionalities [1], [2]. This is widely considered as a best practice, as it leads developers to save time, energy, and maintenance costs, other than relying on source code that has been previously tested [3], [4].

Contemporary Object-Oriented (OO) programming languages, *e.g.*, Java, provide developers with various mechanisms supporting code reusability: examples are design patterns [5], [6], the use of third-party libraries [7], [8], and programming abstractions [9]. These latter, in particular, have caught the attention of researchers since the rise of object-orientation and were found to be a valuable element to increase software quality and reusability [10], [11], [12], [13], [14].

When focusing on Java, there are two well-known abstraction mechanisms such as *inheritance* and *delegation* [15].

Inheritance is the process by which one class takes the property of another class: the new classes, known as derived or children classes, inherit the attributes and/or the behavior of the pre-existing classes, which are referred to as base, super, or parent classes. Delegation is, instead, the mechanism through which a class uses an object instance of another class by forwarding it messages and letting it performing actions [15].

The importance of inheritance and delegation has been remarked multiple times by the research community. In 1994, Chidamber and Kemerer [16] included in their Object-Oriented metric suite the Depth of the Inheritance Tree (DIT) metric, a measure of the number of classes that inherit from one another. Later on, various metric catalogs proposed variations of DIT as well as other inheritance metrics [17], [18], [19]. In addition, the sub-optimal adoption of inheritance and delegation mechanisms had led to the definition and investigation of reusability-specific code smells [20], [21], [22], [23]: as an example, Fowler [24] defined the *Refused Bequest* and *Middle Man* code smells, which refer to the poor use of inheritance and delegation in Object-Oriented programs that might lead to deteriorate their code quality [22], [25], [26], [27]. These studies have also led to the definition of automated code smell detection and refactoring approaches [28], [29], [30].

Still from an empirical standpoint, a number of studies targeted the role of inheritance and delegation mechanisms for monitoring software quality. In particular, researchers devoted extensive effort on the understanding of the potential impact of those mechanisms on software metrics [31], [32], [33], maintainability effort and costs [34], [35], [36], [37], design patterns [38], [39], change-proneness [40], [41], [42], [43], and source code defectiveness [44], [45], [46], [47].

While the current body of knowledge provides compelling evidence of the value of reusability mechanisms for the analysis of source code quality properties, we can still identify a noticeable research gap: as Mens and Demeyer [48] already reported in the early 2000s, the *long-term* evolution of source code quality metrics might provide a different perspective of the nature of a software project, possibly revealing complementary or even contrasting findings with respect to the studies that investigated code metrics in a fixed point of software evolution. To the best of our knowledge, Nasseri *et al.* [49] were the only researchers studying the evolution of reusability metrics. They specifically focused on the size of the inheritance hierarchies and aimed at assessing whether

Software analysis, evolution, and Reengineering (SANER)

Multiple Families of Software Systems

AI-enabled Systems

Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study

Giammaria Giordano[1], Giusy Annunziata[1], Andrea De Lucia[1] and Fabio Palomba[1]

[1]University of Salerno (Italy) - SeSa Lab

Abstract

To deal with continuous change requests and the strict time-to-market, practitioners and big companies constantly update their software systems to meet users' requirements. This practice force developers to release immature products, neglecting best practices to reduce delivery times. As a possible result, *technical debt* can arise, *i.e.*, potential design issues that can negatively impact software maintenance and evolution and, in turn, increase both the time-to-market and costs. *Code smells*—sub-optimal design decisions identifiable by computing software metrics and providing a general overview of code quality —are common symptoms of technical debt. While previous research focused on code smells primarily considering them in the context of Java, the growing popularity of Python, particularly for developing artificial intelligence (AI)-Enabled systems, calls for additional investigations. This preliminary analysis addresses this gap by exploring the diffusion of Python-specific code smells, and the activities performed by developers that induce the introduction of code smells in their systems. To perform our preliminary investigation, we selected 200 AI-Enabled systems available in the Niche dataset; We extracted 10,611 information on the releases using PyDriller, and PySmell to extract information about code smells. The results reveal several insights: 1) Code smells related to object-oriented principles are rarely detected in Python; 2) Complex List Comprehension is the most prevalent and the most long-alive smell; 3) The main activities that can induce code smells are evolutionary. This study fills a critical gap in the literature by providing empirical evidence on the evolution of code smells in Python-based AI-enabled systems.

Keywords

Software Maintenance, Software Evolution, Software Refactoring, Code Smells,

1. Introduction

To meet customers' needs and due to the continuous environmental changes, practitioners and big companies update their source code as fast as possible [1]. The continuous *change requests* and the stringent *time-to-market* force developers to release immature products, putting aside best practices to decrease the delivery time [2, 3]. As a possible output of this process could be the introduction of *technical debt* [4]—*i.e.*, potential design issues that can negatively impact the software system during maintenance and evolution. One of the *symptoms* of technical debt is

IWSM/MENSURA '23, September 14–15, 2023, Rome, Italy

giagiordano@unisa.it (G. Giordano); gannunziata@unisa.it (G. Annunziata); adelucia@unisa.it (A. D. Lucia); fpalomba@unisa.it (F. Palomba)

https://giammariagiordano.github.io/giammaria-giordano/ (G. Giordano); https://giusyann.github.io/ (G. Annunziata); https://docenti.unisa.it/003241/home (A. D. Lucia); https://fpalomba.github.io/ (F. Palomba)

0000-0003-2567-440X (G. Giordano); 0009-0002-0742-7261 (G. Annunziata); 0000-0002-4238-1425 (A. D. Lucia); 0000-0001-9337-5116 (F. Palomba)

International Conference on Software Process and Product Measurement (MENSURA)

When Code Smells Meet AI: On the Lifecycle of AI-specific Code Smells in AI-enabled Systems

Gilberto Recupito
Sesa Lab - University of Salerno
Salerno, Italy
grecupito@unisa.it

Giammaria Giordano
Sesa Lab - University of Salerno
Salerno, Italy
giagiordano@unisa.it

Filomena Ferrucci
Sesa Lab - University of Salerno
Salerno, Italy
fferrucci@unisa.it

Dario Di Nucci
Sesa Lab - University of Salerno
Salerno, Italy
ddinucci@unisa.it

Fabio Palomba
Sesa Lab - University of Salerno
Salerno, Italy
fpalomba@unisa.it

ABSTRACT

Context. The adoption of Artificial Intelligence (AI)–enabled systems is steadily increasing. Nevertheless, there is a shortage of AI-specific quality assurance approaches, possibly because of the limited knowledge of how quality-related concerns emerge and evolve in AI-enabled systems. **Objective.** We aim to investigate the emergence and evolution of specific types of quality-related concerns known as AI-specific code smells, *i.e.*, sub-optimal implementation solutions applied on AI pipelines that may significantly decrease both quality and maintainability of AI-enabled systems. More specifically, we present a plan to study AI-specific code smells by empirically analyzing (i) their prevalence in real AI-enabled systems, (ii) how they are introduced and removed, and (iii) their survivability. **Method.** We will conduct an exploratory study, mining a large dataset of AI-enabled systems and analyzing over 400k commits about 337 projects. We will track and inspect the introduction and evolution of AI smells through CodeSmile, a novel AI smell detector that we will build to enable our investigation and will detect 22 AI-specific code smells.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Technical Debt; AI-Specific Code Smells; Software Quality Assurance; Software Engineering for Artificial Intelligence.

ACM Reference Format:

Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. 2023. When Code Smells Meet AI: On the Lifecycle of AI-specific Code Smells in AI-enabled Systems. In *MSR 2024: 21st International Conference on Mining Software Repositories, April 15–16, 2024, Lisbon, ES.* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

MSR 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Artificial Intelligence (AI) evolved through the emergence of complex software integrating AI modules, defined as AI-enabled systems [13]. Self-driving cars, voice assistance instruments, or conversational agents like ChatGPT[1] are just some examples of the successful integration of AI within software engineering projects. However, the strict time-to-market and change requests pressure practitioners to roll out immature software to keep pace with competitors, leading to the possible emergence of technical debt [7] *i.e.*, a technical trade-off that can give benefits in a short period, but that can compromise the software health in the long run. *Code smells* is a manifestation of technical debt. They are symptoms of poor design and implementation choices that, if left unaddressed, can deteriorate the overall quality of the system [8].

Sculley et al. [19] showed that AI-enabled systems are incredibly prone to technical debt and code smells, raising the need for a quality assurance process for AI components. Cardozo et al. [3] and Van Oort et al. [24] argued that while the issues in those systems are emerging, there is a lack of quality assurance tools and practices that AI developers can use. This lack of quality management assets stimulates the proliferation of code smells in AI-enabled systems [12]. Consequently, given the complex nature of those systems, new types of code smells have emerged. Considering the aspects that AI developers face when dealing with AI pipelines, Zhang et al. [29] defined a new form of code smells, *AI-specific code smells* (AI-CSs). Similarly to traditional code smells, an AI-CS is defined as a *sub-optimal implementation solution for AI pipelines that may significantly decrease the quality of AI-enabled systems.* A key example of those quality issues is using a loop operation instead of exploiting the corresponding Pandas function for data handling, leading to *Unnecessary Iteration* smell [29].

While some work underlines the need to explore AI-CSs [6, 29], there is still a lack of knowledge on this type of quality issues. Among the various possible causes, we outline a lack of knowledge on how AI-CSs emerge and evolve and what motivations lead developers to introduce and remove them. This poor knowledge significantly threatens the release of AI-CSs detectors aimed at improving the system's quality. Researchers and practitioners cannot define crucial aspects of smell detection and refactoring, such as (i) how often the detection should be performed *e.g.*, commit-by-commit or

[1]https://chat.openai.com/

Mining Software Repository (MSR) - (Registered-Report)

Research Method Overview

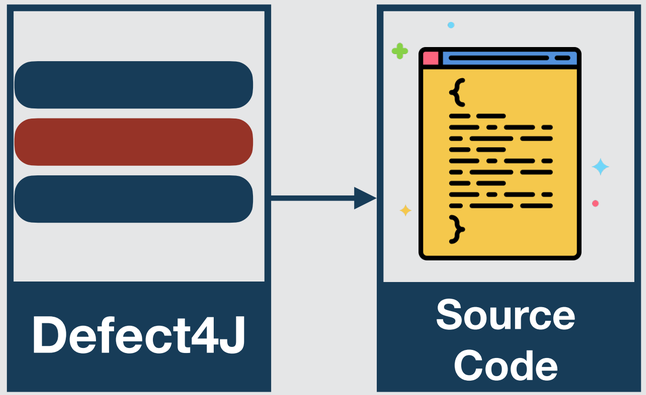
More in Detail…

Since the **crash of ARIANE** 5 was **caused** by a bug due to **incorrect code reuse** during a **maintenance** task, we will use the paper on how **reusability mechanisms affect defect proneness** and **maintenance effort** to give a general overview of how our work was conducted

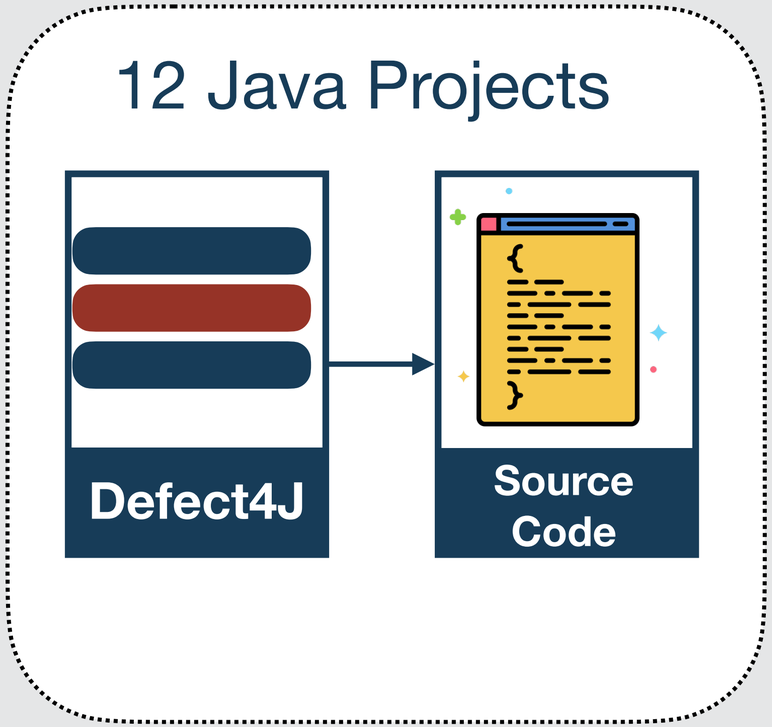
On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

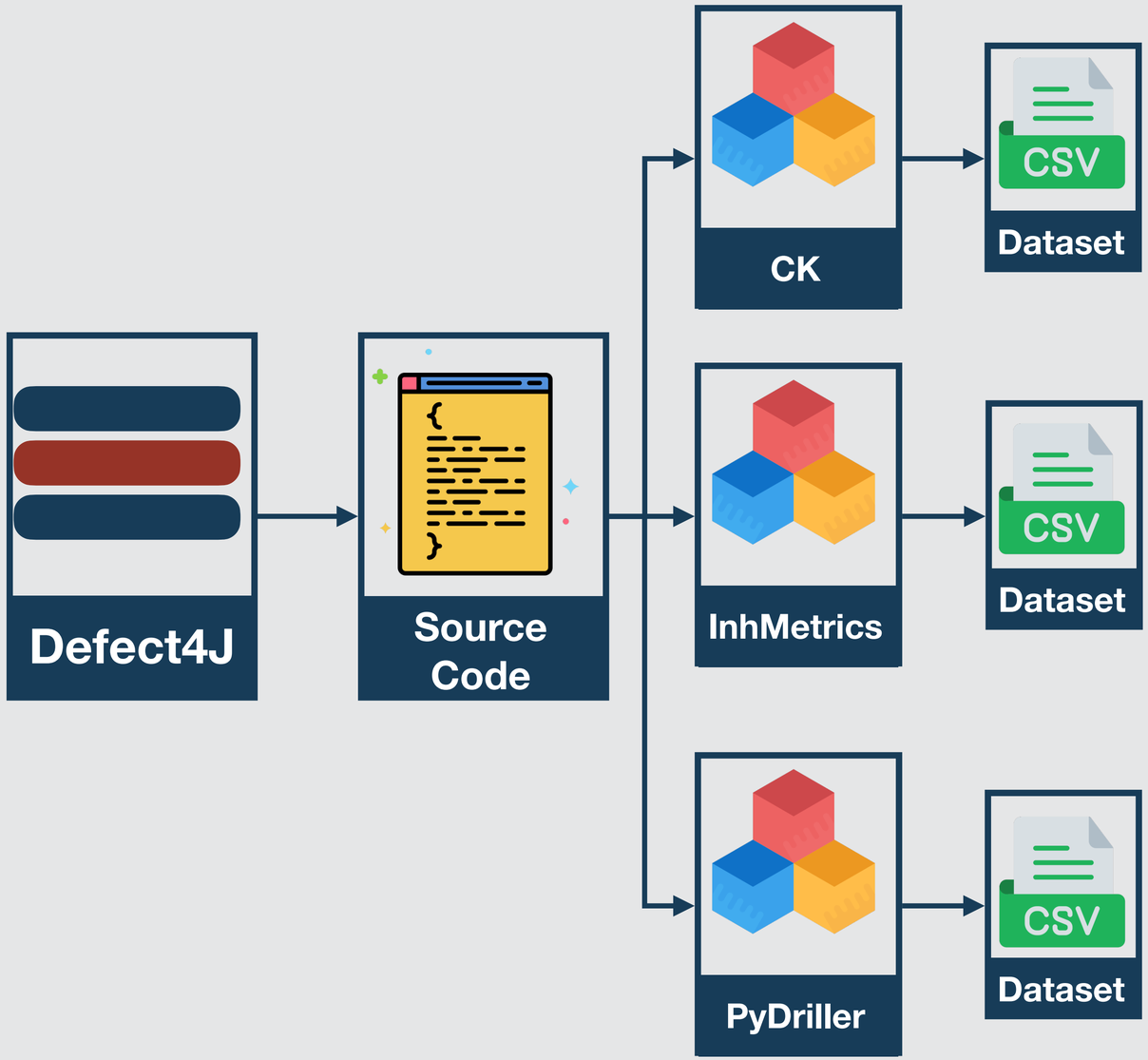
On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort



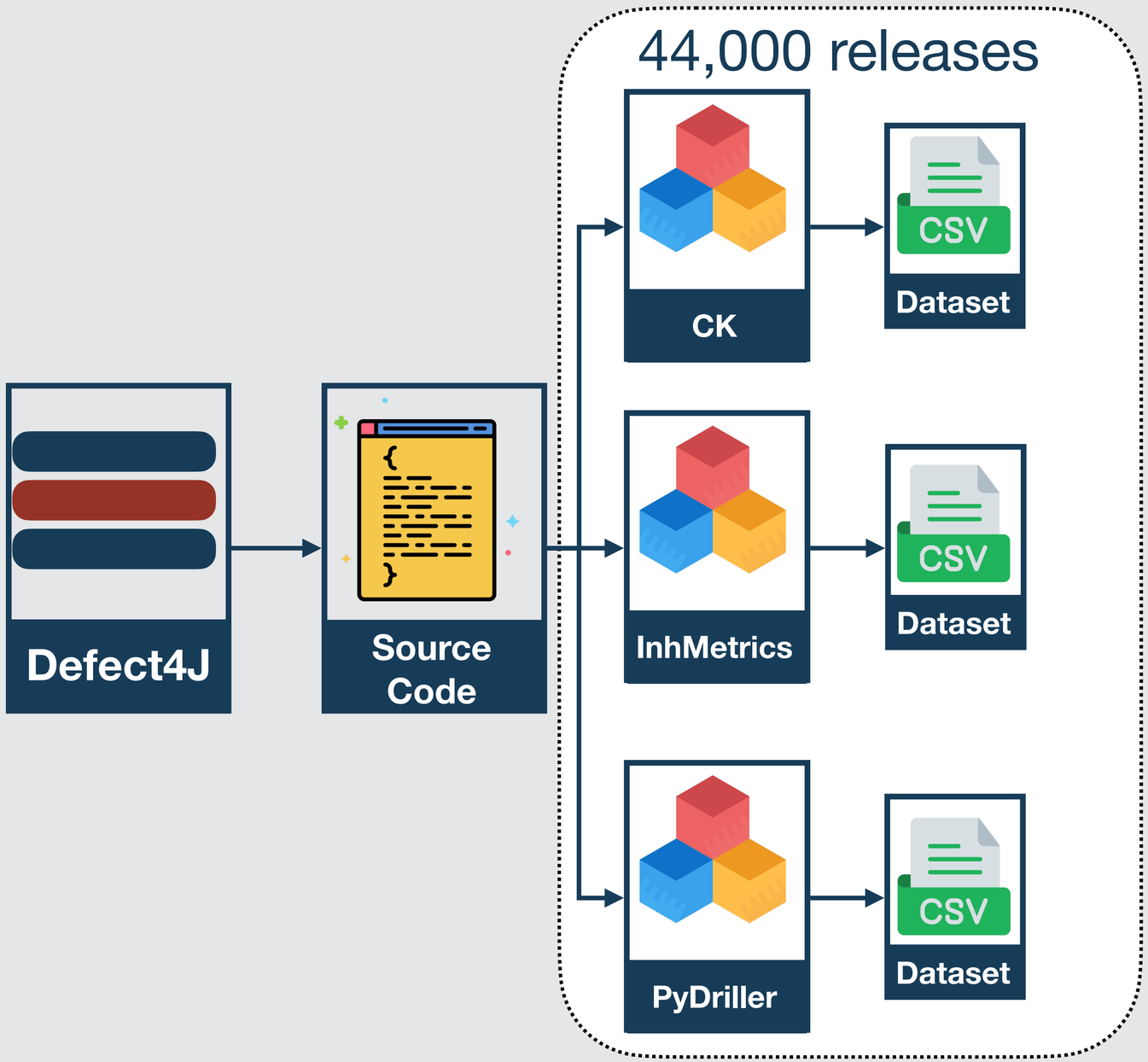
On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort



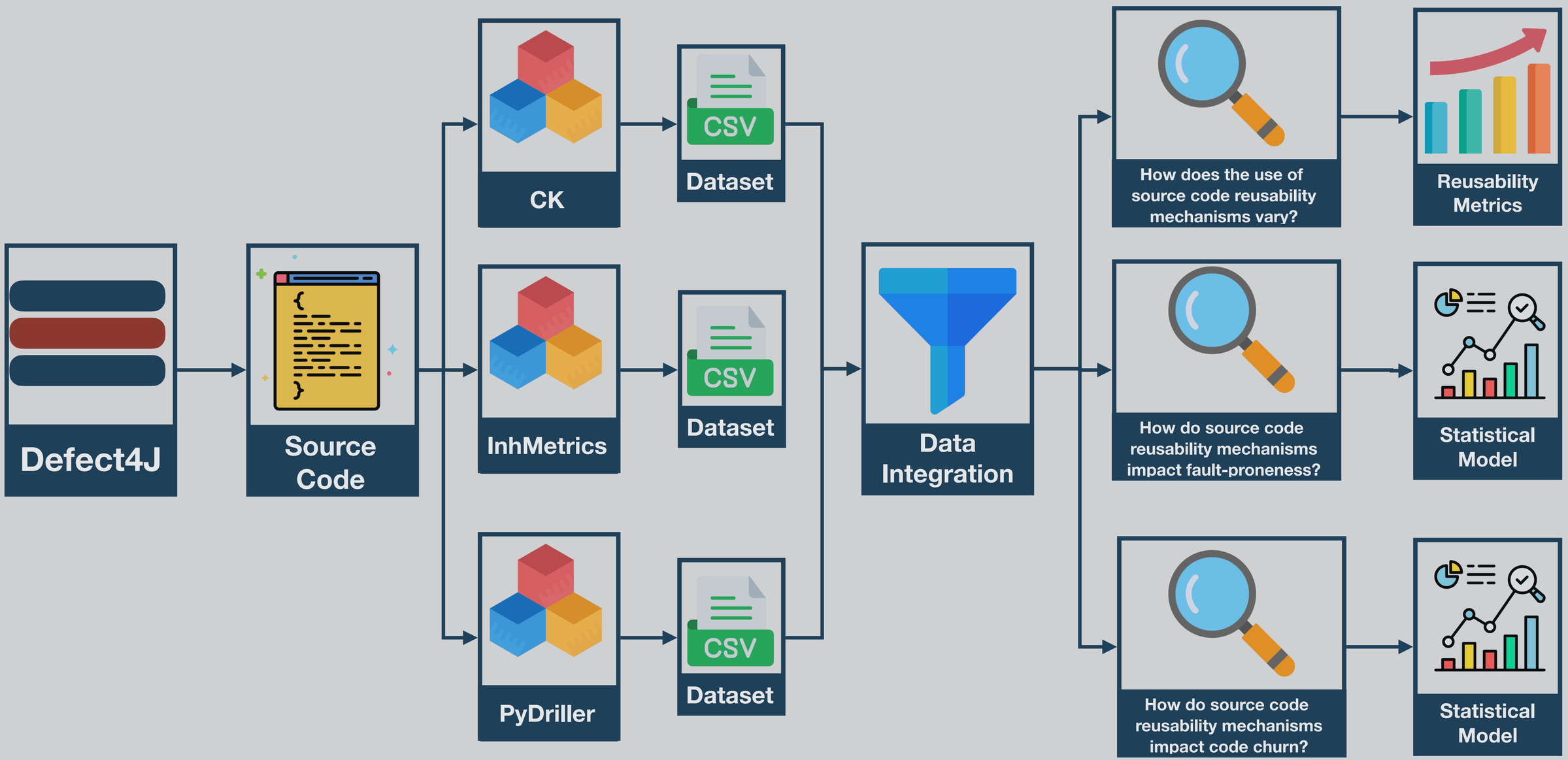
On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort



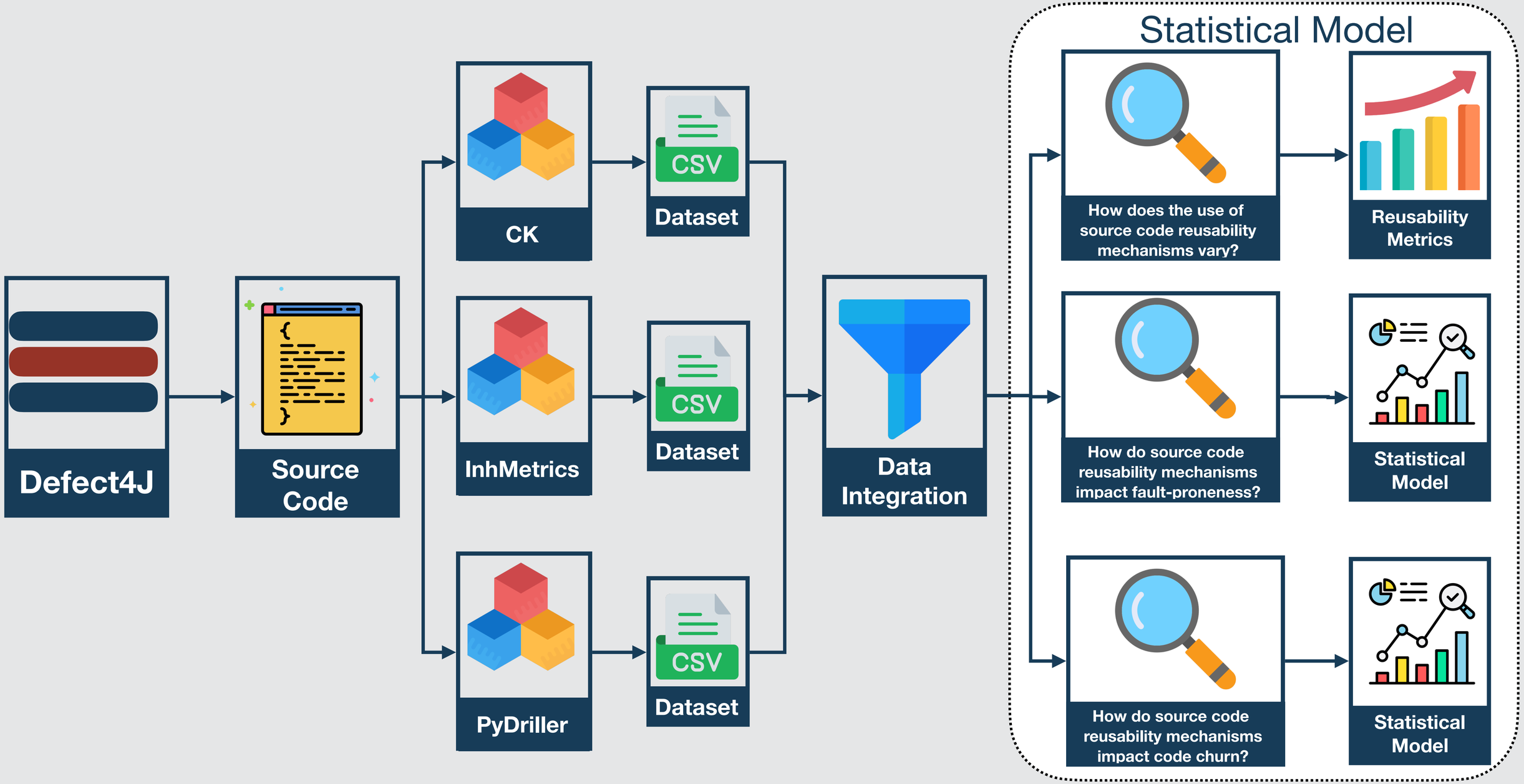
On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort



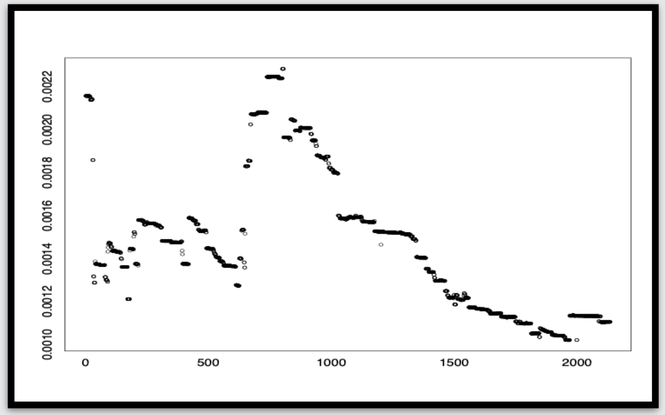
On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

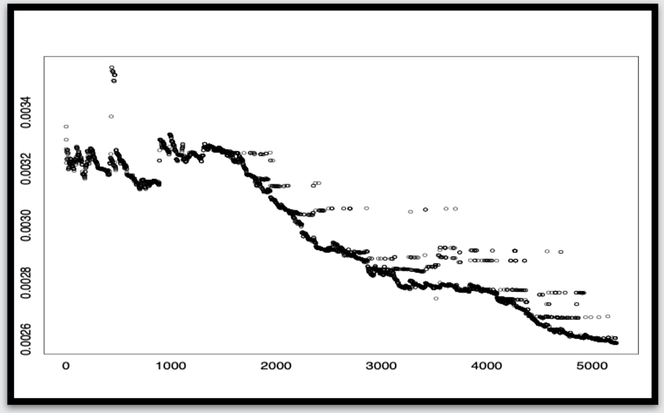


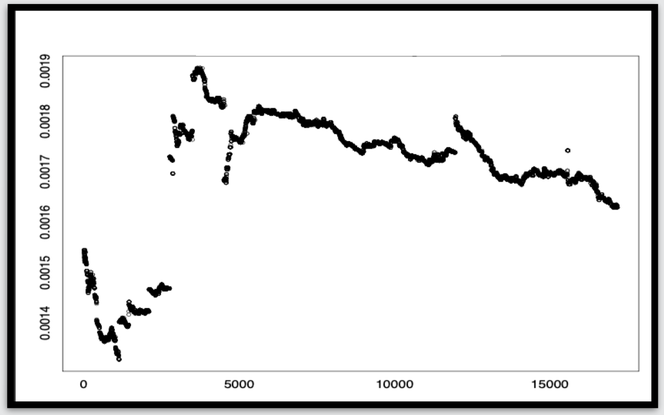
On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

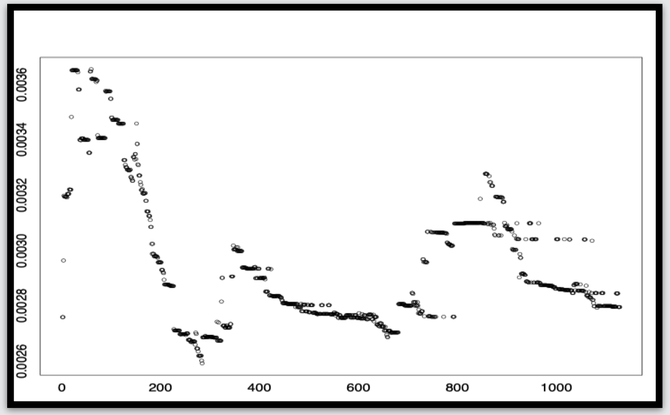


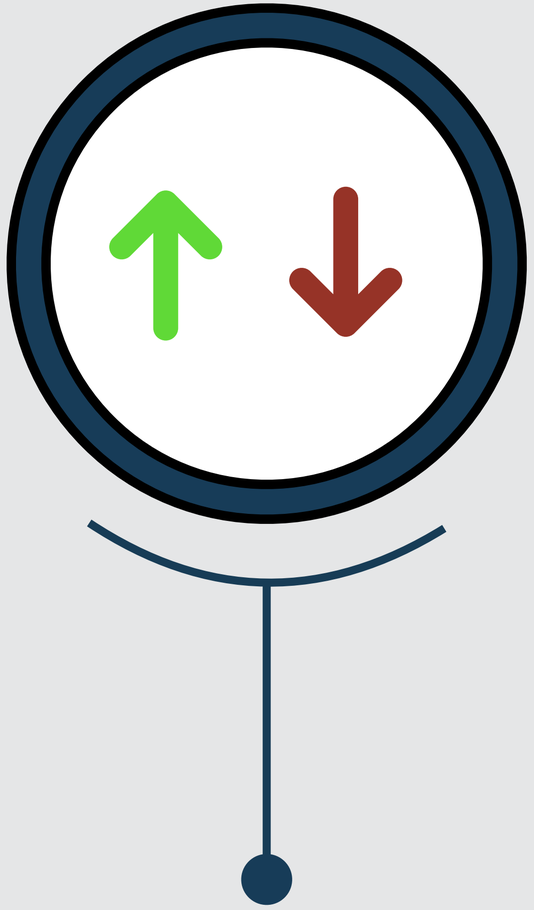
RG: Reusability Evolves Over Time





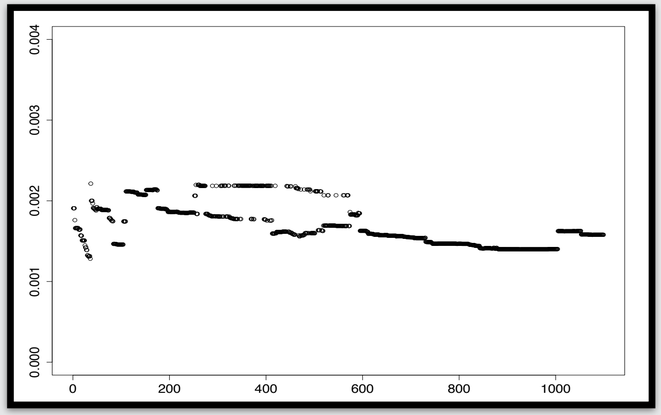


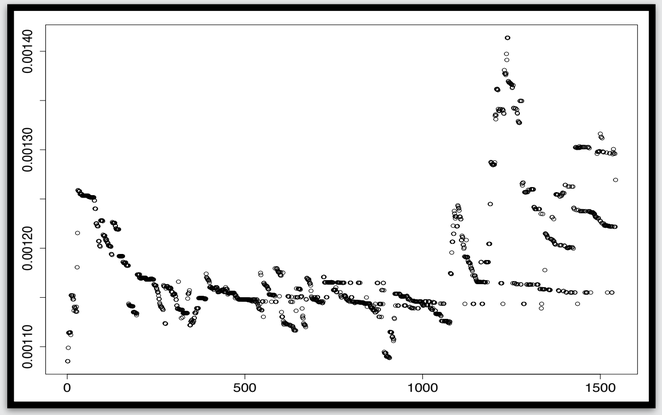


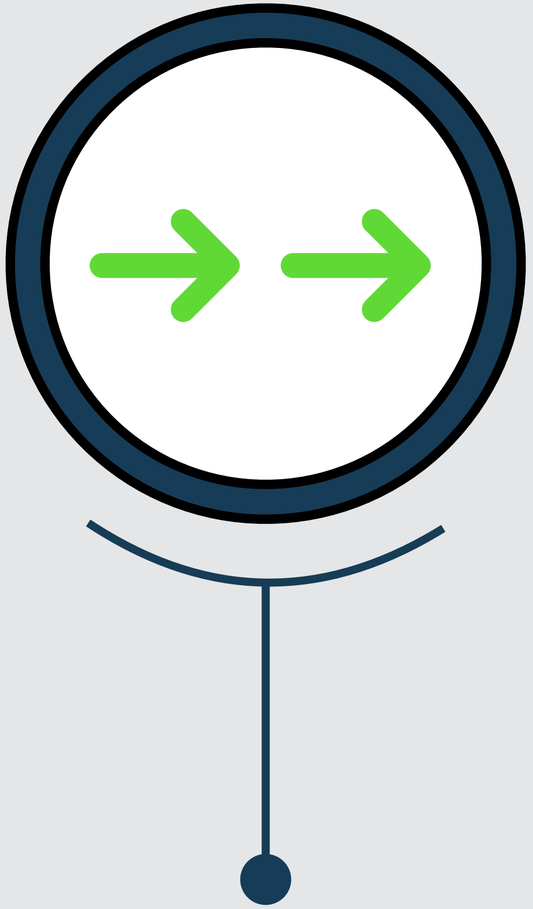


Increase/Decrease

RG: Reusability Evolves Over Time

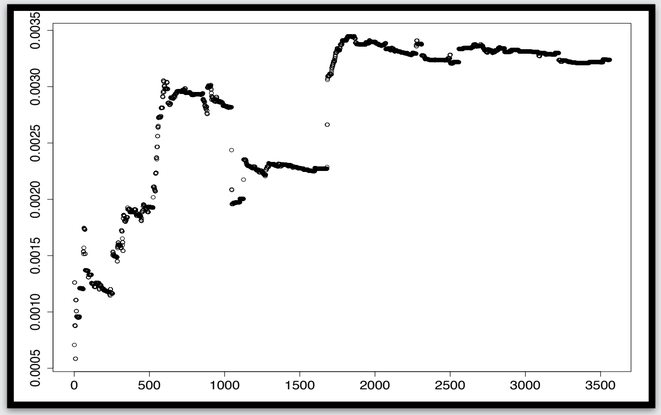


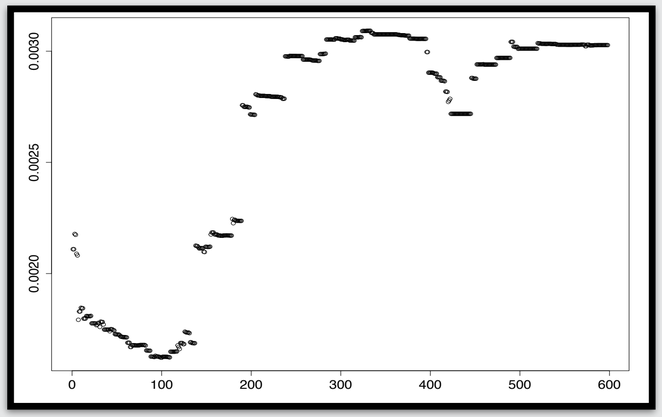


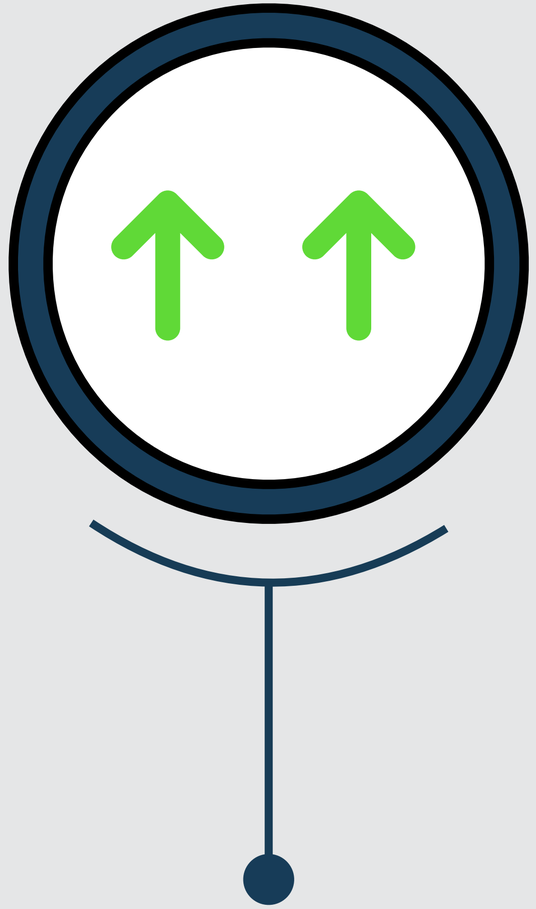


Stable

RG: Reusability Evolves Over Time

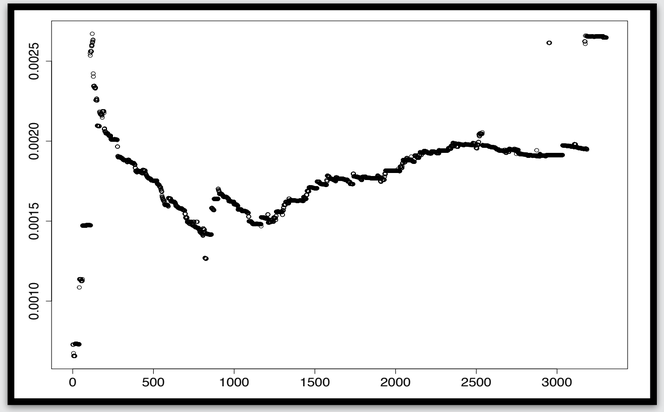


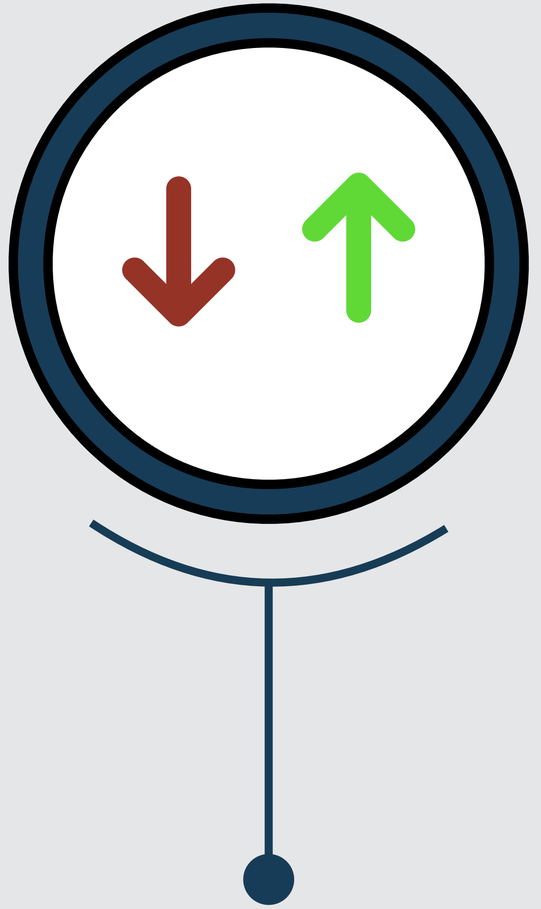




Steady-Increasing

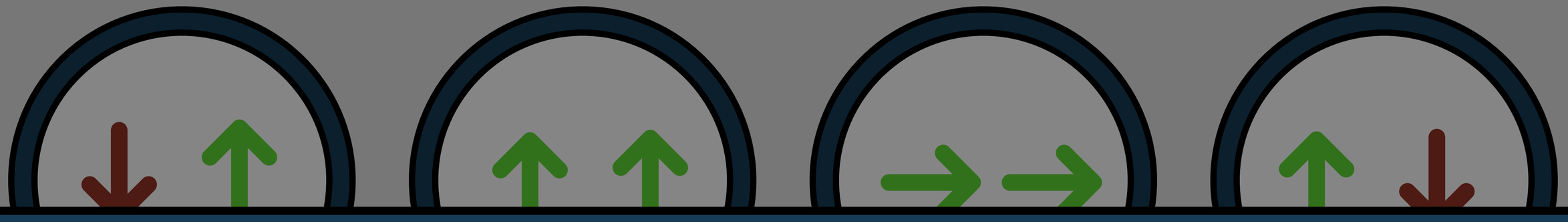
RG: Reusability Evolves Over Time





Decrease/Increase

RG: Reusability Evolves Over Time



Programming Abstraction evolves over time, but **not** in a **statistically significant way**

Decrease/Increase

Steady-Increasing

Stable

Increase/Decrease

Reusability Mechanisms On Defect Proneness and Maintenance Effort

Inheritance

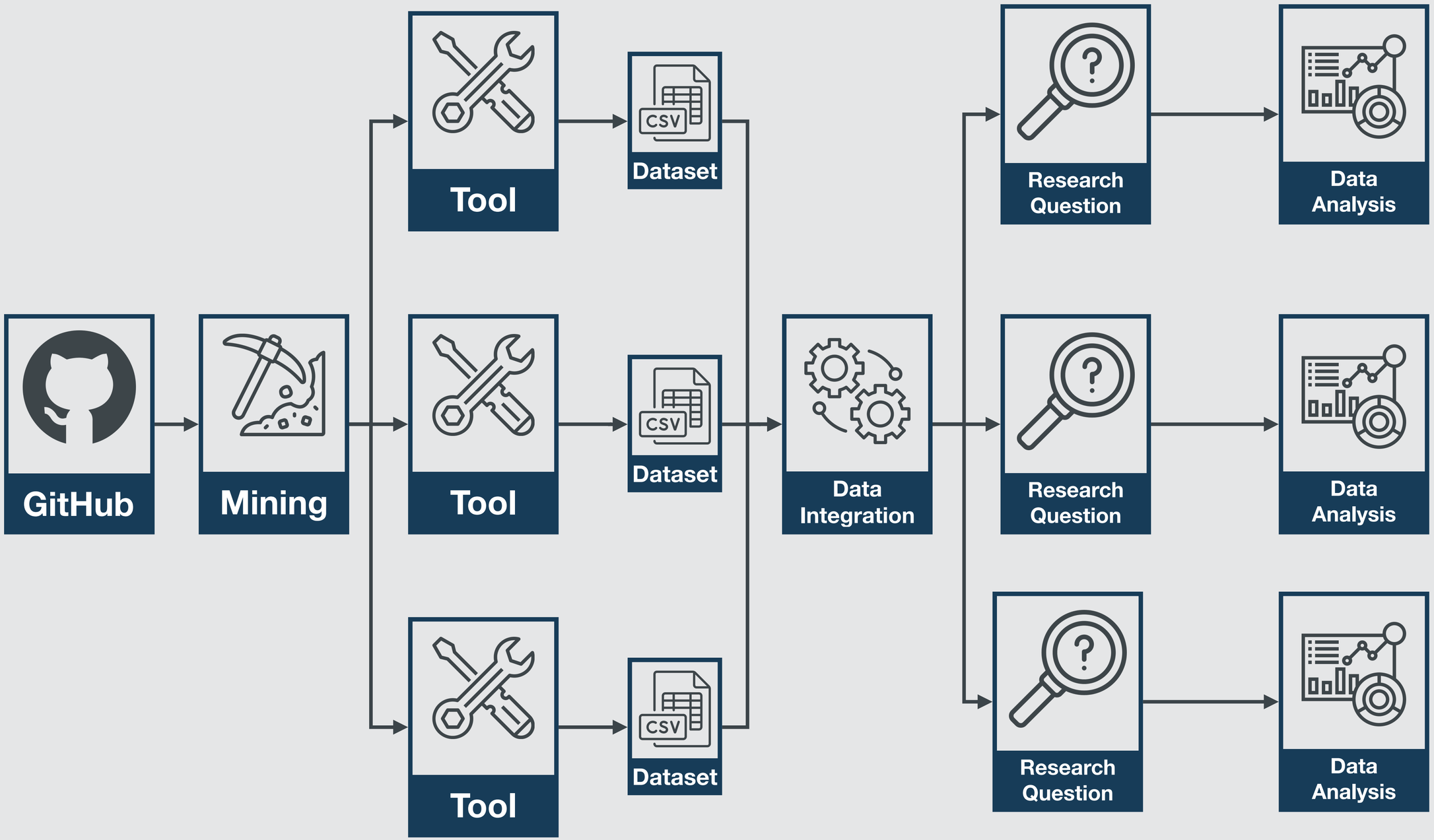
The **class hierarchies** tend to **influence** the likelihood of **defective source code**

Limited connection **between code quality** metrics and **defect-proneness**

Reusability metrics mostly **reduce the effort** required to **fix defects**

Delegation

Research Method Overview



On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Inheritance and Delegation and Code Smells

On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Giammaria Giordano,[1] Antonio Fasulo,[1] Gemma Catolino,[2]
Fabio Palomba,[1] Filomena Ferrucci,[1] Carmine Gravino[1]
[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
giagiordano@unisa.it, a.fasulo@studenti.unisa.it, g.catolino@tilburguniversity.edu
fpalomba@unisa.it, gravino@unisa.it, fferrucci@unisa.it

***Abstract*—Source code reuse is considered one of the holy grails of modern software development. Indeed, it has been widely demonstrated that this activity decreases software development and maintenance costs while increasing its overall trustworthiness. The Object-Oriented (OO) paradigm provides different internal mechanisms to favor code reuse, i.e., specification inheritance, implementation inheritance, and delegation. While previous studies investigated how inheritance relations impact source code quality, there is still a lack of understanding of their evolutionary aspects and, more particular, of how these mechanisms may impact source code quality over time. To bridge this gap of knowledge, this paper proposes an empirical investigation into the evolution of specification inheritance, implementation inheritance, and delegation and their impact on the variability of source code quality attributes. First, we assess how the implementation of those mechanisms varies over 15 releases of three software systems. Second, we devise a statistical approach with the aim of understanding how inheritance and delegation let source code quality—as indicated by the severity of code smells—vary in either positive or negative manner. The key results of the study indicate that inheritance and delegation evolve over time, but not in a statistically significant manner. At the same time, their evolution often leads code smell severity to be reduced, hence possibly contributing to improve code maintainability.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.**

I. Introduction

Software reusability refers to the development practice through which developers make use of existing code when implementing new functionalities [1], [2]. This is widely considered as a best practice, as it leads developers to save time, energy, and maintenance costs, other than relying on source code that has been previously tested [3], [4].

Contemporary Object-Oriented (OO) programming languages, *e.g.,* Java, provide developers with various mechanisms supporting code reusability: examples are design patterns [5], [6], the use of third-party libraries [7], [8], and programming abstractions [9]. These latter, in particular, have caught the attention of researchers since the rise of object-orientation and were found to be a valuable element to increase software quality and reusability [10], [11], [12], [13], [14].

When focusing on Java, there are two well-known abstraction mechanisms such as *inheritance* and *delegation* [15].

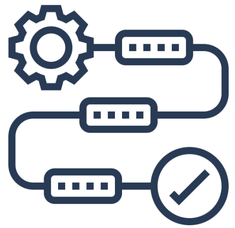
Inheritance is the process by which one class takes the property of another class: the new classes, known as derived or children classes, inherit the attributes and/or the behavior of the pre-existing classes, which are referred to as base, super, or parent classes. Delegation is, instead, the mechanism through which a class uses an object instance of another class by forwarding it messages and letting it performing actions [15].

The importance of inheritance and delegation has been remarked multiple times by the research community. In 1994, Chidamber and Kemerer [16] included in their Object-Oriented metric suite the Depth of the Inheritance Tree (DIT) metric, a measure of the number of classes that inherit from one another. Later on, various metric catalogs proposed variations of DIT as well as other inheritance metrics [17], [18], [19]. In addition, the sub-optimal adoption of inheritance and delegation mechanisms had led to the definition and investigation of reusability-specific code smells [20], [21], [22], [23]: as an example, Fowler [24] defined the *Refused Bequest* and *Middle Man* code smells, which refer to the poor use of inheritance and delegation in Object-Oriented programs that might lead to deteriorate their code quality [22], [25], [26], [27]. These studies have also led to the definition of automated code smell detection and refactoring approaches [28], [29], [30].

Still from an empirical standpoint, a number of studies targeted the role of inheritance and delegation mechanisms for monitoring software quality. In particular, researchers devoted extensive effort on the understanding of the potential impact of those mechanisms on software metrics [31], [32], [33], maintainability effort and costs [34], [35], [36], [37], design patterns [38], [39], change-proneness [40], [41], [42], [43], and source code defectiveness [44], [45], [46], [47].

While the current body of knowledge provides compelling evidence of the value of reusability mechanisms for the analysis of source code quality properties, we can still identify a noticeable research gap: as Mens and Demeyer [48] already reported in the early 2000s, the *long-term* evolution of source code quality metrics might provide a different perspective of the nature of a software project, possibly revealing complementary or even contrasting findings with respect to the studies that investigated code metrics in a fixed point of software evolution. To the best of our knowledge, Nasseri *et al.* [49] were the only researchers studying the evolution of reusability metrics. They specifically focused on the size of the inheritance hierarchies and aimed at assessing whether

Software analysis, evolution, and Reengineering (SANER)



3 Java Systems and 15 releases analyzed

Inheritance and Delegation Most of Time **Statistically** Contribute to the **Decrease** of **Code Smell Severity**

The Yin and Yang of Software Quality: On the Relationship between Design Patterns and Code Smells

The Yin and Yang of Software Quality:
On the Relationship between
Design Patterns and Code Smells

Giammaria Giordano, Giulia Sellitto, Aurelio Sepe, Fabio Palomba, Filomena Ferrucci
Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy
giagiordano@unisa.it, gisellitto@unisa.it, a.sepe21@studenti.unisa.it, fpalomba@unisa.it, fferrucci@unisa.it

***Abstract*—Software reuse is considered the silver bullet of software engineering. It has been largely demonstrated that the proper implementation of design and reuse principles can substantially reduce the effort, time, and costs required to develop software systems. Design patterns are one of the most affirmed techniques for source code reuse. While previous work pointed out their benefits in terms of maintainability and understandability, some seem to raise the opposite concern, suggesting that they can negatively impact code quality from the developers' perspectives. We recognize such discrepancy in the literature, and we aim to fill this gap by investigating whether and how design patterns are related to the emergence of issues compromising code understandability, namely the *Complex Class*, *God Class*, and *Spaghetti Code* smells, which have been also shown to increase the change- and fault-proneness of code. We perform an empirical evaluation on 15 Java projects evolving over 542 releases, and we find that, although design patterns are supposed to improve code quality without prejudice, they can be related to dangerous issues, as we observe the emergence of code smells in the classes participating in their implementation. From our findings, we distil a number of implications for developers and project managers to support them in dealing with design patterns.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance Effort; Empirical Software Engineering.**

I. Introduction

Software reusability is considered the silver bullet of Software Engineering. The term refers to reusing available source code or already tested solutions to solve a similar problem when implementing new features or refactoring the existing ones [6]. The proper application of reusability practices guarantees developers to reduce time, effort, and costs of the maintenance tasks [21], [30]. Most programming languages, especially the ones implementing the Object-Oriented paradigm, provide a wide range of mechanisms to support developers' in applying encouraged best practices of software reuse, *i.e.*, leveraging third-party libraries, implementing program abstractions, and introducing design patterns [12].

The idea of design patterns was proposed in 1995 by the *Gang of Four*, who defined them as reusable solutions to commonly occurring problems that arise during the design and development of software applications [11]. Adopting such reusability mechanisms can provide several advantages from the developers' perspectives, as their flexibility makes them re-appliable by changing the context, the environment, and the programming languages, without changing the philosophy driving a given pattern [37]. The large spread of Object Oriented programming languages boosted developers to reuse instance classes and to create hierarchies that can be easily used as a basis for the introduction of design patterns. Previous studies investigated the use of design patterns in Java [14], [7], [29], mainly because (1) Java offers, by design, mechanisms and data structures that make large use of reusability principles, especially linked to inheritance, and (2) although the fluctuating trends, Java is still one the most adopted programming languages in large companies and open-source communities.[1] While most research emphasize the importance of reusability mechanisms to guarantee high quality of the software, a number of studies seem to go in the opposite direction, highlighting that a sub-optimal implementation of design patterns can, in turn, increase the code complexity and negatively impact the code in terms of maintainability and comprehension [19]. Fowler and Beck identified *code smells* as indicators of the poor quality of code, affecting its cohesion, coupling, and comprehensibility, ultimately making the code difficult to maintain [10].

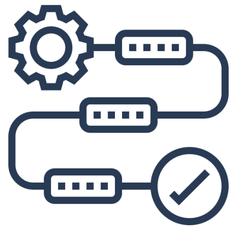
In this paper, we investigate the role that design patterns play in the presence of code smells. We analyze 15 open-source Java projects spanning over 542 releases, by extracting information about the implemented design patterns and the code smells affecting the classes, and assessing (1) the co-occurrences of design patterns and code smells, and (2) whether the presence of design patterns is correlated with the formation of code smells. We find that, although design patterns are intended to improve the quality of the code, as they represent a reuse mechanism, there is no guarantee on them enhancing the goodness of the software; on the contrary, design patterns can in fact determine the appearance of code smells in certain cases. We point out the importance of carefully dealing with design patterns by applying them properly and monitoring their evolution in the software projects. Our main points of contribution can be summarized as follows:

1) An empirical investigation of design patterns and their impact on code smells, that can enhance the state of the art on software reusability and, at the same time, can aid practitioners in monitoring the changes in complexity and comprehension when implementing design patterns;

[1]Source: https://www.tiobe.com/tiobe-index/

Euromicro SEAA 2023

Design Patterns and Code Smells



15 Java Systems and 543 releases analyzed

The **presence** of some **design patterns** can increment the **probability** of the rise of **specific code smells**

Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Systems

Built-in features and Code Smells

Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study

Giammaria Giordano[1], Giusy Annunziata[1], Andrea De Lucia[1] and Fabio Palomba[1]

[1]*University of Salerno (Italy) - SeSa Lab*

Abstract

To deal with continuous change requests and the strict time-to-market, practitioners and big companies constantly update their software systems to meet users' requirements. This practice force developers to release immature products, neglecting best practices to reduce delivery times. As a possible result, *technical debt* can arise, *i.e.*, potential design issues that can negatively impact software maintenance and evolution and, in turn, increase both the time-to-market and costs. *Code smells*—sub-optimal design decisions identifiable by computing software metrics and providing a general overview of code quality —are common symptoms of technical debt. While previous research focused on code smells primarily considering them in the context of Java, the growing popularity of Python, particularly for developing artificial intelligence (AI)-Enabled systems, calls for additional investigations. This preliminary analysis addresses this gap by exploring the diffusion of Python-specific code smells, and the activities performed by developers that induce the introduction of code smells in their systems. To perform our preliminary investigation, we selected 200 AI-Enabled systems available in the Niche dataset; We extracted 10,611 information on the releases using PyDriller, and PySmell to extract information about code smells. The results reveal several insights: 1) Code smells related to object-oriented principles are rarely detected in Python; 2) Complex List Comprehension is the most prevalent and the most long-alive smell; 3) The main activities that can induce code smells are evolutionary. This study fills a critical gap in the literature by providing empirical evidence on the evolution of code smells in Python-based AI-enabled systems.

Keywords

Software Maintenance, Software Evolution, Software Refactoring, Code Smells,

1. Introduction

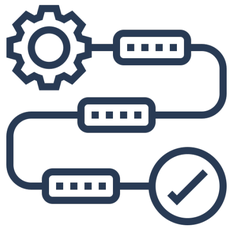
To meet customers' needs and due to the continuous environmental changes, practitioners and big companies update their source code as fast as possible [1]. The continuous *change requests* and the stringent *time-to-market* force developers to release immature products, putting aside best practices to decrease the delivery time [2, 3]. As a possible output of this process could be the introduction of *technical debt* [4]—*i.e.*, potential design issues that can negatively impact the software system during maintenance and evolution. One of the *symptoms* of technical debt is

IWSM/MENSURA 23, September 14–15, 2023, Rome, Italy

giagiordano@unisa.it (G. Giordano); gannunziata@unisa.it (G. Annunziata); adelucia@unisa.it (A. D. Lucia); fpalomba@unisa.it (F. Palomba)

https://giammariagiordano.github.io/giammaria-giordano/ (G. Giordano); https://giusyann.github.io/ (G. Annunziata); https://docenti.unisa.it/003241/home (A. D. Lucia); https://fpalomba.github.io/ (F. Palomba)

0000-0003-2567-440X (G. Giordano); 0009-0002-0742-7261 (G. Annunziata); 0000-0002-4238-1425 (A. D. Lucia); 0000-0001-9337-5116 (F. Palomba)



200 Python AI-enabled Systems and 10,611 releases analyzed

Traditional Code Smells analyzed

The **built-in features** of **Python often** are related to the rise of code smells

International Conference on Software Process and Product Measurement (MENSURA)

At the end of the story…

At the end of the story…

If they had monitored **attributes related to class hierarchy** they would have **decreased** the likelihood of **defects** in the code

The Ariane 5 **reused** the code from the inertial reference platform from the Ariane 4, but the early part of the Ariane 5's flight path differed from the Ariane 4 in having higher horizontal velocity values. This caused an internal value BH (Horizontal Bias) calculated in the alignment function to be unexpectedly high.

At the end of the story…

If they had monitored **attributes related to class hierarchy** they would have **decreased** the likelihood of **defects** in the code

if they had made more **optimal** use of **reuse mechanisms** they would have **decreased smell** and **maintenance effort**

Key Findings

Key Findings

Key Findings

A correct use of **inheritance and delegation,** most time, **statistically significantly increases code quality attributes**

Key Findings

A correct use of **inheritance and delegation,** most time, **statistically significantly increases code quality attributes**

The **CK** metrics show a **limited connection** to **defect proneness**

Key Findings

A correct use of **inheritance and delegation,** most time, **statistically significantly increases code quality attributes**

The **CK** metrics show a **limited connection** to **defect proneness**

Design Patterns are **not always** the **best solution** to **increase code quality**

Future Work

When Code Smells Meet ML: On the Lifecycle of ML-specific Code Smells in ML-enabled systems

When Code Smells Meet ML: On the Lifecycle of ML-specific Code Smells in ML-enabled Systems

Gilberto Recupito
Sesa Lab - University of Salerno
Salerno, Italy
grecupito@unisa.it

Giammaria Giordano
Sesa Lab - University of Salerno
Salerno, Italy
giagiordano@unisa.it

Filomena Ferrucci
Sesa Lab - University of Salerno
Salerno, Italy
fferrucci@unisa.it

Dario Di Nucci
Sesa Lab - University of Salerno
Salerno, Italy
ddinucci@unisa.it

Fabio Palomba
Sesa Lab - University of Salerno
Salerno, Italy
fpalomba@unisa.it

ABSTRACT

Context. The adoption of Machine Learning (ML)–enabled systems is steadily increasing. Nevertheless, there is a shortage of ML-specific quality assurance approaches, possibly because of the limited knowledge of how quality-related concerns emerge and evolve in ML-enabled systems. **Objective.** We aim to investigate the emergence and evolution of specific types of quality-related concerns known as ML-specific code smells, *i.e.*, sub-optimal implementation solutions applied on ML pipelines that may significantly decrease both quality and maintainability of ML-enabled systems. More specifically, we present a plan to study ML-specific code smells by empirically analyzing (i) their prevalence in real ML-enabled systems, (ii) how they are introduced and removed, and (iii) their survivability. **Method.** We will conduct an exploratory study, mining a large dataset of ML-enabled systems and analyzing over 400k commits about 337 projects. We will track and inspect the introduction and evolution of ML smells through CodeSmile, a novel ML smell detector that we will build to enable our investigation and to detect ML-specific code smells.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Technical Debt; ML-Specific Code Smells; Software Quality Assurance; Software Engineering for Artificial Intelligence.

ACM Reference Format:
Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. 2023. When Code Smells Meet ML: On the Lifecycle of ML-specific Code Smells in ML-enabled Systems. In *MSR 2024: 21st International Conference on Mining Software Repositories, April 15–16, 2024, Lisbon, ES.* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

MSR 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Machine Learning (ML) evolved through the emergence of complex software integrating ML modules, defined as ML-enabled systems [13]. Self-driving cars, voice assistance instruments, or conversational agents like ChatGPT[1] are just some examples of the successful integration of ML within software engineering projects.

However, the strict time-to-market and change requests pressure practitioners to roll out immature software to keep pace with competitors, leading to the possible emergence of technical debt [7] *i.e.*, a technical trade-off that can give benefits in a short period, but that can compromise the software health in the long run. *Code smells* is a manifestation of technical debt. They are *symptoms* of poor design and implementation choices that, if left unaddressed, can deteriorate the overall quality of the system [8].

Sculley et al. [19] showed that ML-enabled systems are incredibly prone to technical debt and code smells, raising the need for a quality assurance process for ML components. Cardozo et al. [3] and Van Oort et al. [24] argued that while the issues in those systems are emerging, there is a lack of quality assurance tools and practices that ML developers can use. This lack of quality management assets stimulates the proliferation of code smells in ML-enabled systems [12]. Consequently, given the complex nature of those systems, new types of code smells have emerged. Considering the aspects that ML developers face when dealing with ML pipelines, Zhang et al. [29] defined a new form of code smells, *AI-specific code smells* (ML-CSs). Similarly to traditional code smells, an ML-CS is defined as a *sub-optimal implementation solution for ML pipelines that may significantly decrease the quality of ML-enabled systems.* A key example of those quality issues is using a loop operation instead of exploiting the corresponding Pandas function for data handling, leading to *Unnecessary Iteration* smell [29].

While some work underlines the need to explore AIML-CSs [6, 29], there is still a lack of knowledge on this type of quality issues. Among the various possible causes, we outline a lack of knowledge on how ML-CSs emerge and evolve and what motivations lead developers to introduce and remove them. This poor knowledge significantly threatens the release of ML-CSs detectors aimed at improving the system's quality. Researchers and practitioners cannot define crucial aspects of smell detection and refactoring, such as (i) the conditions where ML-CSs are more prone to be introduced and

[1] https://chat.openai.com/

Mining Software Repository (MSR) -
(Registered-Report)

ML-specific Code Smells

337 Python ML-enabled Systems and
400,000 releases will be analyzed
ML-specific Code Smells

Code Smells related to the **misuse** of
third-party libraries that enable **ML**
components are the **most frequent**

Future Work

Future Work

Automatic just-in-time refactoring tools able to **improve code quality attributes** based on the **specific domain** where software system work

Future Work

Automatic just-in-time refactoring tools able to **improve code quality attributes** based on the **specific domain** where software system work

Analyzing **different aspects** related to **Code Quality** for Smart Systems (e.g., ML-enabled Systems or IoT Systems) such as **Privacy** and **Security Aspects**

Implications

Implications

Implications

Practitioners need to be **informed** on the **benefits** and **drawbacks** of the use of **reusability** mechanisms during software **maintenance** and **evolution** activities

Implications

Practitioners need to be **informed** on the **benefits** and **drawbacks** of the use of **reusability** mechanisms during software **maintenance** and **evolution** activities

The **SE** community needs to conduct further research to **identify more representative metrics** for measuring **code quality**

Research Direction

Research Direction

Research Direction

Software Engineering community needs to study **quality aspects** by considering programming languages **other than Java** (e.g., Python) to have a greater understanding of **how quality attributes vary over time**

Research Direction

Software Engineering community needs to study **quality aspects** by considering programming languages **other than Java** (e.g., Python) to have a greater understanding of **how quality attributes vary over time**

Software Engineering community needs to propose **mechanisms** and **release frameworks** to **facilitate** the **paradigm shift**

Sum Up

Ultimate Goal

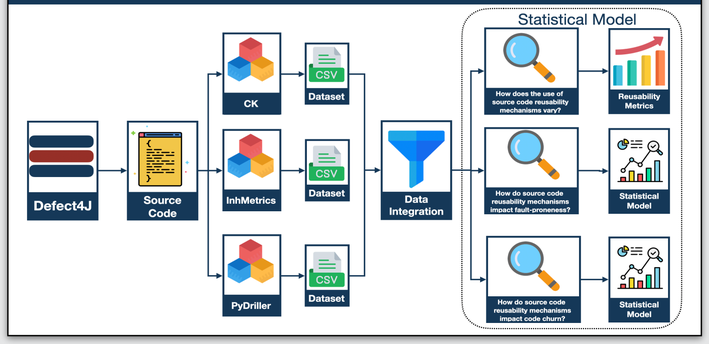

Medium

Bad Code Quality Case Study: Boeing 737 MAX Aircraft

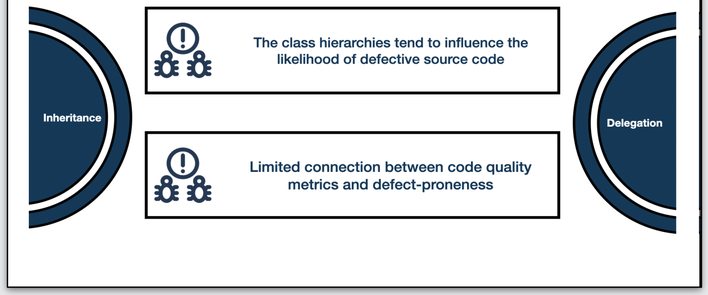
Lion Air Flight 610 was a scheduled domestic flight from Jakarta, Indonesia to Pangkal Pinang. 13 minutes after takeoff the plane crashed into the Java Sea, killing all 189 passengers and crew. It was the first major accident involving the new Boeing 737 MAX aircraft.

The ultimate *goal* of this Ph.D. project is to develop a **continuous quality assurance framework** that **IoT engineers** and software engineers can use to assess **code quality over time**

On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort



Reusability Mechanisms On Defect Proneness and Maintenance Effort



Key Findings

Correct use of **reusability** mechanisms **over time**, most time, **statistically significant increases code quality attributes** (e.g., decreasing the maintenance effort)

Systems that use "*smart technologies*" e.g., artificial intelligence components (AI) can be **impacted by code quality** issues **over time**

Maintenance and Evolutionary activities should be carefully **planned** to **avoid system collapse**

giagiordano@unisa.it

@GiammariaGiord1

https://giammariagiordano.github.io/giammaria-giordano/

sesa lab
SOFTWARE ENGINEERING SALERNO

Thanks!

Backup Slide

Systems that Intensively Use Built-in Features and Code Smells

RG: Systems that Intensively Use Built-in Features and Code Smells

Built-in

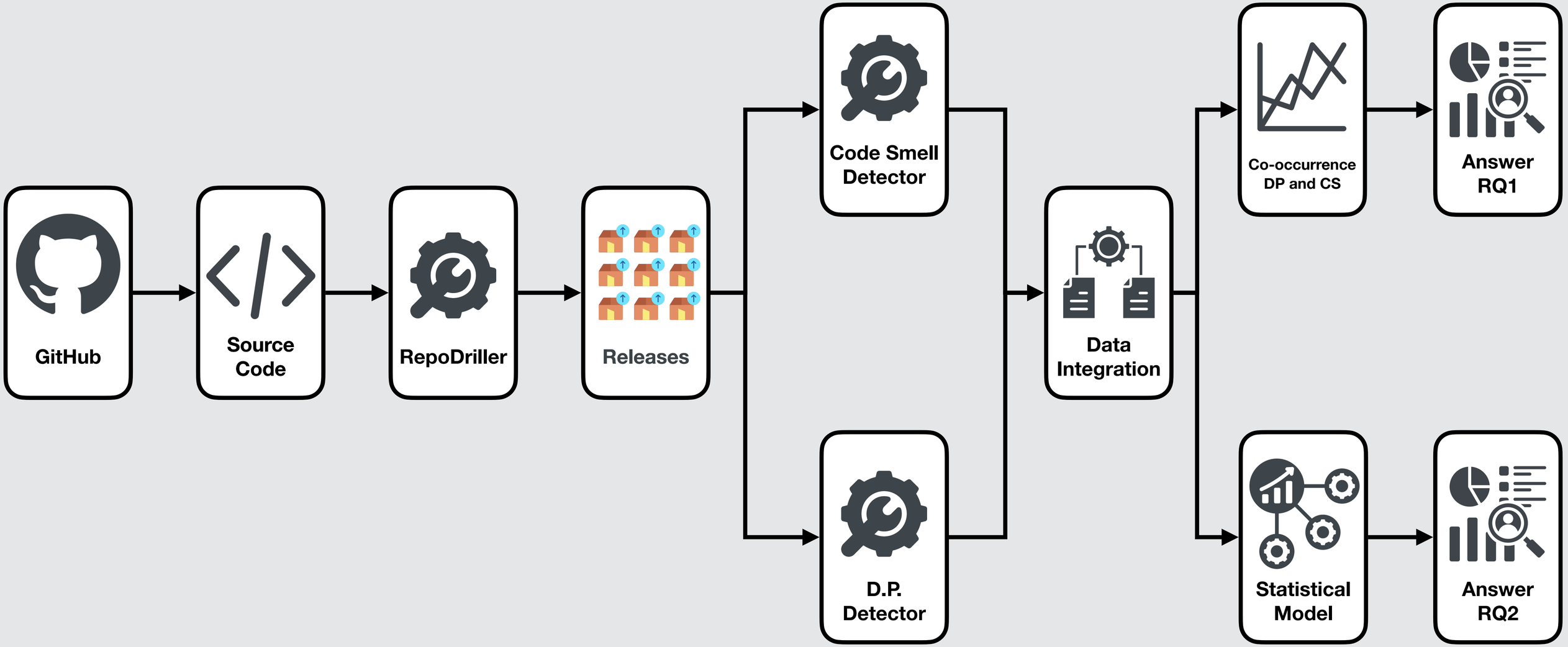
The **Two Code Smells Most Detected** (Complex List Comprehension and Long Ternary Conditional) are due to **sub-optimal use** of **Built-in Features**

70% of the Cases **Code Smells** are **Introduced** Due To **Evolutionary Activities**

Code Smells

Design Patterns On Code Smells

On the Relationship between Design Patterns and Code Smells



Co-occurrence of Design Patterns and Code Smells

In all projects where exists a
co-occurrence between **Design Patterns** and **Code Smells**, the classes implementing **State/Strategy** are affected by **God Class**

In **8** projects the Design Pattern
State/Strategy was also affected by
Spaghetti Code, while in other 4 projects **Complex Class** was identified

60%

of projects are characterized by a **statistical correlation** between **Design Patterns** and **Code Smells**

On Design Patterns and how they affect Code Smells

	God Class	Spaghetti Code	Complex Class
Adapter/Command	+ + +	-	
Bridge	+ + +		
Component	+ + +		
Singleton	+ + +	- -	
Factory Method		+ + +	
Template Method		+ + +	
State/Strategy		+ + +	
Observer			
Proxy			
Decorator			

+ Low Positive Statistical Correlation
++ Medium Positive Statistical Correlation
+++ Strong Positive Statistical Correlation

- Low Negative Statistical Correlation
- - Medium Negative Statistical Correlation
- - - Strong Negative Statistical Correlation

Reusability Mechanisms On Code Smells

RG: Reusability Mechanisms On Code Smells Over Time

Inheritance

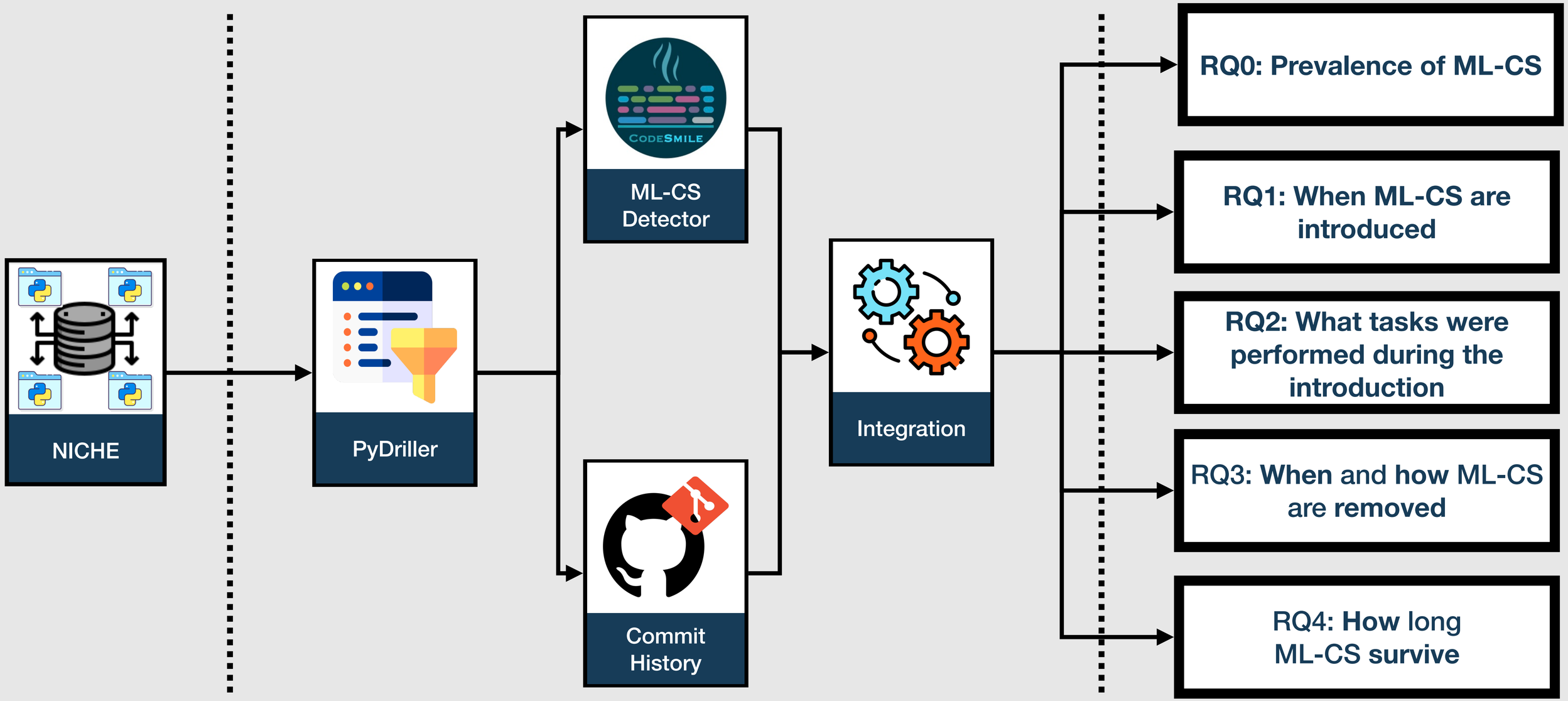
Reusability mechanisms do not statistically influence the defect-proneness of source code

Limited connection between code quality metrics and defect-proneness

Delegation

AI-Specific Code Smells

Research Process

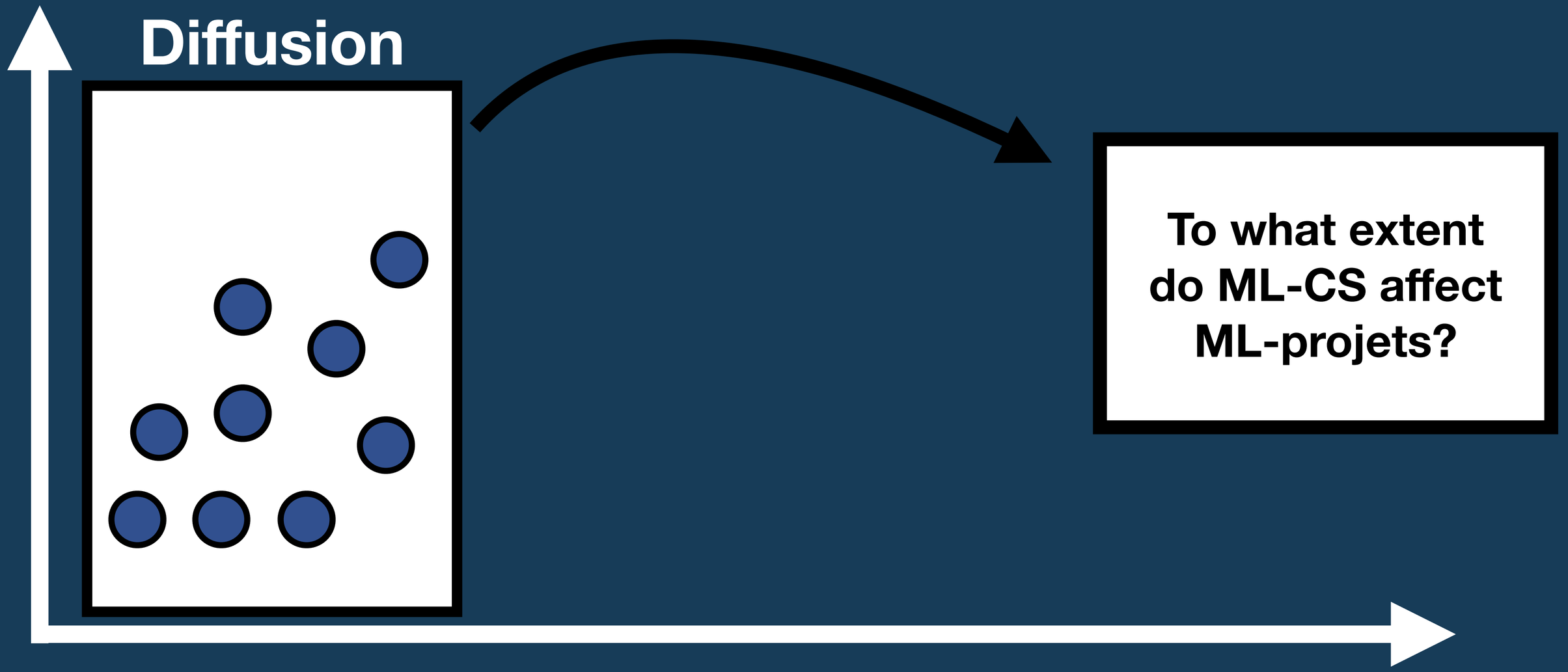


Code Smell Detector

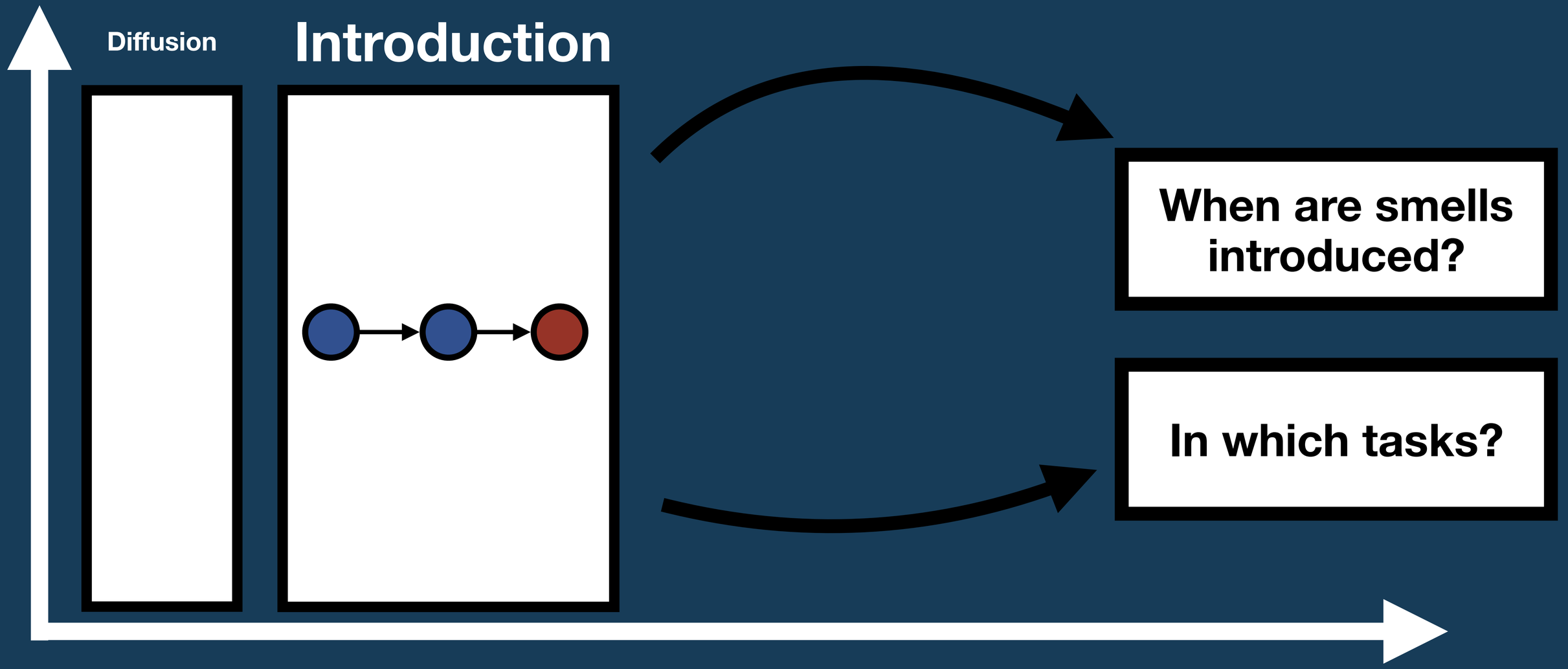
14

ML-specific code smells will be detected

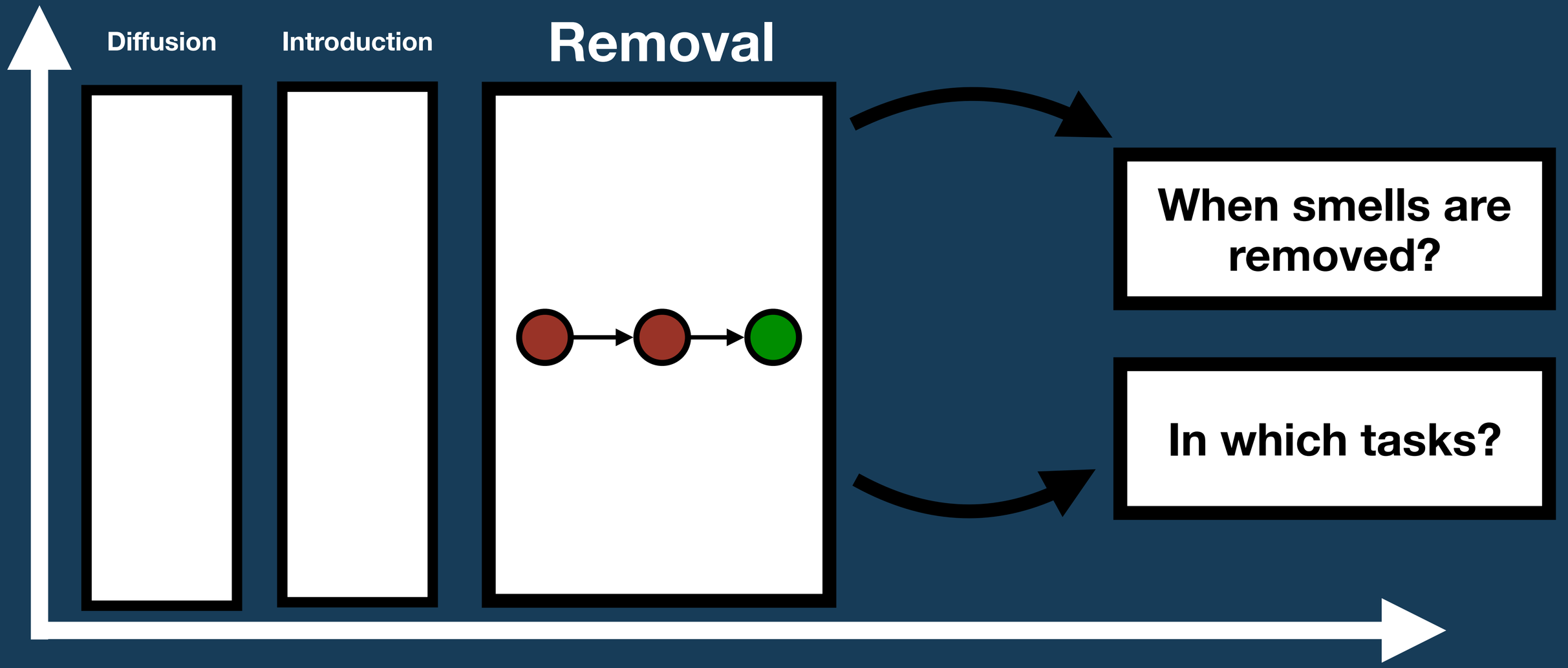
What will we investigate?



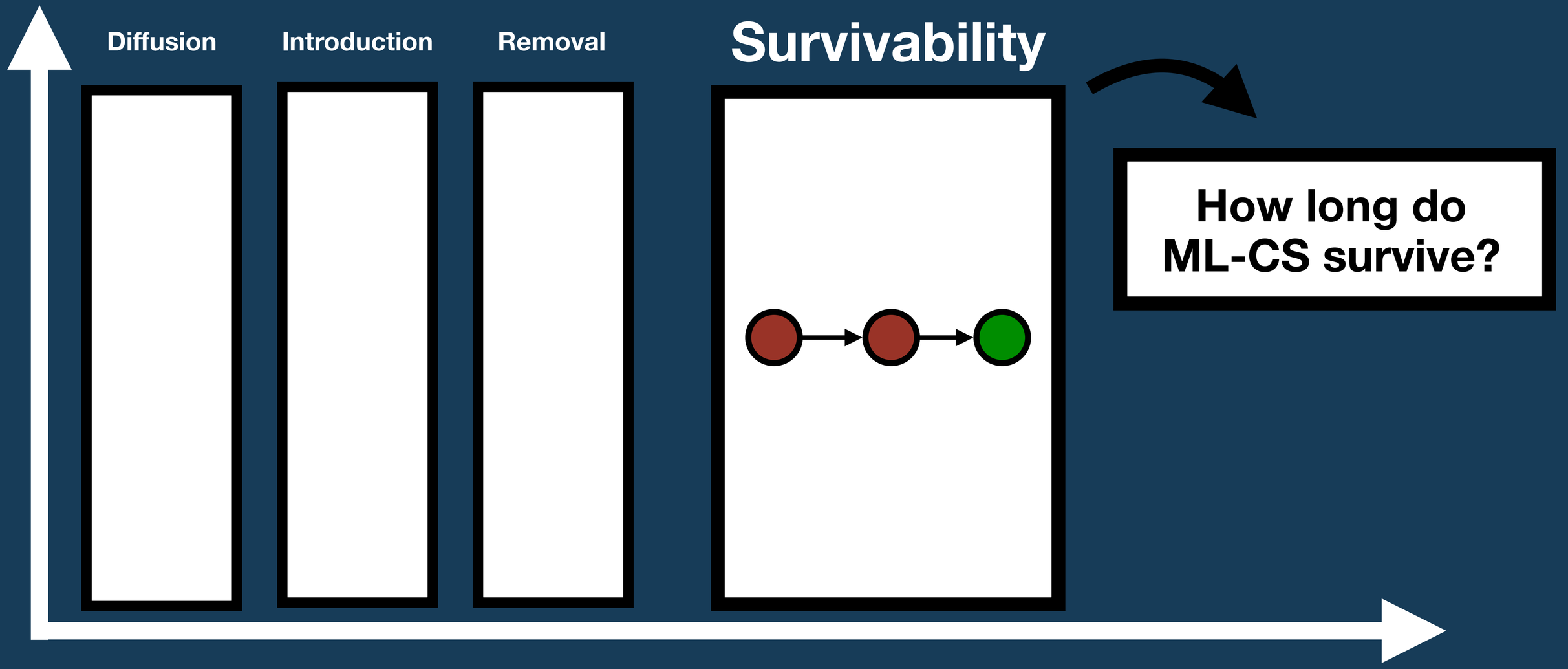
What will we investigate?



What will we investigate?

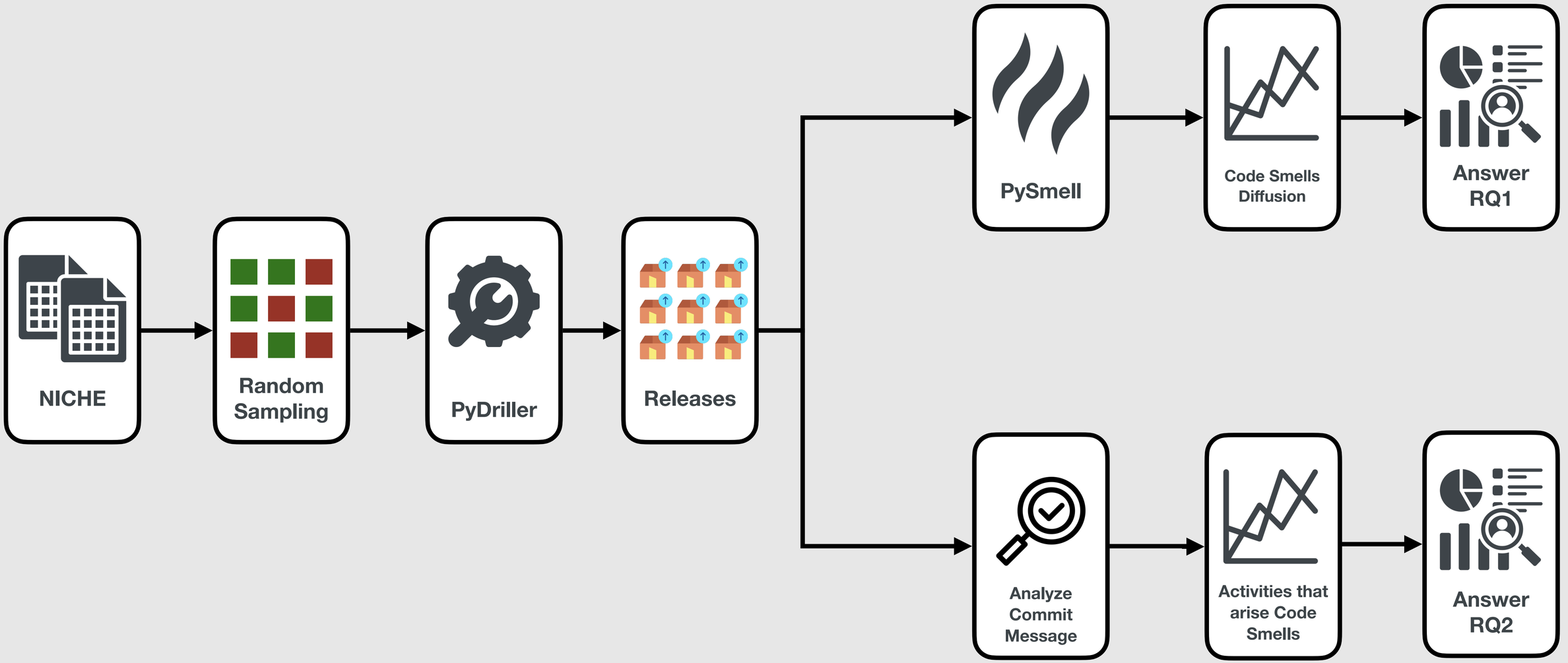


What will we investigate?



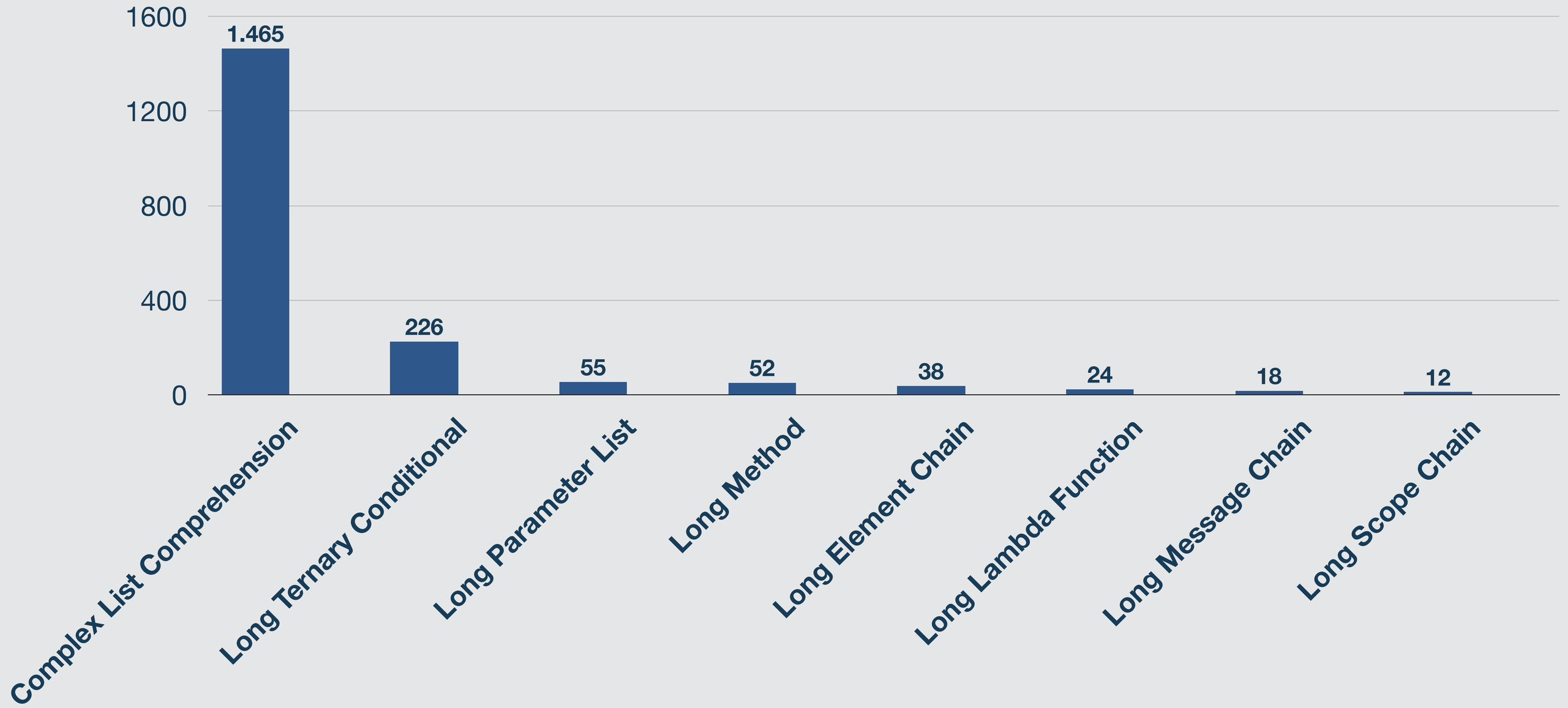
Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Systems

Research Process



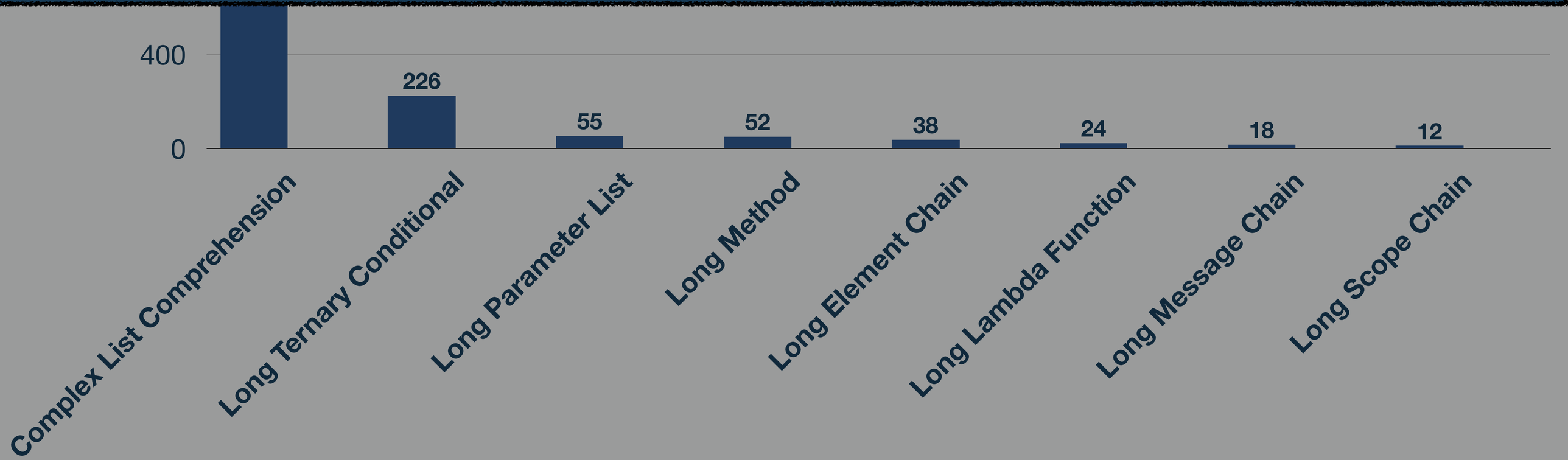
On the **frequency** of **Code Smells** in **AI-Enabled Systems**

On the frequency of Code Smells



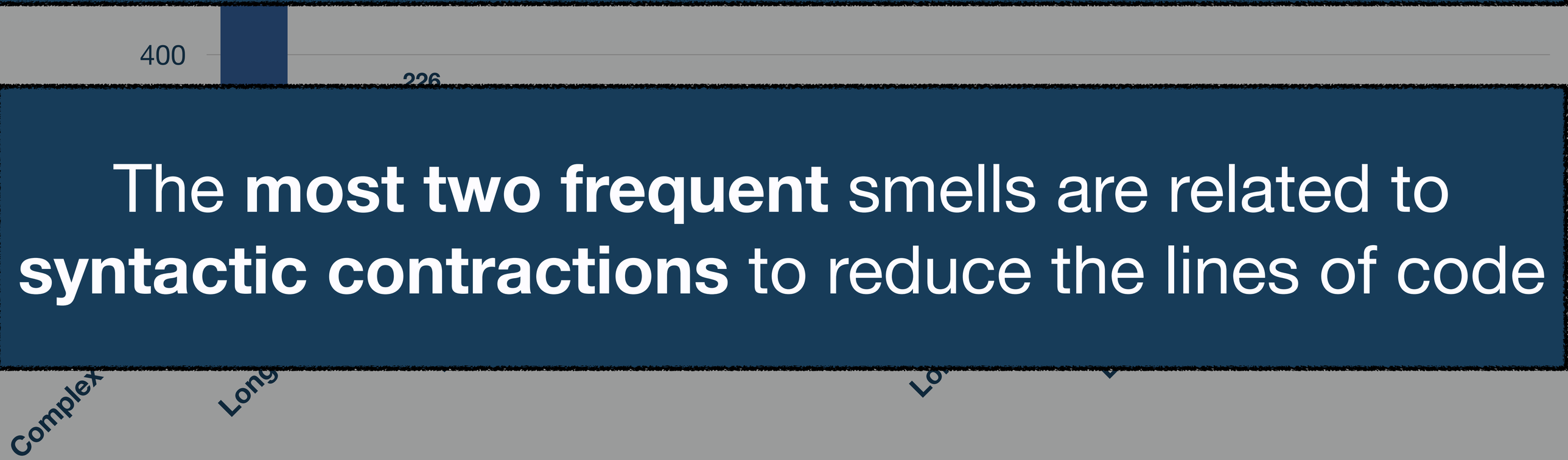
On the frequency of Code Smells

Code Smells related to **Object-Oriented programming languages** (e.g., complex class) are **never detected**



On the frequency of Code Smells

Code Smells related to **Object-Oriented programming languages** (e.g., complex class) are **never detected**



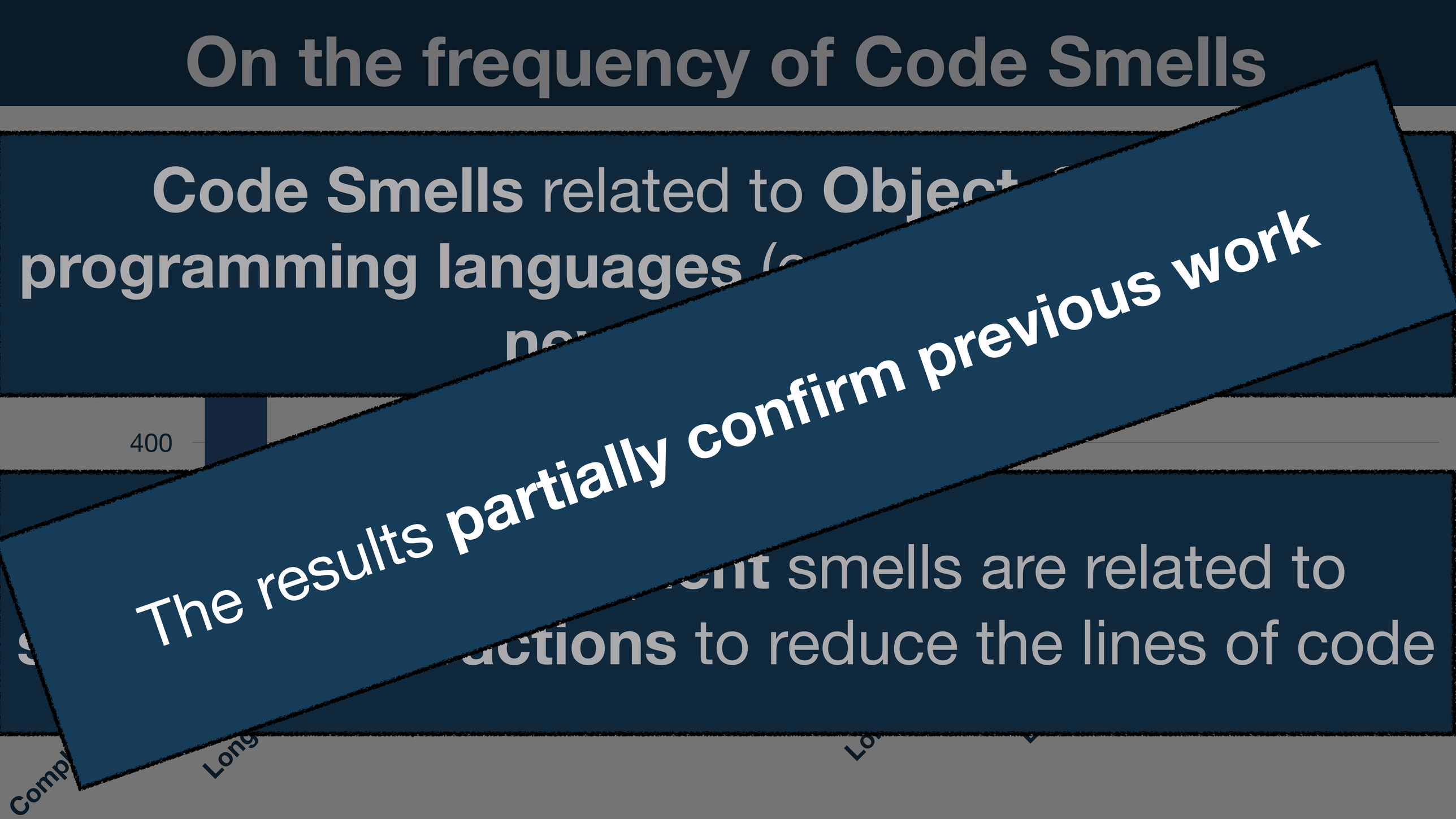
The **most two frequent** smells are related to **syntactic contractions** to reduce the lines of code

On the frequency of Code Smells

Code Smells related to **Object**
programming languages (e
ne

The results **partially confirm previous work**

ent smells are related to
actions to reduce the lines of code



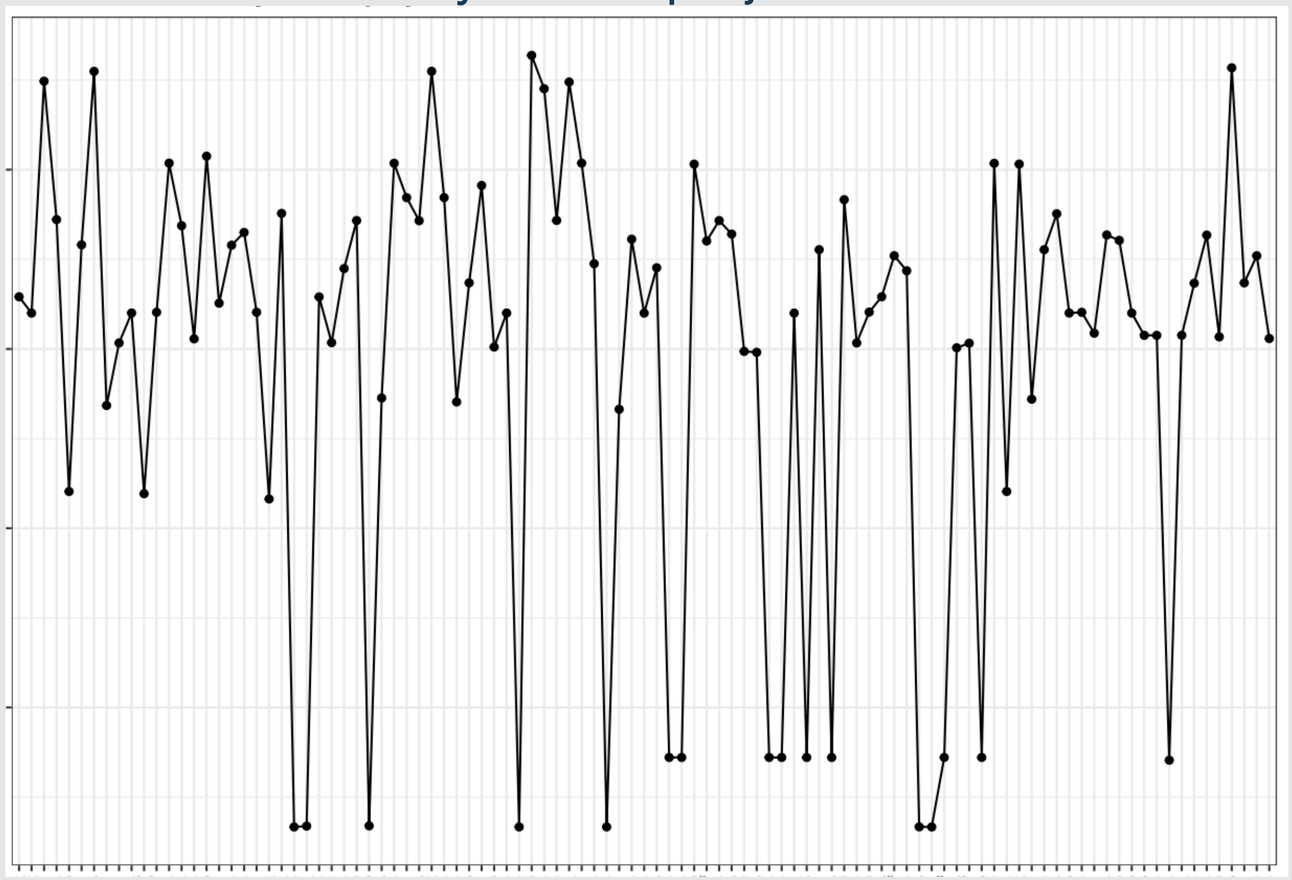
80%

Of the projects were affected at least once by a *Complex List Comprehension*

On the **density** of **Code Smells** in **AI-Enabled Systems**

On the density of Code Smells

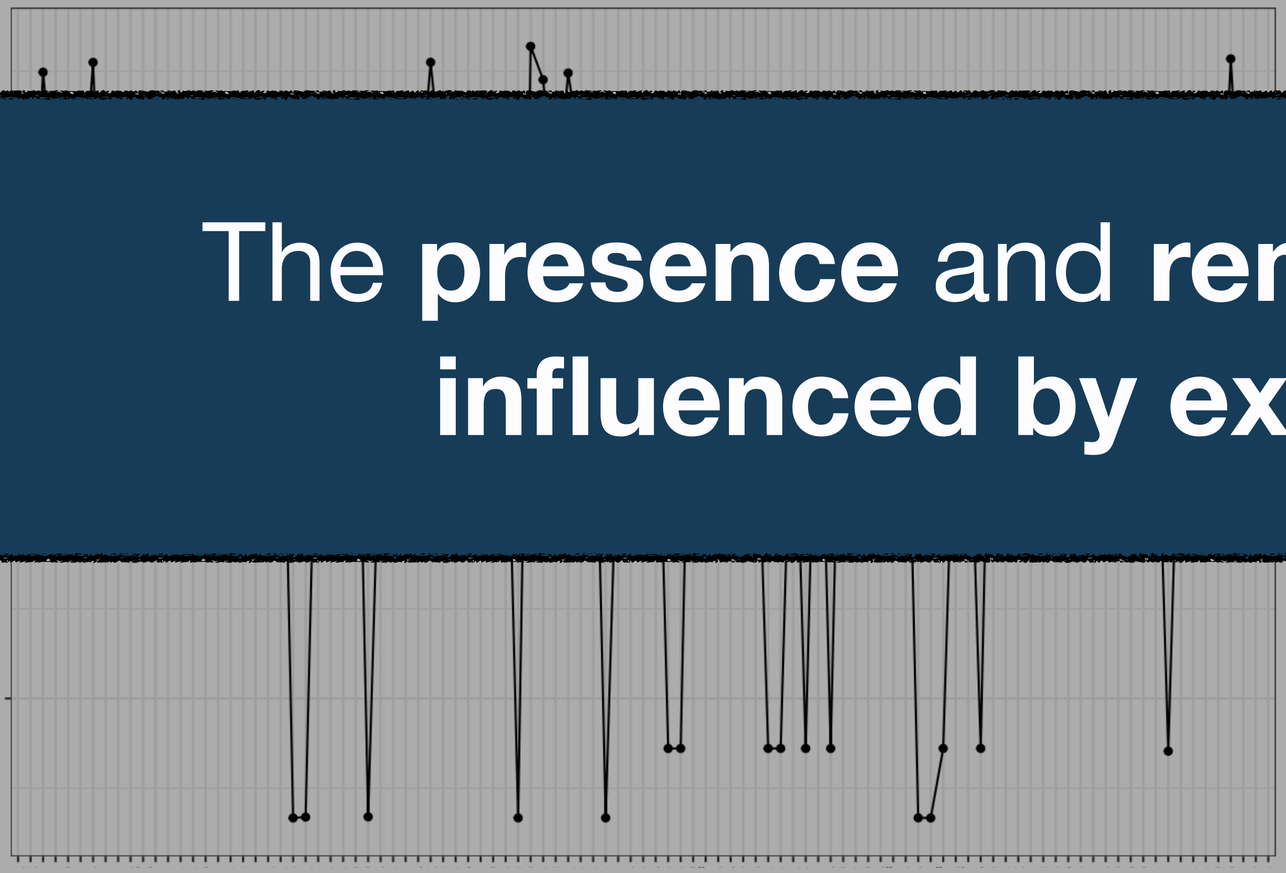
Code Smell density for the project MindMeld



We observed that the **density** of **Code Smells does not follow** a **specific trend** of **increase/ decrease over time**

On the density of Code Smells

Code Smell density for the project MindMeld



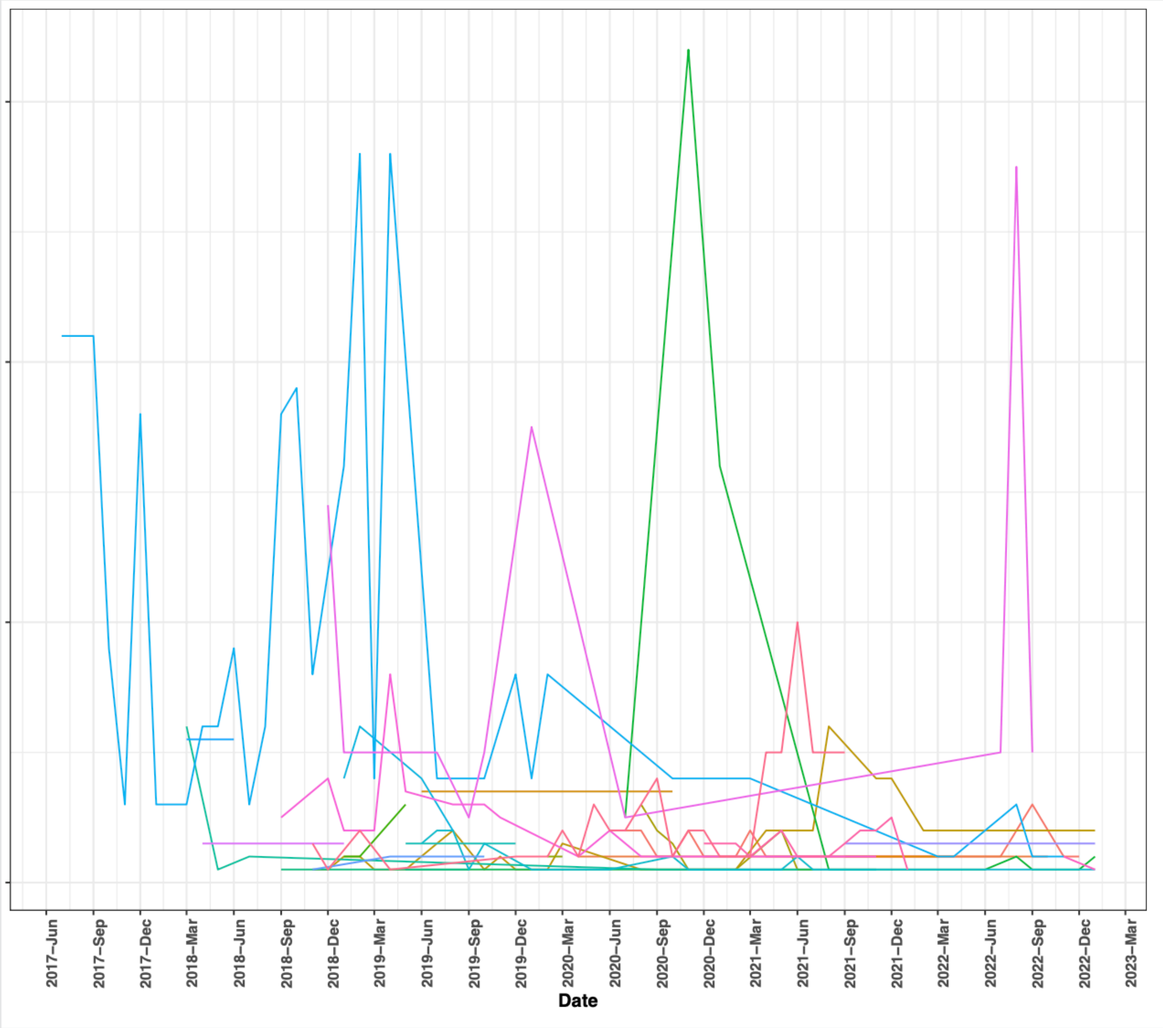
We observed that the **trend** of **increase/ decrease over time**

The **presence** and **removal** appear to be **influenced by external factors**

On the **survival** of **Code Smells** in **AI-Enabled Systems**

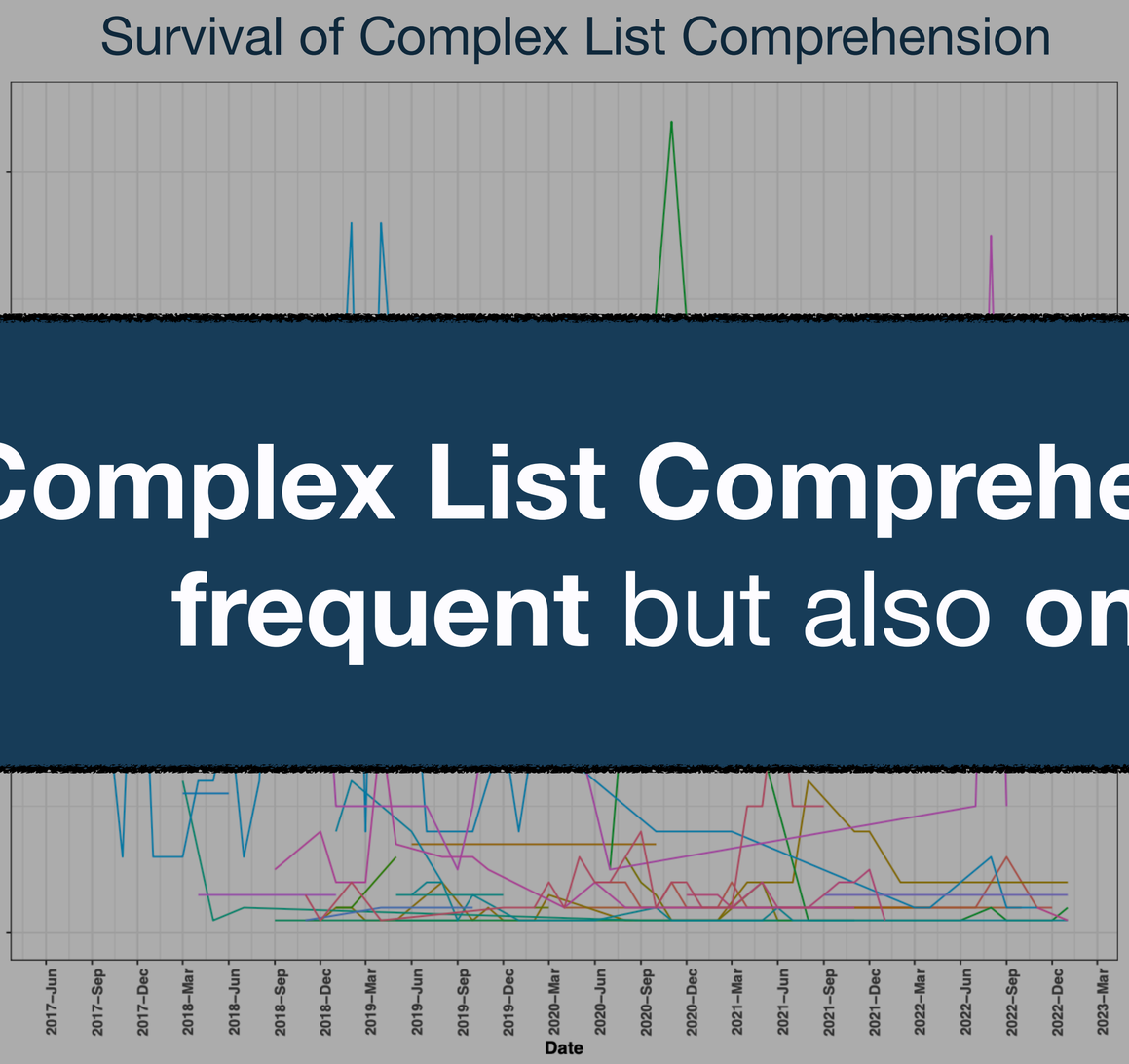
On the survival of Code Smells

Survival of Complex List Comprehension



In some cases, we found that Code Smells survived for a time period of 6 years!

On the survival of Code Smells



Complex List Comprehension is **not only** the **most frequent** but also **one of the longest-lived**

On the **activities** that led developers to introduce Code Smells in AI-Enabled Systems

70%

of Code Smells has been introduced due to Evolutionary Activities

30%

In **most cases** the **introduction** of **code smells** is due to **merge operations**

of Code Smells has been introduced due to Evolutionary Activities

NICHE Dataset

ML Projects

NICHE: A Curated Dataset of Engineered Machine Learning Projects in Python

Ratnadira Widyasari, Zhou Yang, Ferdian Thung, Sheng Qin Sim, Fiona Wee, Camellia Lok, Jack Phan, Haodi Qi, Constance Tan, Qijin Tay, and David Lo
School of Computing and Information System, Singapore Management University
{ratnadiraw.2020,zyang,ferdianthung,sqsim.2018,fiona.wee.2018,camellialok.2017,jack.phan.2018}@smu.edu.sg
{haodi.qi.2017,hytan.2018,qijin.tay.2018,davidlo}@smu.edu.sg

***Abstract*—Machine learning (ML) has gained much attention and been incorporated into our daily lives. While there are numerous publicly available ML projects on open source platforms such as GitHub, there have been limited attempts in filtering those projects to curate ML projects of high quality. The limited availability of such high-quality dataset poses an obstacle in understanding ML projects. To help clear this obstacle, we present *NICHE*, a manually labelled dataset consisting of 572 ML projects. Based on evidences of good software engineering practices, we label 441 of these projects as engineered and 131 as non-engineered. This dataset can help researchers understand the practices that are followed in high-quality ML projects. It can also be used as a benchmark for classifiers designed to identify engineered ML projects.**

***Index Terms*—Engineered Software Project, Machine Learning, Python, Open Source Projects**

I. Introduction

There are many valuable pieces of information stored in a version control system of a project; they include: source code, documentation, issue reports, test cases, list of contributors, etc. Researchers mine these software repositories to get useful insights related to how bugs are fixed [1], how developers collaborate [2] and so on. With the abundance of the open source repositories in GitHub, researchers can mine for insights and validate hypotheses on a large corpus of data. However, Kalliamvakou et al. showed that most repositories in Github are of low-quality [3], [4], which can lead to wrong and biased conclusions. To avoid skewed findings, researchers usually take some measures to filter out low-quality projects, e.g., by choosing projects with a high number of stars (which is considered to reflect the projects' popularity). Unfortunately, popularity may not be correlated with project quality [5]. Therefore, Munaiah et al. propose an approach to find high-quality software projects, more specifically; by identifying engineered software projects [6]. Such projects are essential for mining software repository (MSR) research, as they allow for high-quality findings to be uncovered (from high-quality data).

Machine learning (ML) projects are becoming increasingly popular and play essential roles in various domain, e.g., code processing [7], [8], self-driving cars, speech recognition [9], etc. Despite widespread usage and popularity, only a few research works try to examine AI and ML projects to identify unique properties, development patterns, and trends. Gonzalez et al. [10] find that the AI & ML community has unique characteristics that should be considered in future software engineering and MSR research. For example, more support is needed for Python as the main programming language, and there are significant differences between internal and external contributors in AI & ML projects. We coin a term for such research: Mining Machine Learning Repository (MLR). Similar to conventional MSR research, MLR also requires high-quality projects. In GitHub, there are many tutorials, resource pages, courseworks and toy projects that are related to ML; some of which are very popular but unsuitable for MLR research. To facilitate MLR research, we present a curated dataset of E**N**g**I**neered Ma**CH**ine L**E**arning Projects in Python (*NICHE*). We first automatically identify projects in GitHub that: (1) use one of the popular ML libraries, and (2) satisfy some basic quantitative quality metrics. This process returns 572 ML projects from GitHub. Next, we manually analyze the 572 ML projects and label them as engineered or not engineered. This dataset can be used as the raw material for MLR research, or as the benchmark for evaluating classifiers designed to identify engineered ML projects.

We label the dataset manually to ensure high quality and accurate labels. Our criteria for assessing an ML project are rooted in Munaiah et al. work [6]. We check 8 distinct dimensions of a project (architecture, community, CI, documentation, history, issues, license and unit testing) to evaluate whether the project is engineered or not. Out of the 572 projects we collected, 441 projects are labelled as engineered ML projects, and 131 projects are labelled as non-engineered ML projects. There are several related datasets in the literature. Datasets from [6] and [11] have labels indicating whether a project is engineered or not, but they do not contain ML projects. Gonzalez et al. [10] collected a dataset of ML & AI projects, but these projects are not comprehensively assessed based on their adoption of good software engineering practices. They only eliminate tutorials, homework assignments and so on. We make our dataset publicly available[1].

The rest of this paper is organized as follow. Section 2 describes the methodology used to collect and filter the dataset, as well as how the dataset is stored. Section 3 gives an overview of the dataset. In Section 4, we propose some

[1]https://doi.org/10.6084/m9.figshare.21967265

572 ML projects

"**engineered**" and "**not engineered**" according to **8 dimensions** including CI

- Architecture
- Community
- Continuous Integration
- Documentation
- History
- Issues
- License
- Unit Testing

Activities

International Journal Papers

J01 - **Giordano G.**, Ferrucci, F., and Palomba, F. (2022). On the Use of Artificial Intelligence to Deal with Privacy in IoT Systems: A Systematic Literature Review. Journal of Systems and Software (JSS), Vol. 193, 111475

J02- Amato, F., Cicalese, M., Contrasto,L., Cubicciotti, G., D'Ambola, G., LaMarca, A., Pagano, G., Tomeo, F., Robertazzi, G. A., Vassallo, G., Acampora, G., Vitiello, A., Catolino, G., **Giordano, G.**, Lambiase, S., Pontillo, V., Sellitto, G., Ferrucci, F., and Palomba, F. (2023). QuantuMoonLight: A low-code platform to experiment with quantum machine learning. SoftwareX, 22, 101399.

J03- **Giordano, G**., Festa, G., Catolino, G., Palomba, F., Ferrucci ,F., and Gravino,C., On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort. Empirical Software Engineering (EMSE) 29, (2024)

International Conference Papers

C01 - **Giordano, G.,** Fasulo, A., Catolino, G., Palomba, F., Ferrucci, F., and Gravino, C. On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality. IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)

C02 - **Giordano, G.,** Palomba, F., and Ferrucci, F. A Preliminary Conceptualization and Analysis on Automated Static Analysis Tools for Vulnerability Detection in Android App. 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)

C03- **Giordano, G.,** Pontillo, V., Annunziata,G., Cimino, A., Ferrucci, F., and Palomba, F. How May Deep Learning Testing Inform Model Generalizability? The Case of Image Classification. In Proceedings of the 15th Seminar on Advanced Techniques & Tools for Software Evolution (SATToSE)

C04 - **Giordano, G.,** Sellitto, G., Sepe, A., Palomba, F., and Ferrucci, F. The Yin and Yang of Software Quality: On the Relationship between Design Patterns and Code Smells. 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)

C05 - **Giordano, G.,** Annunziata, De Lucia, A., and Palomba, F. Understanding Developer Practices and Code Smells Diffusion in AI-enabled software: A Preliminary Study. International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement 2023

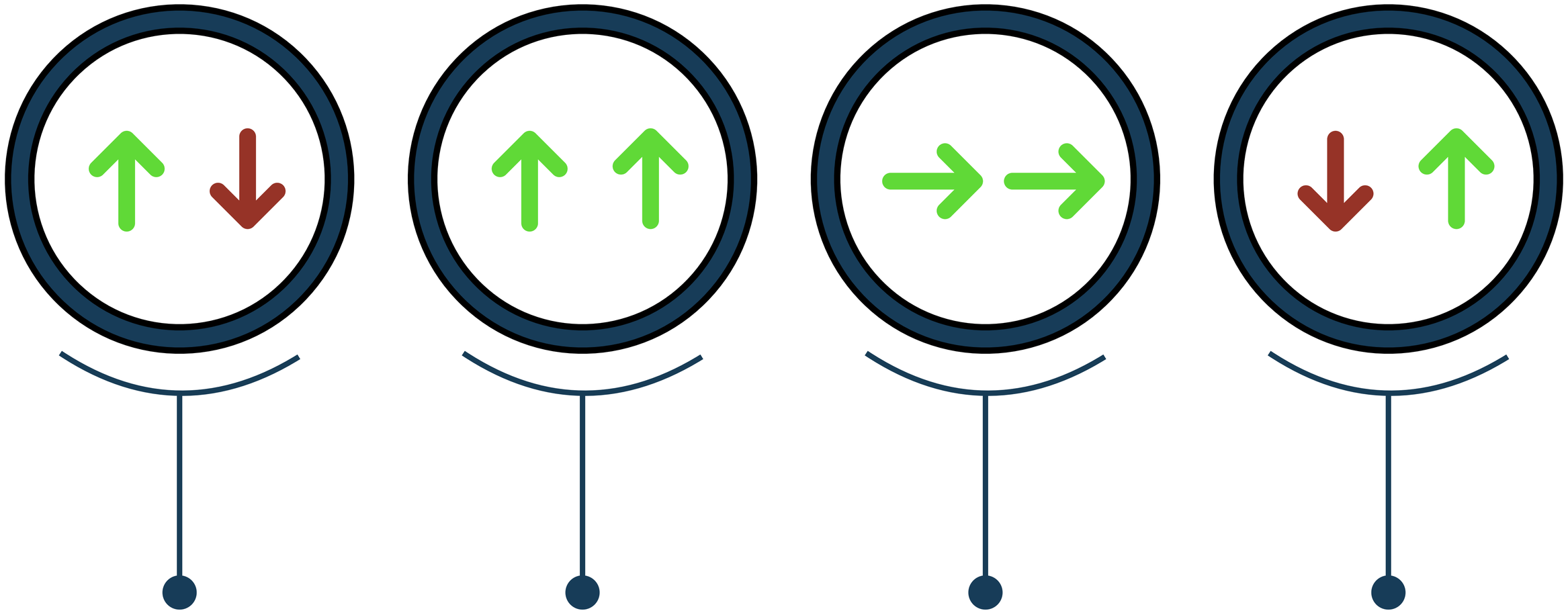
I reviewed over 50 papers among international conferences and journals (e.g., Journal of Systems and Software (JSS), Empirical Software Engineering (EMSE), and Transactions on Software Engineering and Methodology (TOSEM))

I have co-advised 17 B.Sc. and 1 M. Sc. students in Computer Science at the University of Salerno

Results

4 Reusability Mechanisms Patterns

RG: Reusability Evolves Over Time



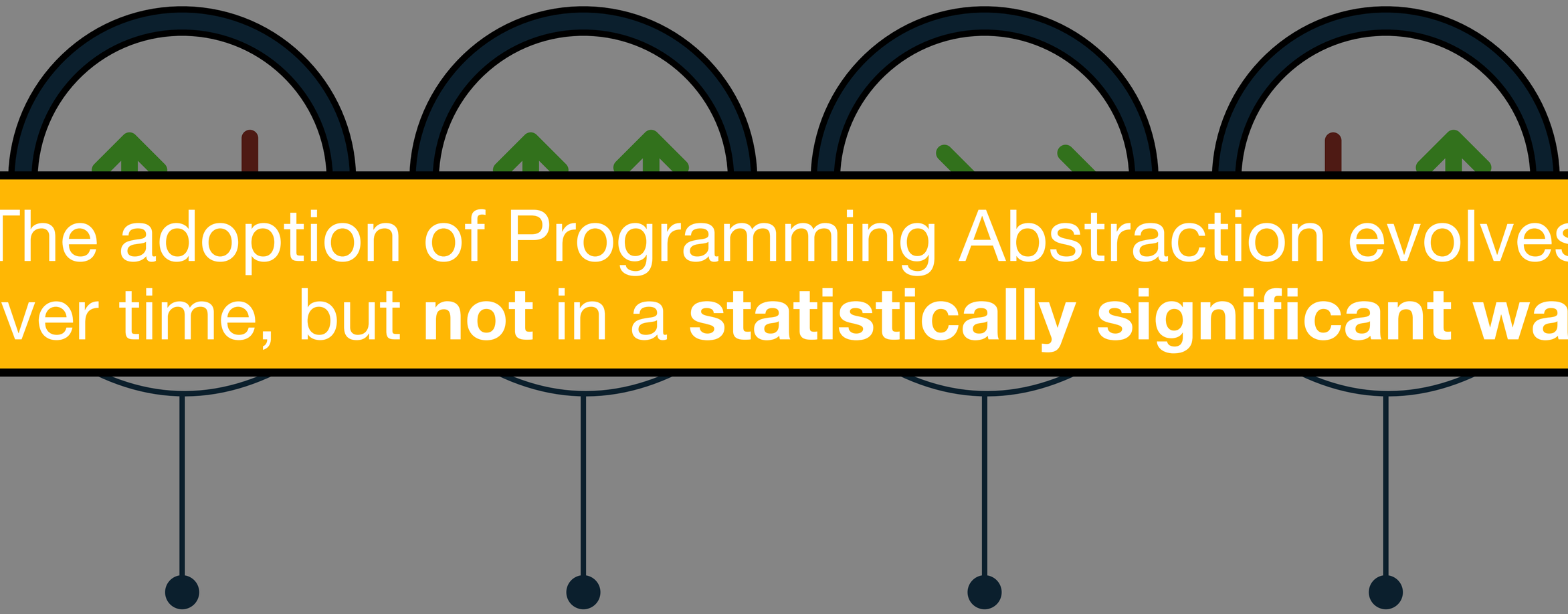
Increase/Decrease

Steady-Increasing

Stable

Decrease/Increase

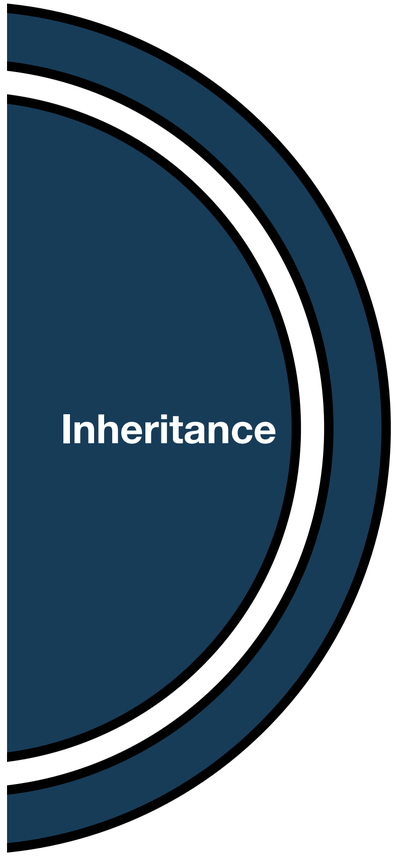
RG: Reusability Evolves Over Time



The adoption of Programming Abstraction evolves over time, but **not** in a **statistically significant way**

Increase/Decrease **Steady-Increasing** **Stable** **Decrease/Increase**

RG: Reusability Mechanisms On Code Smells Over Time

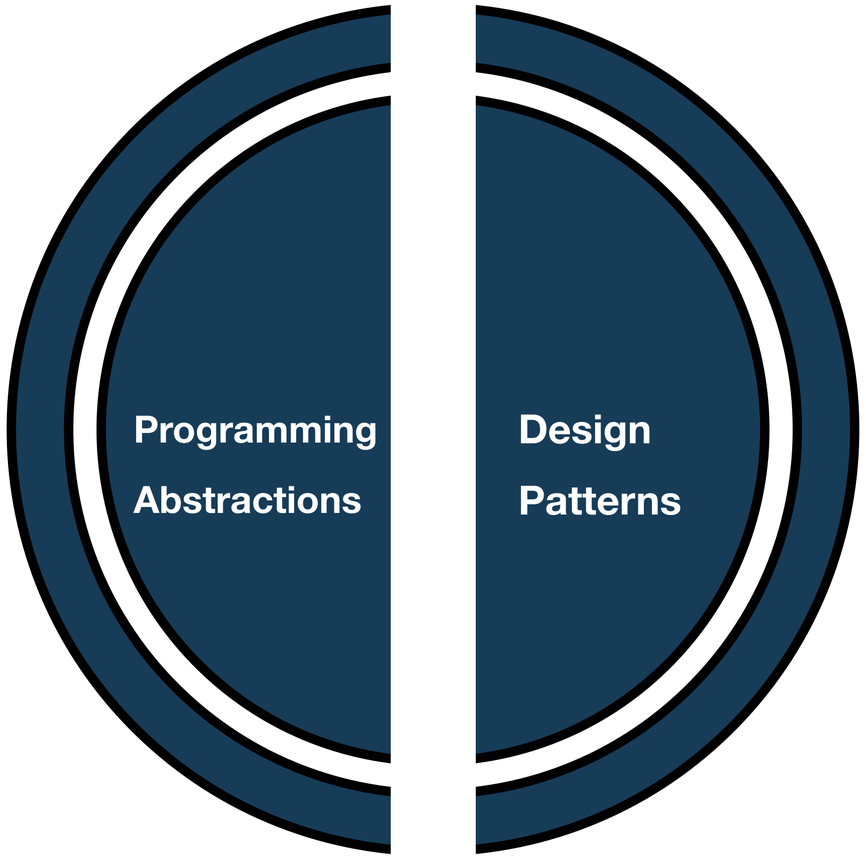


Reusability mechanisms do not statistically influence the defect-proneness of source code

Limited connection between code quality metrics and defect-proneness

Reusability Mechanisms On Code Smells

RG: Reusability Mechanisms On Code Smells Over Time



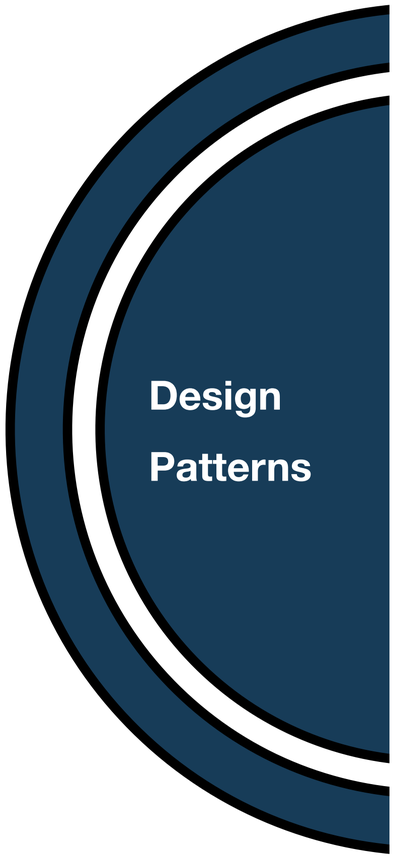
RG: Reusability Mechanisms On Code Smells Over Time

Programming Abstractions

Inheritance and Delegation Most of Time **Statistically** Contribute to the **Decrease** of **Code Smell Severity**

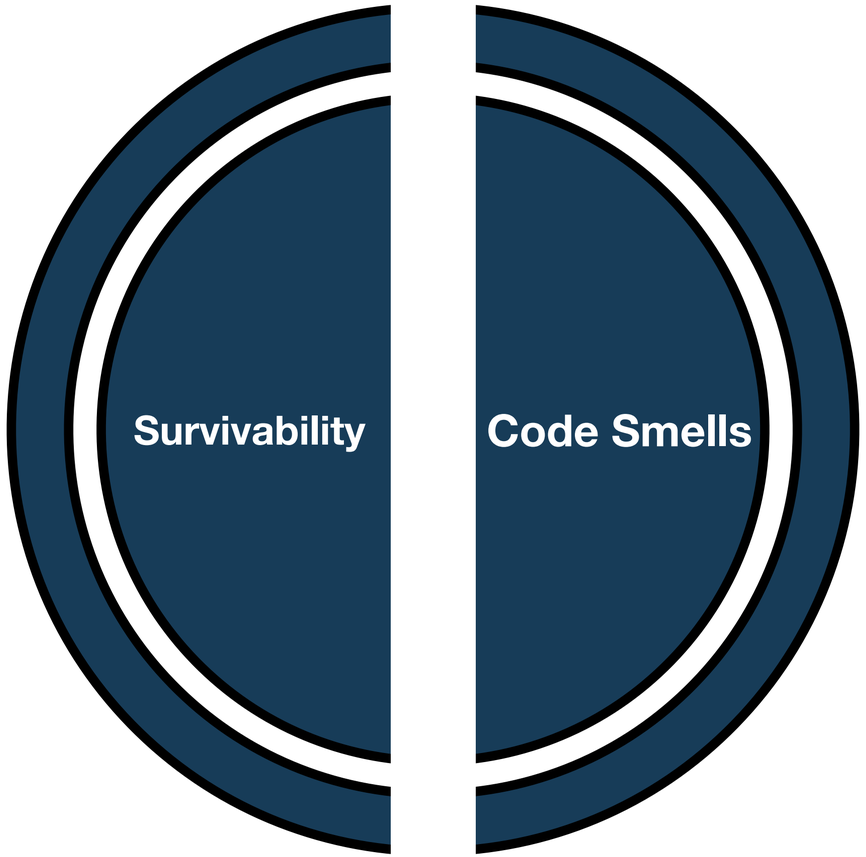
RG: Reusability Mechanisms On Code Smells Over Time

A Considerable Number of Design Patterns are Statistically Correlated with the Emergence of Code Smells



Survivability Code Smells

RG: Survivability Of Code Smells Over Time



RG: Survivability Of Code Smells Over Time

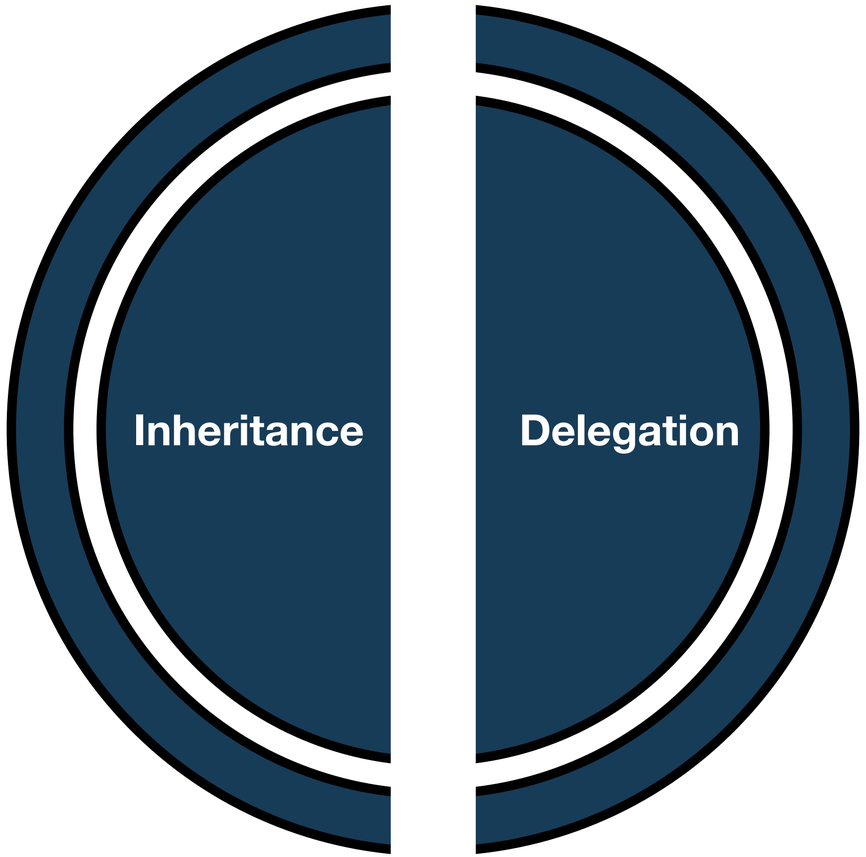
Survivability

Our Results Indicate that Code Smells Can Survive in Software Systems for Many Years

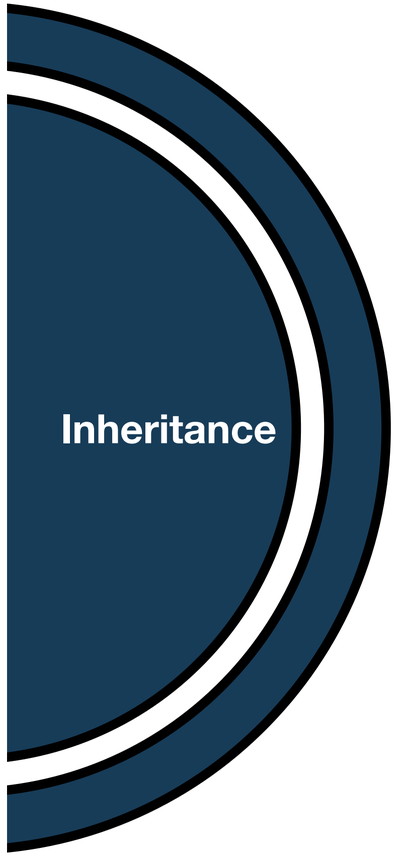
Code Smells

Reusability Mechanisms On Defect Proneness

RG: Reusability Mechanisms On Code Smells Over Time



RG: Reusability Mechanisms On Code Smells Over Time



Reusability mechanisms do not statistically influence the defect-proneness of source code

Limited connection between code quality metrics and defect-proneness

RG: Reusability Mechanisms On Code Smells Over Time

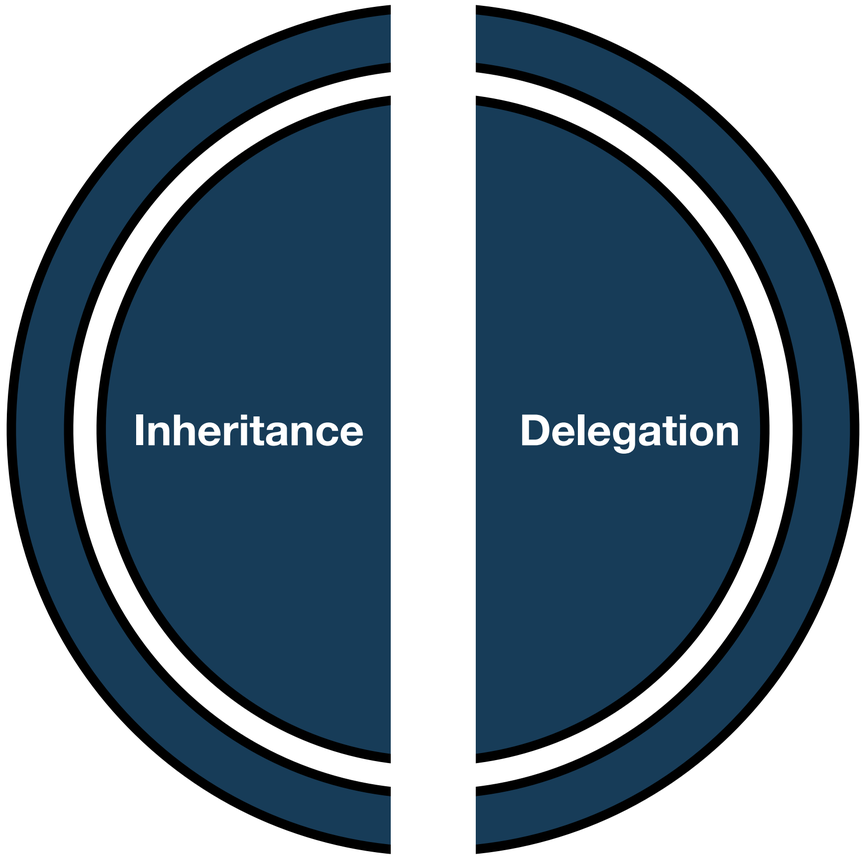
Reusability mechanisms do not statistically

Previous research that considers the relationship between quality metrics and defects should be reconsidered

Limited connection between code quality metrics and defect-proneness

Reusability Mechanisms On Maintenance Effort

RG: Reusability Mechanisms On Code Smells Over Time



RG: Reusability Mechanisms On Code Smells Over Time

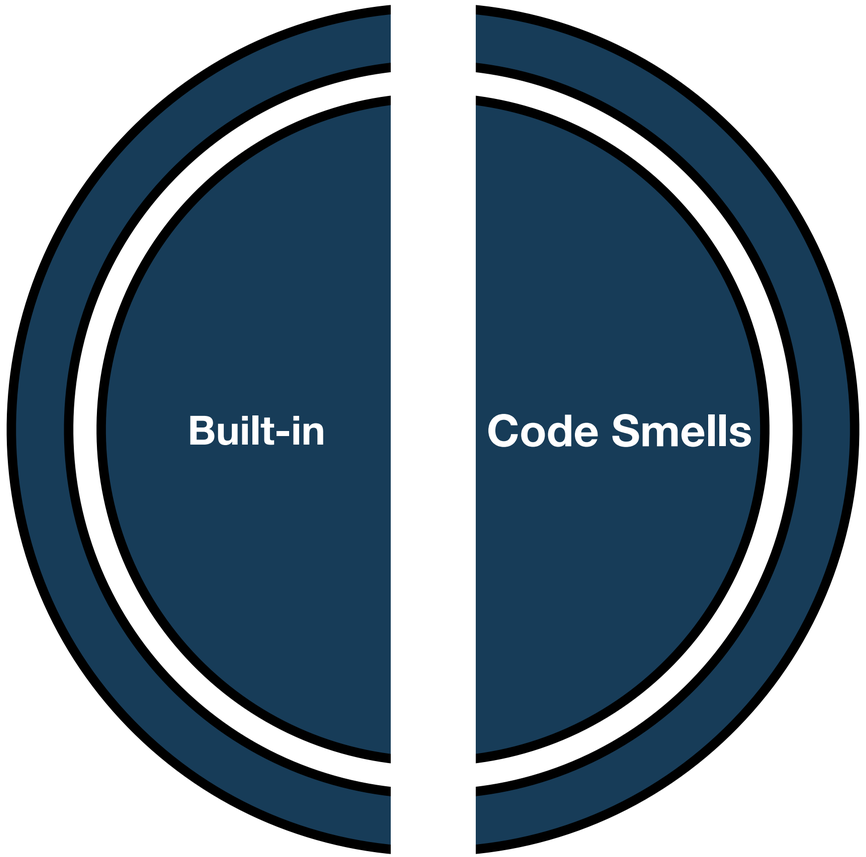
Inheritance

Reusability mechanisms statistically contribute to reduce the effort required to fix a bug into source code

Delegation

Systems that Intensively Use Built-in Features and Code Smells

RG: Systems that Intensively Use Built-in Features and Code Smells



RG: Systems that Intensively Use Built-in Features and Code Smells

Built-in

The **Two Code Smells Most Detected** (Complex List Comprehension and Long Ternary Conditional) are due to **sub-optimal use** of **Built-in Features**

70% of the Cases **Code Smells** are **Introduced** Due To **Evolutionary Activities**

Code Smells


Code Quality Over Time
On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality
The Yin and Yang of Software Quality: On the Relationship between Design Patterns and Code Smells
On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort
Key Findings
Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software

Key Findings

Code Quality Over Time

Programming Abstractions evolves over time, but non in a statistically significant way

Inheritance and Delegation Most of Time **Statistically** Contribute to the **Decrease** of **Code Smell Severity**

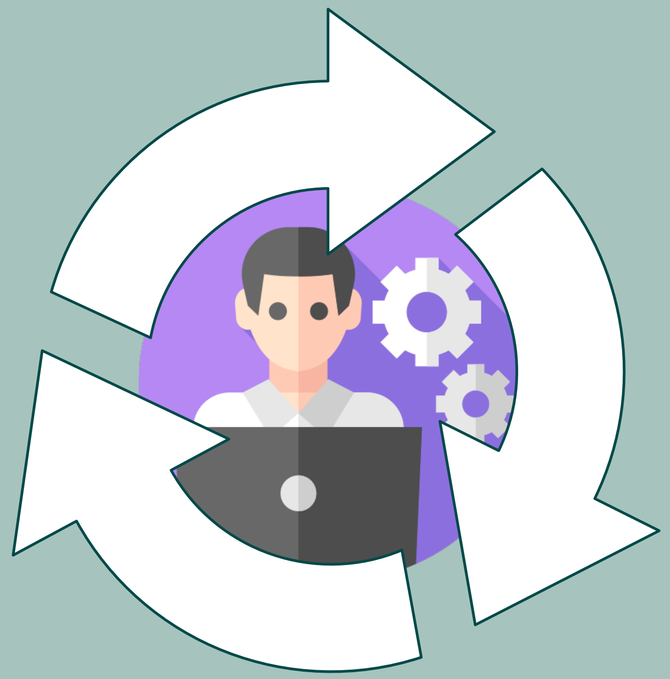
Programming Abstraction statistically contributes to reducing the effort required to fix a bug in source code

A Considerable Number of Design Patterns are Statistically Correlated with the Emergence of Code Smells

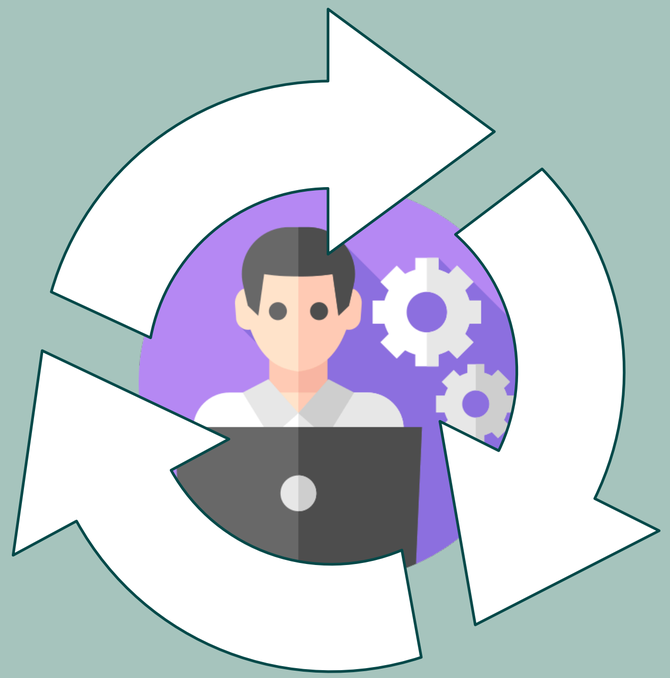
Programming Abstraction does not statistically contribute to decrease the defect proneness

The **Two Code Smells Most Detected** (Complex List Comprehension and Long Ternary Conditional) are due to **sub-optimal use** of **Built-in Features**

Background & Context

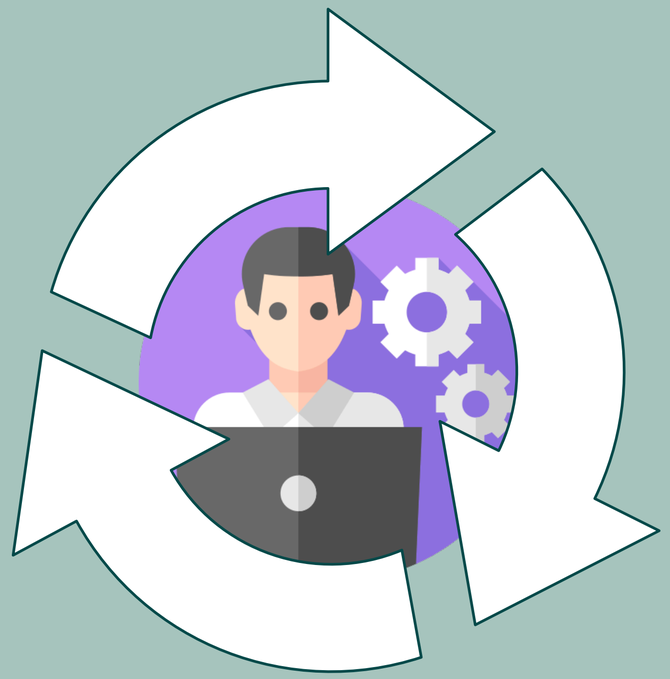


Background & Context



Cost

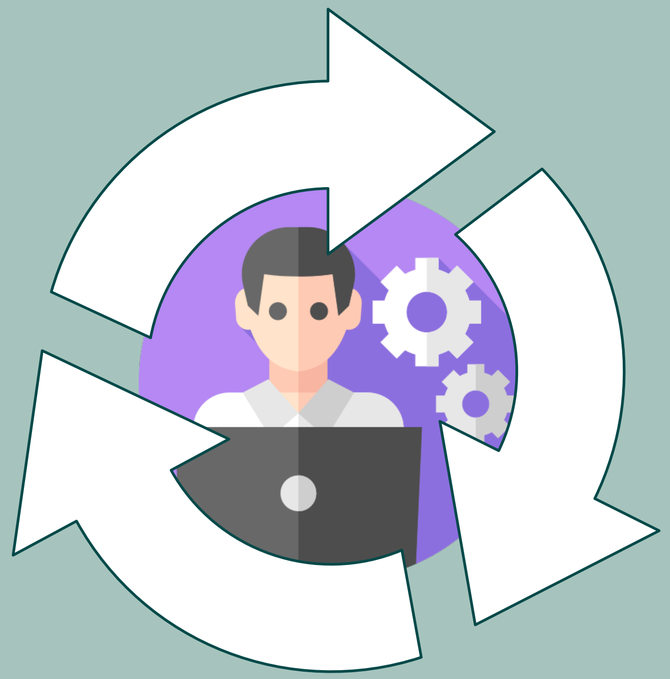
Background & Context



Cost

Time

Background & Context

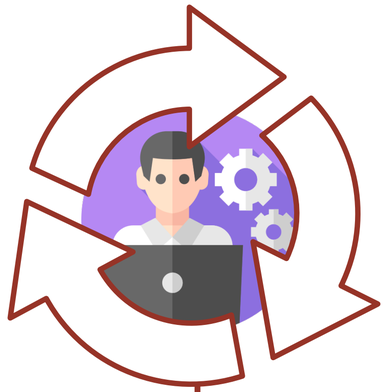


Cost

Time

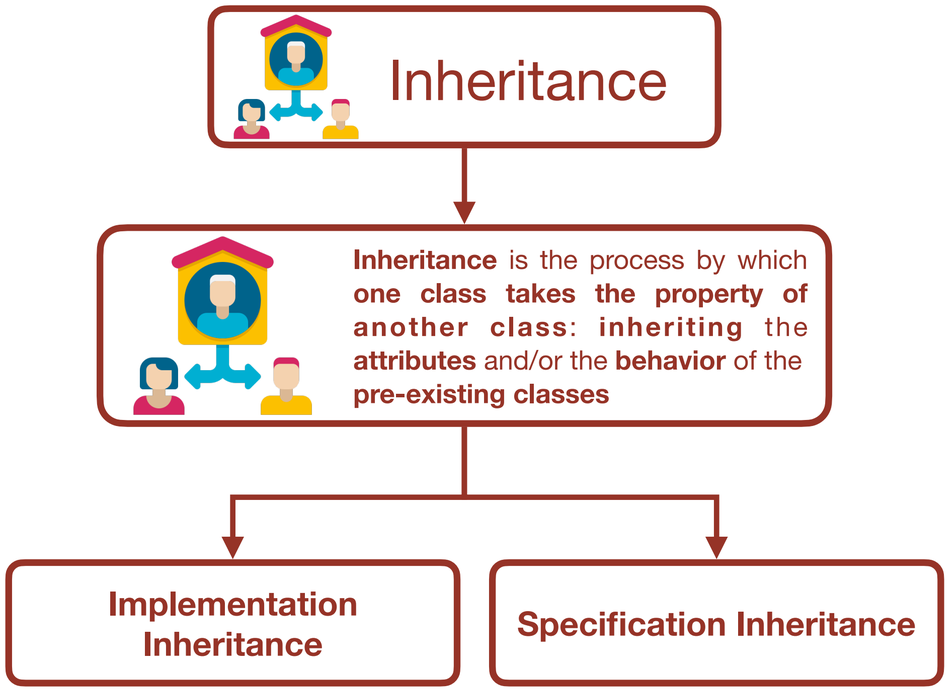
Effort

Background & Context



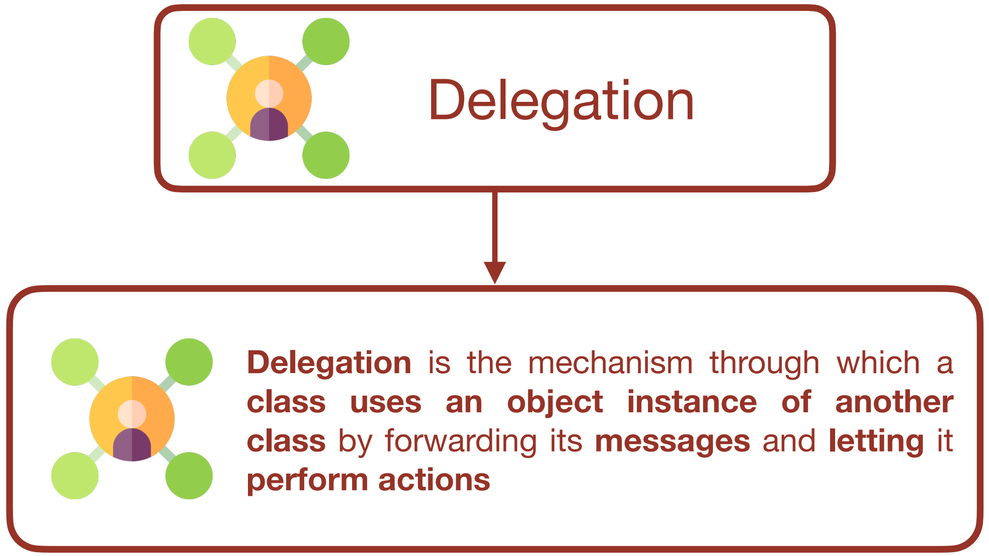
- Program Abstraction
- Third-Party Libraries
- Design Patterns
- System Wrapping

Program Abstraction



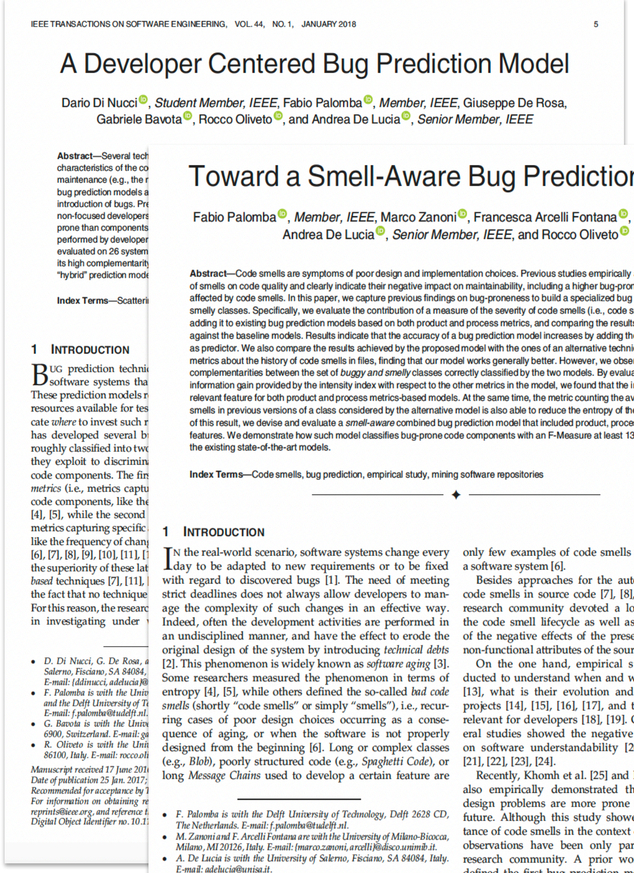

```
public class A{
        . . .
}

Public class B extend A {
              . . .
}
```

Program Abstraction




```
public class A{
 public int foo (int x) {
      return b.bar(x);
  }

}
```

State-of-The-Art



IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 1, JANUARY 2018 5

A Developer Centered Bug Prediction Model

Dario Di Nucci, *Student Member, IEEE*, Fabio Palomba, *Member, IEEE*, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia, *Senior Member, IEEE*

Abstract—Several tech characteristics of the co maintenance (e.g., the r bug prediction models a introduction of bugs. Pre non-focused developers prone than components performed by developer evaluated on 26 system its high complementarity "hybrid" prediction mode

Index Terms—Scatteri

1 Introduction

BUG prediction techni software systems tha These prediction models r resources available for tes cate *where* to invest such r has developed several bi roughly classified into tw they exploit to discrimin code components. The fir *metrics* (i.e., metrics captu code components, like the [4], [5], while the second metrics capturing specific like the frequency of chan [6], [7], [8], [9], [10], [11], [the superiority of these lat *based* techniques [7], [11], the fact that no technique For this reason, the researc in investigating under

• D. Di Nucci, G. De Rosa, Salerno, Fisciano, SA 84084, E-mail: {ddinucci, adelucia}@
• F. Palomba is with the Univ and the Delft University of Te E-mail: f.palomba@tudelft.nl.
• G. Bavota is with the Univ 6900, Switzerland. E-mail: ga
• R. Oliveto is with the Uni 86100, Italy. E-mail: rocco.ol

Manuscript received 17 June 201
Date of publication 25 Jan. 2017;
Recommended for acceptance by
For information on obtaining re
reprints@ieee.org, and reference t
Digital Object Identifier no. 10.1

Toward a Smell-Aware Bug Prediction Model

Fabio Palomba, *Member, IEEE*, Marco Zanoni, Francesca Arcelli Fontana, *Member, IEEE*, Andrea De Lucia, *Senior Member, IEEE*, and Rocco Oliveto

Abstract—Code smells are symptoms of poor design and implementation choices. Previous studies empirically assessed the impact of smells on code quality and clearly indicate their negative impact on maintainability, including a higher bug-proneness of components affected by code smells. In this paper, we capture previous findings on bug-proneness to build a specialized bug prediction model for smelly classes. Specifically, we evaluate the contribution of a measure of the severity of code smells (i.e., code smell intensity) by adding it to existing bug prediction models based on both product and process metrics, and comparing the results of the new model against the baseline models. Results indicate that the accuracy of a bug prediction model increases by adding the code smell intensity as predictor. We also compare the results achieved by the proposed model with the ones of an alternative technique which considers metrics about the history of code smells in files, finding that our model works generally better. However, we observed interesting complementarities between the set of *buggy and smelly* classes correctly classified by the two models. By evaluating the actual information gain provided by the intensity index with respect to the other metrics in the model, we found that the intensity index is a relevant feature for both product and process metrics-based models. At the same time, the metric counting the average number of code smells in previous versions of a class considered by the alternative model is also able to reduce the entropy of the model. On the basis of this result, we devise and evaluate a *smell-aware* combined bug prediction model that included product, process, and smell-related features. We demonstrate how such model classifies bug-prone code components with an F-Measure at least 13 percent higher than the existing state-of-the-art models.

Index Terms—Code smells, bug prediction, empirical study, mining software repositories

1 Introduction

IN the real-world scenario, software systems change every day to be adapted to new requirements or to be fixed with regard to discovered bugs [1]. The need of meeting strict deadlines does not always allow developers to manage the complexity of such changes in an effective way. Indeed, often the development activities are performed in an undisciplined manner, and have the effect to erode the original design of the system by introducing *technical debts* [2]. This phenomenon is widely known as *software aging* [3]. Some researchers measured the phenomenon in terms of entropy [4], [5], while others defined the so-called *bad code smells* (shortly "code smells" or simply "smells"), i.e., recurring cases of poor design choices occurring as a consequence of aging, or when the software is not properly designed from the beginning [6]. Long or complex classes (e.g., *Blob*), poorly structured code (e.g., *Spaghetti Code*), or long *Message Chains* used to develop a certain feature are only few examples of code smells that can possibly affect a software system [6].

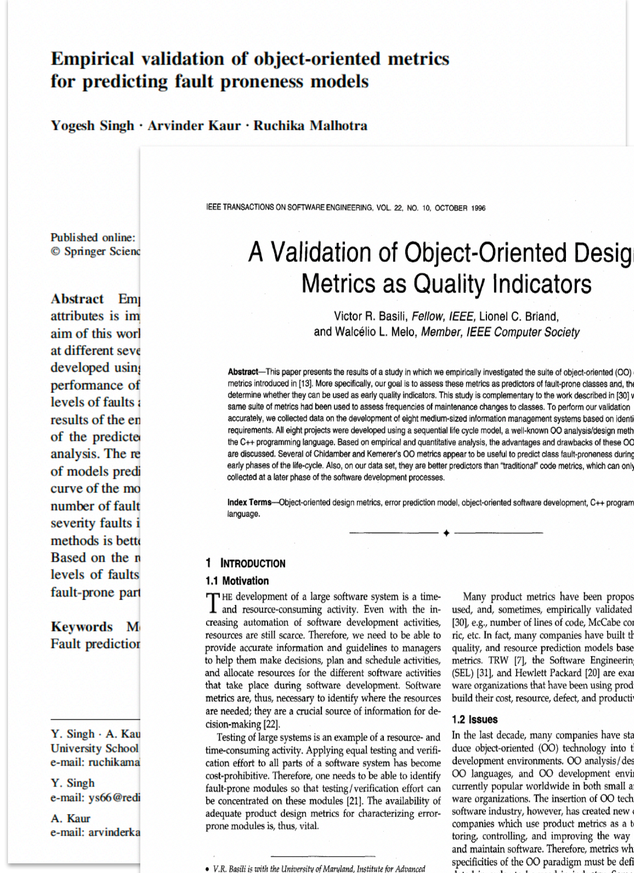
Besides approaches for the automatic identification of code smells in source code [7], [8], [9], [10], [11], [12], the research community devoted a lot of effort in studying the code smell lifecycle as well as in providing evidence of the negative effects of the presence of design flaws on non-functional attributes of the source code.

On the one hand, empirical studies have been conducted to understand when and why code smells appear [13], what is their evolution and longevity in software projects [14], [15], [16], [17], and to what extent they are relevant for developers [18], [19]. On the other hand, several studies showed the negative effects of code smells on software understandability [20] and maintainability [21], [22], [23], [24].

Recently, Khomh et al. [25] and Palomba et al. [26] have also empirically demonstrated that classes affected by design problems are more prone to contain bugs in the future. Although this study showed the potential importance of code smells in the context of bug prediction, these observations have been only partially explored by the research community. A prior work by Taba et al. [27] defined the first bug prediction model that includes code smell information. In particular, they defined three metrics, coined as *antipattern metrics*, based on the history of code smells in files and able to quantify the average number of antipatterns, the complexity of changes involving antipatterns and their recurrence length. Then, a bug prediction model exploiting antipattern measures besides structural metrics was devised and evaluated, showing that the

• F. Palomba is with the Delft University of Technology, Delft 2628 CD, The Netherlands. E-mail: f.palomba@tudelft.nl.
• M. Zanoni and F. Arcelli Fontana are with the University of Milano-Bicocca, Milano, MI 20126, Italy. E-mail: {marco.zanoni, arcelli}@disco.unimib.it.
• A. De Lucia is with the University of Salerno, Fisciano, SA 84084, Italy. E-mail: adelucia@unisa.it.
• R. Oliveto is with the University of Molise, Campobasso 86100, Italy. E-mail: rocco.oliveto@unimol.it.

Manuscript received 12 Jan. 2017; revised 11 Sept. 2017; accepted 12 Oct. 2017. Date of publication 6 Nov. 2017; date of current version 21 Feb. 2019. (Corresponding author: Fabio Palomba.)
Recommended for acceptance by T. Zimmermann.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2770122



Empirical validation of object-oriented metrics for predicting fault proneness models

Yogesh Singh · Arvinder Kaur · Ruchika Malhotra

Published online:
© Springer Scienc

Abstract Emp
attributes is im
aim of this worl
at different seve
developed usin
performance of
levels of faults
results of the en
of the predicte
analysis. The re
of models predi
curve of the mo
number of fault
severity faults i
methods is bett
Based on the n
levels of faults
fault-prone part

Keywords M
Fault prediction

Y. Singh · A. Kau
University School
e-mail: ruchikama

Y. Singh
e-mail: ys66@redi

A. Kaur
e-mail: arvinderka

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 22, NO. 10, OCTOBER 1996 751

A Validation of Object-Oriented Design Metrics as Quality Indicators

Victor R. Basili, *Fellow, IEEE*, Lionel C. Briand, and Walcélio L. Melo, *Member, IEEE Computer Society*

Abstract—This paper presents the results of a study in which we empirically investigated the suite of object-oriented (OO) design metrics introduced in [13]. More specifically, our goal is to assess these metrics as predictors of fault-prone classes and, therefore, determine whether they can be used as early quality indicators. This study is complementary to the work described in [30] where the same suite of metrics had been used to assess frequencies of maintenance changes to classes. To perform our validation accurately, we collected data on the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method and the C++ programming language. Based on empirical and quantitative analysis, the advantages and drawbacks of these OO metrics are discussed. Several of Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle. Also, on our data set, they are better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes.

Index Terms—Object-oriented design metrics, error prediction model, object-oriented software development, C++ programming language.

1 Introduction

1.1 Motivation

THE development of a large software system is a time- and resource-consuming activity. Even with the increasing automation of software development activities, resources are still scarce. Therefore, we need to be able to provide accurate information and guidelines to managers to help them make decisions, plan and schedule activities, and allocate resources for the different software activities that take place during software development. Software metrics are, thus, necessary to identify where the resources are needed; they are a crucial source of information for decision-making [22].

Testing of large systems is an example of a resource- and time-consuming activity. Applying equal testing and verification effort to all parts of a software system has become cost-prohibitive. Therefore, one needs to be able to identify fault-prone modules so that testing/verification effort can be concentrated on these modules [21]. The availability of adequate product design metrics for characterizing error-prone modules is, thus, vital.

Many product metrics have been proposed [16], [26], used, and, sometimes, empirically validated [3], [4], [19], [30], e.g., number of lines of code, McCabe complexity metric, etc. In fact, many companies have built their own cost, quality, and resource prediction models based on product metrics. TRW [7], the Software Engineering Laboratory (SEL) [31], and Hewlett Packard [20] are examples of software organizations that have been using product metrics to build their cost, resource, defect, and productivity models.

1.2 Issues

In the last decade, many companies have started to introduce object-oriented (OO) technology into their software development environments. OO analysis/design methods, OO languages, and OO development environments are currently popular worldwide in both small and large software organizations. The insertion of OO technology in the software industry, however, has created new challenges for companies which use product metrics as a tool for monitoring, controlling, and improving the way they develop and maintain software. Therefore, metrics which reflect the specificities of the OO paradigm must be defined and validated in order to be used in industry. Some studies have concluded that "traditional" product metrics are not sufficient for characterizing, assessing, and predicting the quality of OO software systems. For example, in [12] it was reported that McCabe cyclomatic complexity appeared to be an inadequate metric for use in software development based on OO technology.

To address this issue, OO metrics have recently been proposed in the literature [1], [6], [13]. However, with a few exceptions [10], [30], most of them have not undergone an

• V.R. Basili is with the University of Maryland, Institute for Advanced Computer Studies and Computer Science Dept., A.V. Williams Bldg., College Park, MD 20742. E-mail: basili@cs.umd.edu.
• L.C. Briand is with Fraunhofer-Institute for Experimental Software Engineering, Technologiepark II, Sauerwiesen 6, D-67661, Kaiserslautern, Germany. E-mail: briand@iese.fhg.de.
• W.L. Melo is with the Centre de Recherche Informatique de Montréal, 1801 McGill College Ave., Montréal, Québec, H3A 2N4, Canada. E-mail: wmelo@crim.ca.

Manuscript received Sept. 7, 1995; accepted Aug. 30, 1996.
Recommended for acceptance by H. Muller.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S95819.

Improving change prediction models with code smell-related information

Gemma Catolino[1] · Fabio Palomba[2] · Francesca Arcelli Fontana[3] · Andrea De Lucia[1] · Andy Zaidman[4]

Published online: 2 A
© Springer Science+B

Abstract
Code smells are
effect of negative
Based on this co
tion can be effect
i.e., models havi
future. We exploi
the severity of a c
in the context of
and developer-ba
model with a mod
puted considering
performance of t
(ii) the intensity
onality between t
by the models re
and evaluate a sm
developer-based,
notably higher th

Keywords Chang

1 Introductio

During maintena
to adapt them to
them from potent

Communicated by: I
✉ Gemma Catolin
gcatolino@unis

Predicting Fault-Proneness using OO Metrics An Industrial Case Study

Ping Yu
Network Service Management, Alcatel Canada Inc,
400-4190 Still Creek Dr.
Burnaby, BC, V5C 6C6, Canada
p.yu@alcatel.com

Tarja Systä
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
tsysta@cs.tut.fi

Hausi Müller
Department of Computer Science
University of Victoria
P.O. Box 3055, Victoria, BC, V8W 3P6, Canada
hausi@csr.uvic.ca

Abstract

Software quality is an important external software attribute that is difficult to measure objectively. In this case study, we empirically validate a set of object-oriented metrics in terms of their usefulness in predicting fault-proneness, an important software quality indicator. We use a set of ten software product metrics that relate to the following software attributes: the size of the software, coupling, cohesion, inheritance, and reuse. Eight hypotheses on the correlations of the metrics with fault-proneness are given. These hypotheses are empirically tested in a case study, in which the client side of a large network service management system is studied. The subject system is written in Java and it consists of 123 classes. The validation is carried out using two data analysis techniques: regression analysis and discriminant analysis.

1. Introduction

The unprecedented growth of the software industry has placed an urgent call for the modernization of software engineering practices, i.e., from the current "craft" environment into something more closely resembling conventional engineering [6]. To make the modernization possible, *measures* or *metrics* of software design and process attributes are needed. Software metrics provide quantitative information that can be used in many ways to make assessments of the software products and the development processes, to help engineers in coding practices and project managers in decision making. In software forward engineering, software metrics are traditionally used to revise an improper design in an early phase of the software life cycle. Inadequacies and defects found can be modified and revised with considerably less costs and efforts than in later design phases or during software maintenance.

Metrics data provides quick feedback to software engineers. By analyzing the collected data, complexity and design quality as well as some other properties of the final software can be predicted. This early feedback enables software designers and developers to correct the inadequacies in their design or implementation without too much effort. If appropriately used, it can lead to a significant reduction in costs of the overall implementation and improvements in quality of the final software product. The improved quality, in turn, reduces future maintenance efforts. The metrics data also helps project managers to make decisions on project priorities, personnel assignments, cost organization, and other resource allocations. However, information available in the early design phase is often inaccurate and insufficient. In fact, many useful metrics can not be used during the early design phase. Because of this, information of software design presented by metric data needs to be revisited and acquired in various phases of the software life cycle.

During the software testing phase, software metrics are particularly useful. With the metrics data, various software quality attributes can be unveiled to optimize resource allocation for testing. For instance, more time or personnel

State-of-The-Art

Traditional Systems!

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 44, NO. 1, JANUARY 2018 5

A Developer Centered Bug Prediction Model

Dario Di Nucci, *Student Member, IEEE*, Fabio Palomba, *Member, IEEE*, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia, *Senior Member, IEEE*

Abstract—Several tech
characteristics of the co
maintenance (e.g., the
bug prediction models a
introduction of bugs. Pr
non-focused developers

Toward a Smell-Aware Bug Prediction Model

Fabio Palomba, *Member, IEEE*, Marco Zanoni, Francesca Arcelli Fontana, *Member, IEEE*

Empirical validation of object-oriented metrics for predicting fault proneness models

Yogesh Singh · Arvinder Kaur · Ruchika Malhotra

Improving change prediction models with code smell-related information

Gemma Catolino[1] · Fabio Palomba[2] · Francesca Arcelli Fontana[3] · Andrea De Lucia[1] · Andy Zaidman[4]

Published online: 2 A
© Springer Science+B

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 22, NO. 10, OCTOBER 1996 751

Predicting Fault-Proneness using OO Metrics

metrics in terms of their usefulness in predicting fault-proneness, an important software quality indicator. We use a set of ten software product metrics that relate to the following software attributes: the size of the software, coupling, cohesion, inheritance, and reuse. Eight hypotheses on the correlations of the metrics with fault-proneness are given. These hypotheses are empirically tested in a case study, in which the client side of a large network service management system is studied. The subject system is written in Java and it consists of 123 classes. The validation is carried out using two data analysis techniques: regression analysis and discriminant analysis.

1. Introduction

The unprecedented growth of the software industry has placed an urgent call for the modernization of software engineering practices, i.e., from the current "craft" environment into something more closely resembling conventional engineering [6]. To make the modernization possible, *measures* or *metrics* of software design and process attributes are needed. Software metrics provide quantitative information that can be used in many ways to make assessments of the software products and the development processes, to help engineers in coding practices and later design phases or during software maintenance.

Metrics data provides quick feedback to software engineers. By analyzing the collected data, complexity and design quality as well as some other properties of the final software can be predicted. This early feedback enables software designers and developers to correct the inadequacies in their design or implementation without too much effort. If appropriately used, it can lead to a significant reduction in costs of the overall implementation and improvements in quality of the final software product. The improved quality, in turn, reduces future maintenance efforts. The metrics data also helps project managers to make decisions on project priorities, personnel assignments, cost organization, and other resource allocations. However, information available in the early design phase is often inaccurate and insufficient. In fact, many useful metrics can not be used during the early design phase. Because of this, information of software design presented by metric data needs to be revisited and acquired in various phases of the software life cycle.

During the software testing phase, software metrics are particularly useful. With the metrics data, various software quality attributes can be unveiled to optimize resource allocation for testing. For instance, more time or personnel

1

Research Gaps

Research Gaps

There is a **lack of empirical knowledge** on how **reusability** mechanisms **vary** in **complex systems**

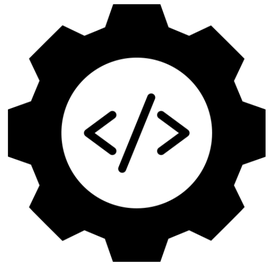
Research Gaps

There is a **lack of empirical knowledge** on how **reusability** mechanisms **vary** in **complex systems**

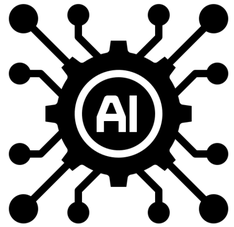
Most studies on the impact of reusability on **code quality** and **reliability** focused only on **traditional systems**

Problem Relevance

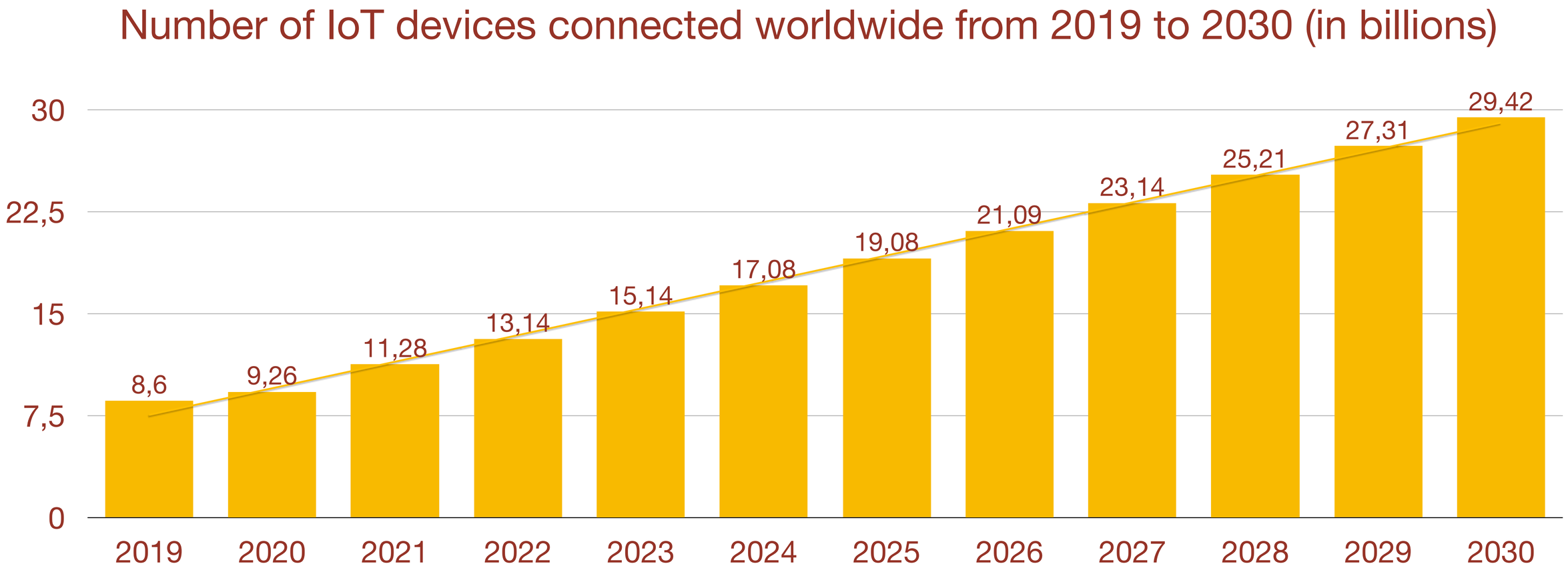
Traditional Systems



Complex Systems



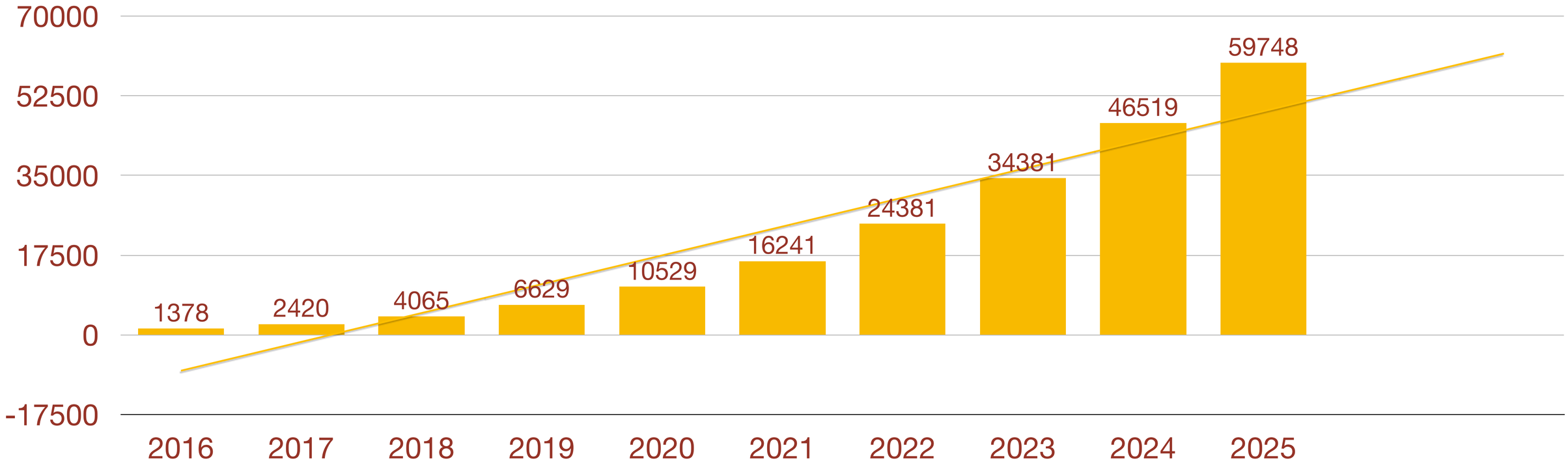
Problem Relevance



https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

Problem Relevance

Revenues from the artificial intelligence market worldwide, from 2016 to 2025 (in millions U.S. Dollars)



https://blog.softwaremill.com/how-to-adopt-artificial-intelligence-in-your-business-65d42e2293b9


MIND THE GAP
Fill

Objective

The **objective** of this Ph.D. project is to understand what is the **impact on code quality and reliability in Complex Systems**

2 Research Questions

Research Questions

What are the differences in terms of reusability between traditional systems and complex systems?

What is the impact of software reuse on code quality and reliability in complex systems?

Research Questions

What are the differences in terms of reusability between traditional systems and complex systems?

What is the impact of software reuse on code quality and reliability in complex systems?

Mixed Methods

Research Questions

What are the differences in terms of reusability between traditional systems and complex systems?

What is the impact of software reuse on code quality and reliability in complex systems?

Mixed Methods

Research Questions

What are the differences in terms of reusability between traditional systems and complex systems?

What is the impact of software reuse on code quality and reliability in complex systems?

Mixed Methods { **Mining Software Repository**

Research Questions

What are the differences in terms of reusability between traditional systems and complex systems?

What is the impact of software reuse on code quality and reliability in complex systems?

Mixed Methods

- **Mining Software Repository**
- **Qualitative Analyses**

Research Questions

What are the differences in terms of reusability between traditional systems and complex systems?

What is the impact of software reuse on code quality and reliability in complex systems?

Mixed Methods

- **Mining Software Repository**
- **Qualitative Analyses**
- **Semi-Structured Interview**

Research Questions

What are the differences in terms of reusability between traditional systems and complex systems?

What is the impact of software reuse on code quality and reliability in complex systems?

Mixed Methods

- **Mining Software Repository**
- **Qualitative Analyses**
- **Semi-Structured Interview**
- **Survey**

RQ1- On the difference between Traditional and Complex Systems

1. An empirical analysis of the use of reusability mechanisms in traditional vs complex systems

2. Survey with developers to understand how they adopt reusability mechanisms in traditional vs complex systems

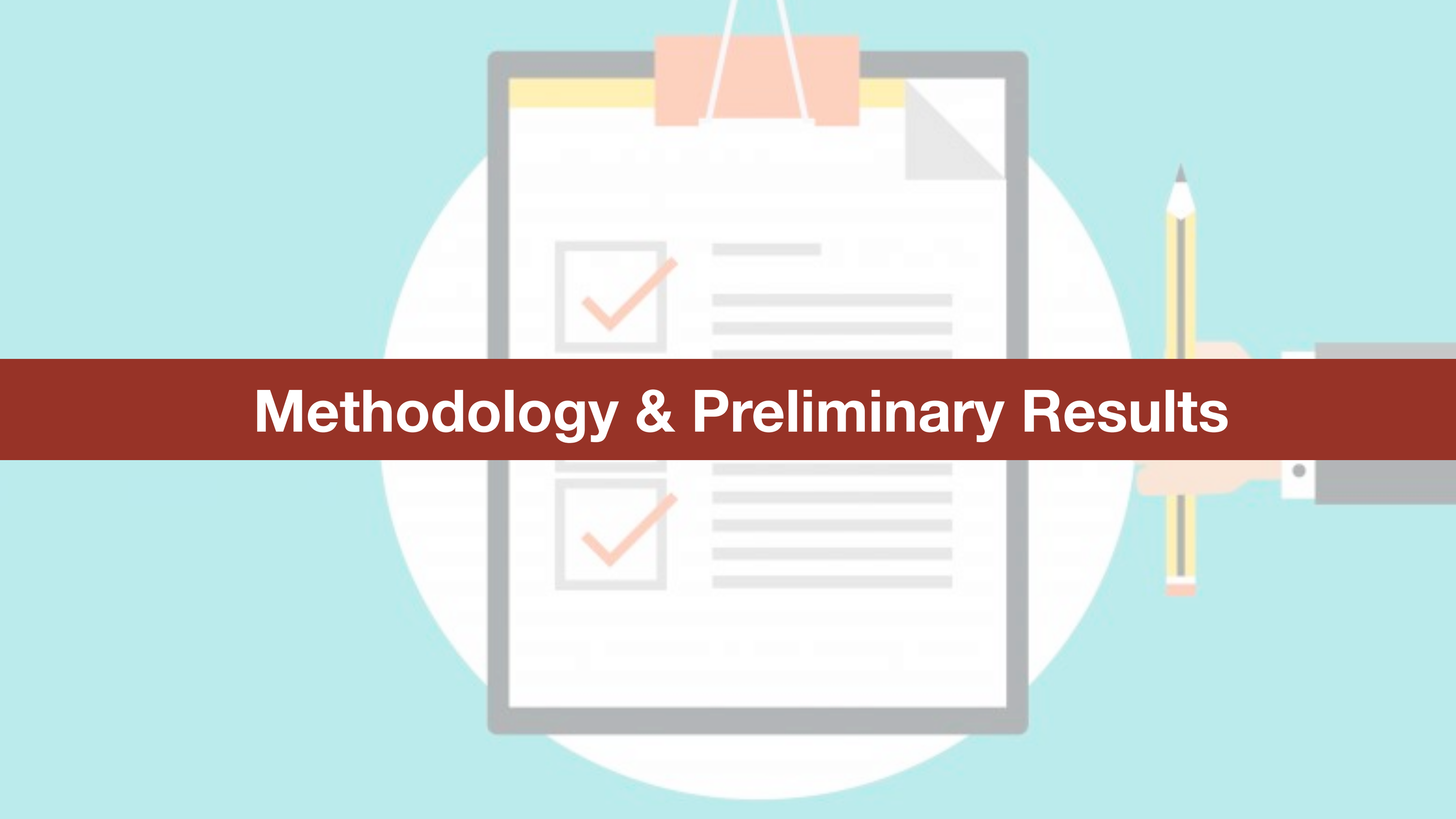
RQ2 - On the impact on code quality and reliability in Complex Systems

1. On the impact of program abstraction, third-party libraries, and design patterns in **complex systems** on **code smells**, **defect proneness**, and **maintenance effort**

2. An **industrial case study** with the support of **Business Solution S.L.R.** on the impact of reuse in complex systems

Final Outcome

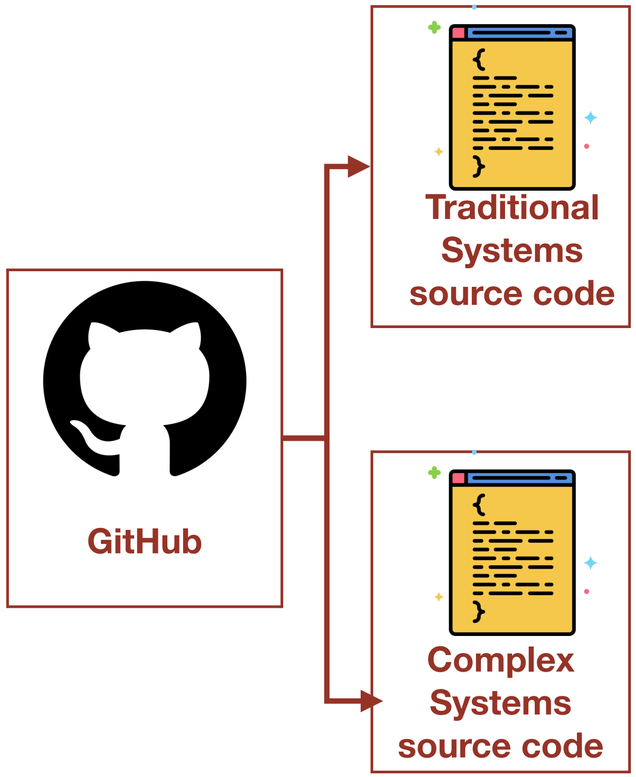
The **outcome** of this Ph.D. project is to provide a **recommended system** that practitioners can use to identify which reusability mechanism have to adopt to increment **code quality** and **reliability** in complex systems



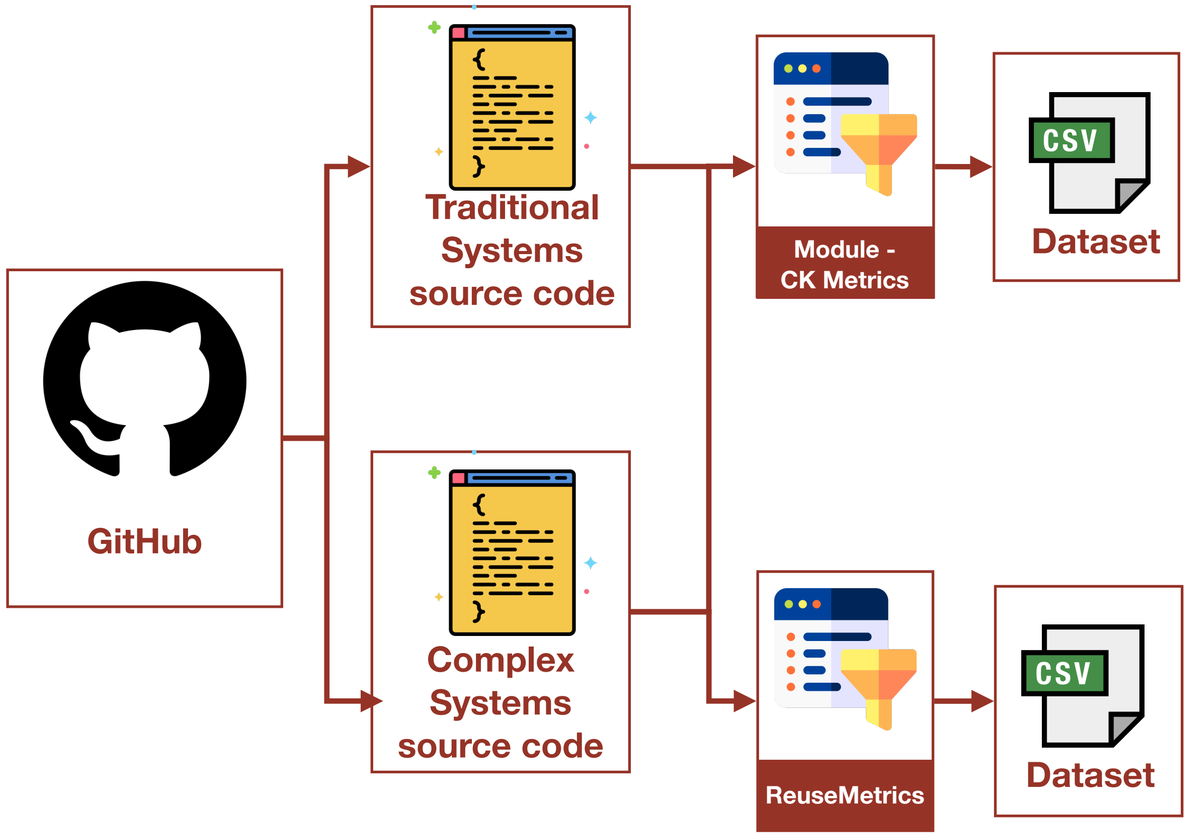
Methodology & Preliminary Results

RQ1 - On the difference between Traditional and Complex Systems

RQ1 - On the difference between Traditional and Complex Systems



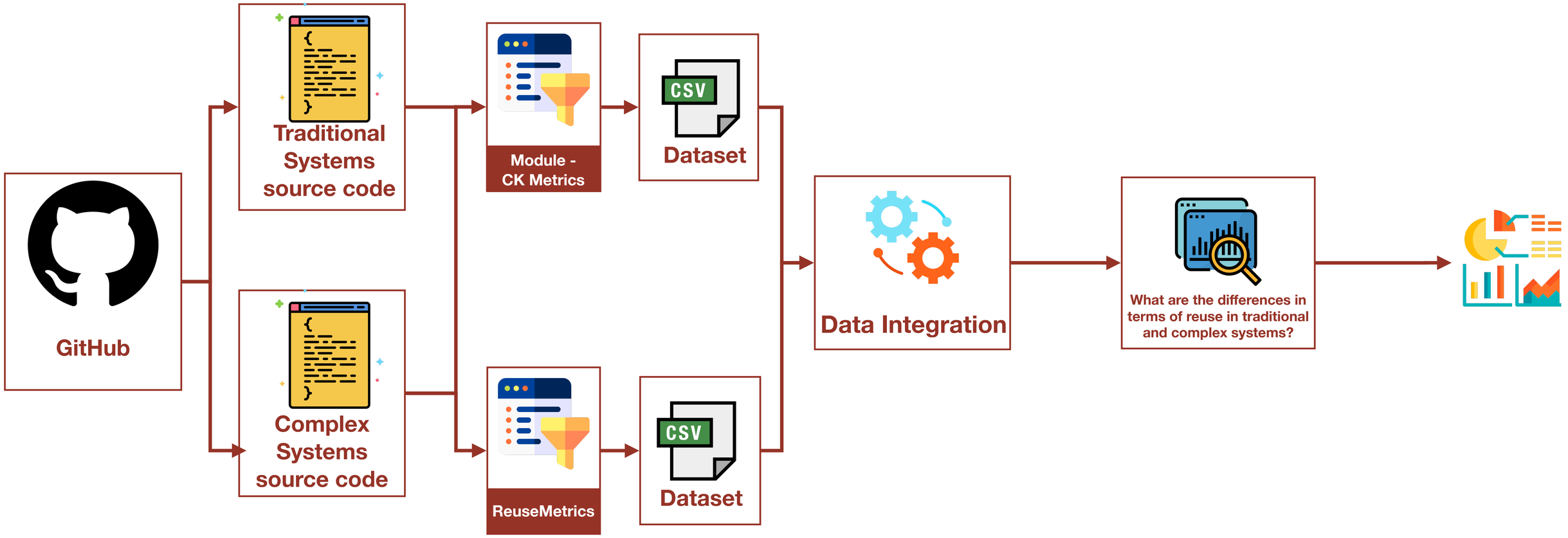
RQ1 - On the difference between Traditional and Complex Systems



0%

100%

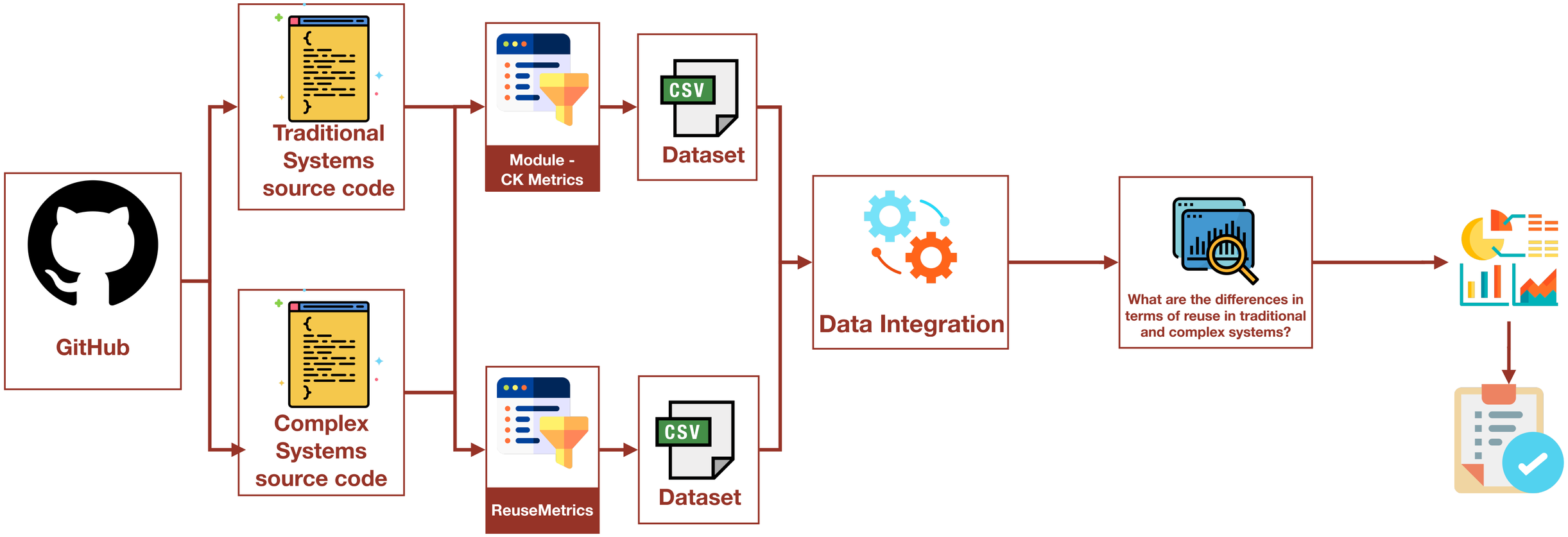
RQ1 - On the difference between Traditional and Complex Systems



0%

100%

RQ1 - On the difference between Traditional and Complex Systems



The use of **implementation** and **specification inheritance** followed an “increasing-decreasing”

The **design erosion** observed by Lehman is **confirmed**

On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

Giammaria Giordano · **Gerardo Festa** · **Gemma Catolino** · **Fabio Palomba** · **Filomena Ferrucci** · **Carmine Gravino**

Received: date / Accepted: date

Abstract Software reusability mechanisms, like inheritance and delegation in Object-Oriented programming, are widely recognized as key instruments of software design that reduce the risks of source code being affected by defects, other than to reduce the effort required to maintain and evolve source code. Previous work has traditionally employed source code reuse metrics for prediction purposes, e.g., in the context of defect prediction. However, our research identifies two noticeable limitations of the current literature. First, still little is known about the extent to which developers actually employ code reuse mechanisms over time. Second, it is still unclear how these mechanisms may contribute to explaining defect-proneness and maintenance effort during software evolution. We aim at bridging this gap of knowledge, as an improved understanding of these aspects might provide insights into the actual support provided by these mechanisms, e.g., by suggesting whether and how to use them for prediction purposes. We propose an exploratory study, conducted on 12 JAVA projects—over 9000 commits—of the DEFECTS4J dataset, aiming at (1) assessing how developers use inheritance and delegation during software evolution; and (2) statistically analyzing the impact of inheritance and delegation on fault proneness and maintenance effort. Our results let emerge various usage patterns that describe the way inheritance and delegation vary over time. In addition, we find out that inheritance and delegation are statistically significant factors that influence both source code defect-proneness and maintenance effort.

Keywords Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.

Giammaria Giordano, Gerardo Festa, Fabio Palomba, Filomena Ferrucci, Carmine Gravino
Software Engineering (SeSa) Lab - University of Salerno (Italy) — E-mail: giagiordano@unisa.it, g.festa22@studenti.unisa.it, fpalomba@unisa.it, fferucci@unisa.it

Gemma Catolino
Jheronimus Academy of Data Science & Tilburg University, The Netherlands — E-mail: g.catolino@tilburguniversity.edu

Under Review-Empirical Software Engineering (EMSE)

Delegation and Inheritance evolve over time, but not in a statistically significant manner

On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Giammaria Giordano,[1] Antonio Fasulo,[1] Gemma Catolino,[2]
Fabio Palomba,[1] Filomena Ferrucci,[1] Carmine Gravino[1]
[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
giagiordano@unisa.it, a.fasulo@studenti.unisa.it, g.catolino@tilburguniversity.edu
fpalomba@unisa.it, gravino@unisa.it, fferrucci@unisa.it

***Abstract*—Source code reuse is considered one of the holy grails of modern software development. Indeed, it has been widely demonstrated that this activity decreases software development and maintenance costs while increasing its overall trustworthiness. The Object-Oriented (OO) paradigm provides different internal mechanisms to favor code reuse, i.e., specification inheritance, implementation inheritance, and delegation. While previous studies investigated how inheritance relations impact source code quality, there is still a lack of understanding of their evolutionary aspects and, more particular, of how these mechanisms may impact source code quality over time. To bridge this gap of knowledge, this paper proposes an empirical investigation into the evolution of specification inheritance, implementation inheritance, and delegation and their impact on the variability of source code quality attributes. First, we assess how the implementation of those mechanisms varies over 15 releases of three software systems. Second, we devise a statistical approach with the aim of understanding how inheritance and delegation let source code quality—as indicated by the severity of code smells—vary in either positive or negative manner. The key results of the study indicate that inheritance and delegation evolve over time, but not in a statistically significant manner. At the same time, their evolution often leads code smell severity to be reduced, hence possibly contributing to improve code maintainability.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.**

I. Introduction

Software reusability refers to the development practice through which developers make use of existing code when implementing new functionalities [1], [2]. This is widely considered as a best practice, as it leads developers to save time, energy, and maintenance costs, other than relying on source code that has been previously tested [3], [4].

Contemporary Object-Oriented (OO) programming languages, *e.g.,* Java, provide developers with various mechanisms supporting code reusability: examples are design patterns [5], [6], the use of third-party libraries [7], [8], and programming abstractions [9]. These latter, in particular, have caught the attention of researchers since the rise of object-orientation and were found to be a valuable element to increase software quality and reusability [10], [11], [12], [13], [14].

When focusing on Java, there are two well-known abstraction mechanisms such as *inheritance* and *delegation* [15].

Inheritance is the process by which one class takes the property of another class: the new classes, known as derived or children classes, inherit the attributes and/or the behavior of the pre-existing classes, which are referred to as base, super, or parent classes. Delegation is, instead, the mechanism through which a class uses an object instance of another class by forwarding it messages and letting it performing actions [15].

The importance of inheritance and delegation has been remarked multiple times by the research community. In 1994, Chidamber and Kemerer [16] included in their Object-Oriented metric suite the Depth of the Inheritance Tree (DIT) metric, a measure of the number of classes that inherit from one another. Later on, various metric catalogs proposed variations of DIT as well as other inheritance metrics [17], [18], [19]. In addition, the sub-optimal adoption of inheritance and delegation mechanisms had led to the definition and investigation of reusability-specific code smells [20], [21], [22], [23]: as an example, Fowler [24] defined the *Refused Bequest* and *Middle Man* code smells, which refer to the poor use of inheritance and delegation in Object-Oriented programs that might lead to deteriorate their code quality [22], [25], [26], [27]. These studies have also led to the definition of automated code smell detection and refactoring approaches [28], [29], [30].

Still from an empirical standpoint, a number of studies targeted the role of inheritance and delegation mechanisms for monitoring software quality. In particular, researchers devoted extensive effort on the understanding of the potential impact of those mechanisms on software metrics [31], [32], [33], maintainability effort and costs [34], [35], [36], [37], design patterns [38], [39], change-proneness [40], [41], [42], [43], and source code defectiveness [44], [45], [46], [47].

While the current body of knowledge provides compelling evidence of the value of reusability mechanisms for the analysis of source code quality properties, we can still identify a noticeable research gap: as Mens and Demeyer [48] already reported in the early 2000s, the *long-term* evolution of source code quality metrics might provide a different perspective of the nature of a software project, possibly revealing complementary or even contrasting findings with respect to the studies that investigated code metrics in a fixed point of software evolution. To the best of our knowledge, Nasseri *et al.* [49] were the only researchers studying the evolution of reusability metrics. They specifically focused on the size of the inheritance hierarchies and aimed at assessing whether

Software analysis, evolution, and Reengineering (SANER)

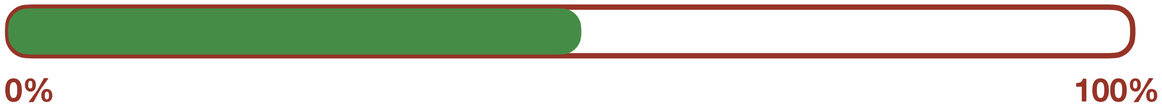
RQ1 - Survey with developers on the role of reusability mechanisms

To understand how complex systems developers use software reusability, we will conduct several surveys using Prolific and Reddit.

Example of Questions

To build complex systems, what kind of reusability mechanisms do you use?

Can you describe the mechanisms for software reusability that are most difficult to implement?



RQ1 - Survey with developers on the role of reusability mechanisms

To understand how complex systems developers use software reusability, we will conduct several surveys using Prolific and Reddit.

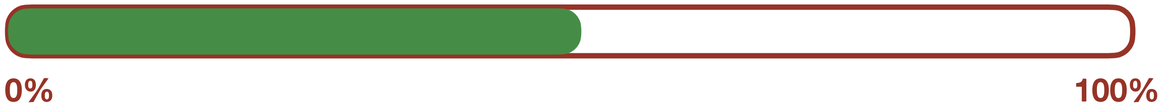
Example of Questions

Design Phase

Can you describe the mechanisms for software reusability that are most difficult to implement?

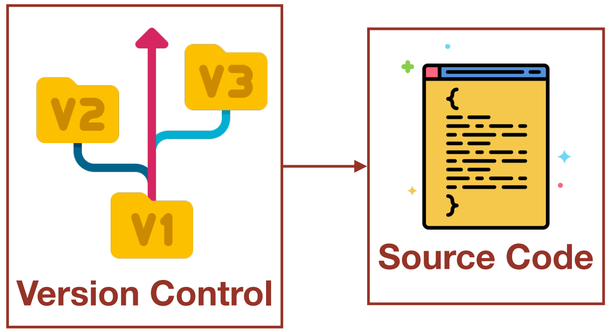
RQ1 - On the difference between Traditional and Complex Systems

At the end of the survey, we will use the information provided by developers to **replicate** studies on the evolution of **reusability** (focused on **program abstraction**, **third-party libraries**, and **design patterns**) in complex systems (AI-enabled and IoT systems)

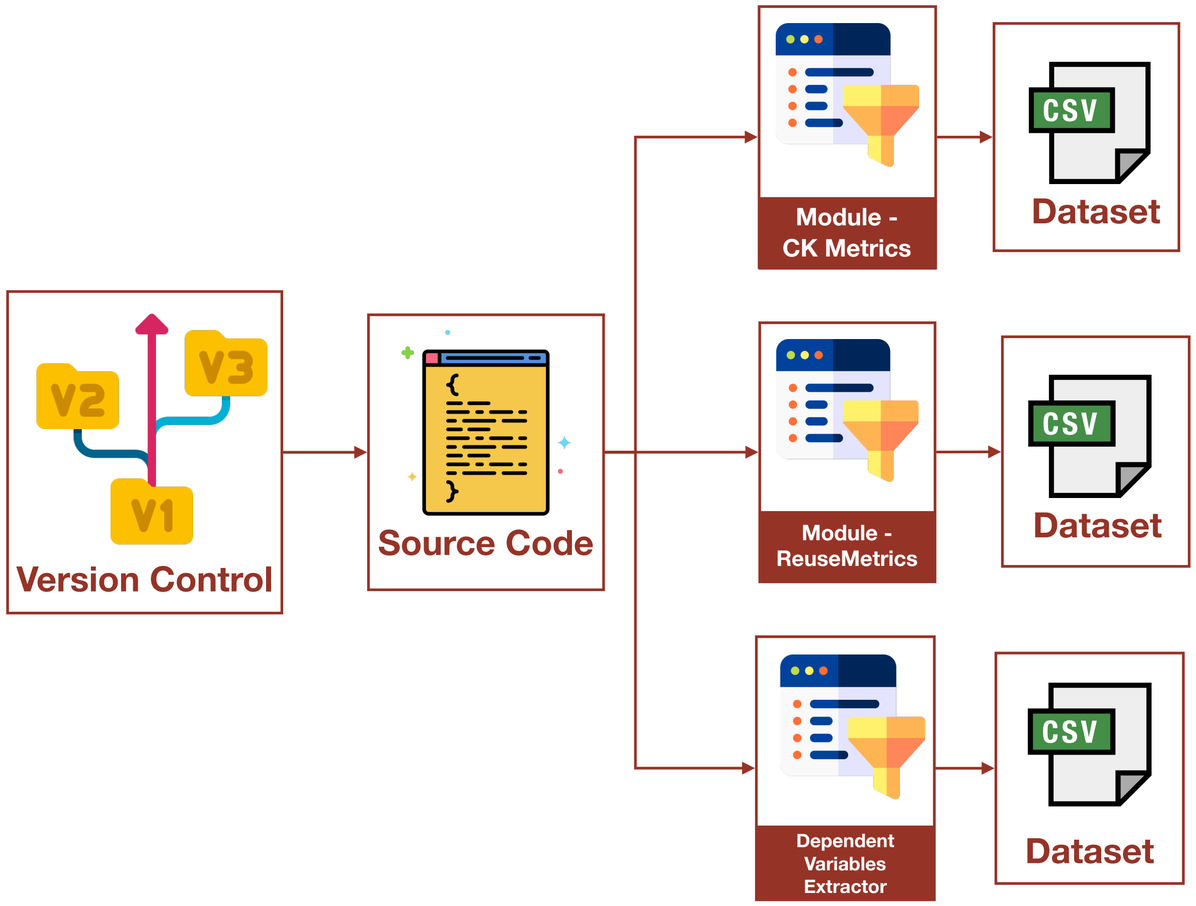


RQ2 - On the impact on code quality and reliability in Complex Systems

RQ2 - On the impact on code quality and reliability in Complex Systems



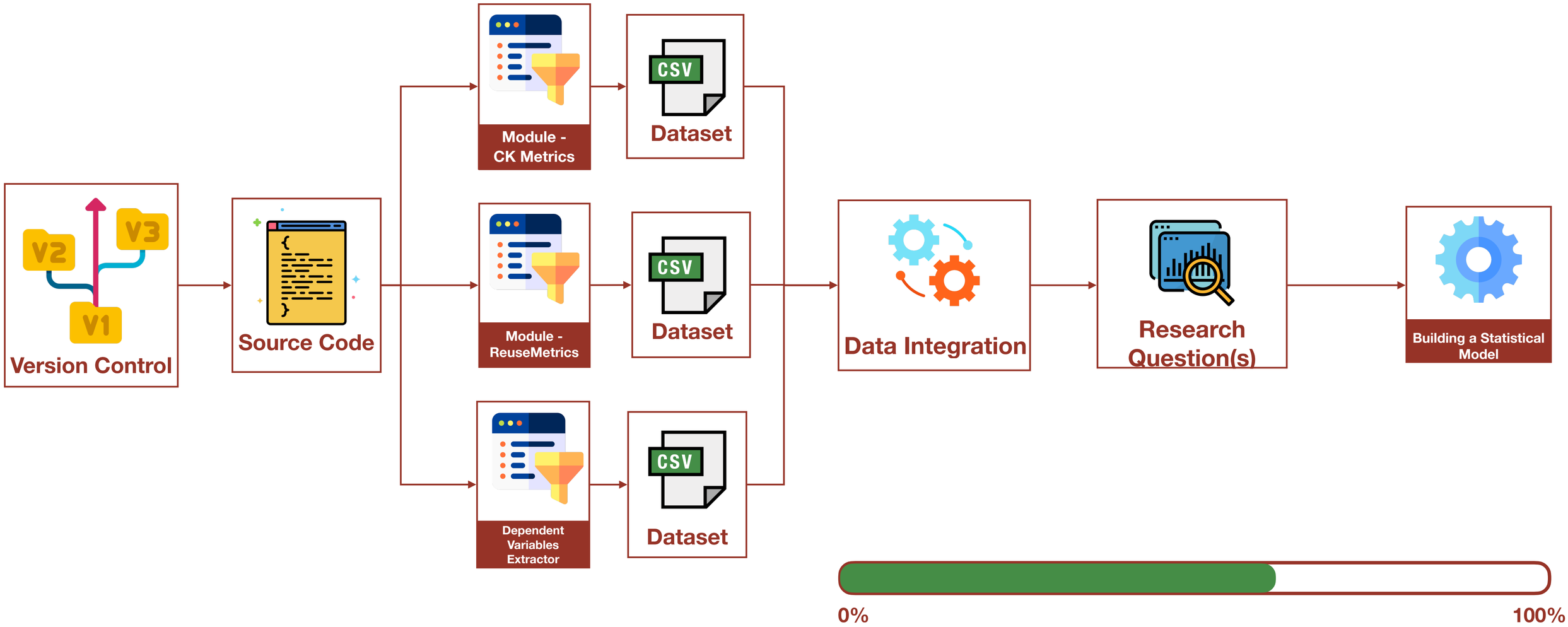
RQ2 - On the impact on code quality and reliability in Complex Systems



0%

100%

RQ2 - On the impact on code quality and reliability in Complex Systems



Delegation and inheritance positively correlate to the decrease of the code smell severity

On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Giammaria Giordano,[1] Antonio Fasulo,[1] Gemma Catolino,[2]
Fabio Palomba,[1] Filomena Ferrucci,[1] Carmine Gravino[1]
[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
giagiordano@unisa.it, a.fasulo@studenti.unisa.it, g.catolino@tilburguniversity.edu
fpalomba@unisa.it, gravino@unisa.it, fferrucci@unisa.it

***Abstract*—Source code reuse is considered one of the holy grails of modern software development. Indeed, it has been widely demonstrated that this activity decreases software development and maintenance costs while increasing its overall trustworthiness. The Object-Oriented (OO) paradigm provides different internal mechanisms to favor code reuse, i.e., specification inheritance, implementation inheritance, and delegation. While previous studies investigated how inheritance relations impact source code quality, there is still a lack of understanding of their evolutionary aspects and, more particular, of how these mechanisms may impact source code quality over time. To bridge this gap of knowledge, this paper proposes an empirical investigation into the evolution of specification inheritance, implementation inheritance, and delegation and their impact on the variability of source code quality attributes. First, we assess how the implementation of those mechanisms varies over 15 releases of three software systems. Second, we devise a statistical approach with the aim of understanding how inheritance and delegation let source code quality—as indicated by the severity of code smells—vary in either positive or negative manner. The key results of the study indicate that inheritance and delegation evolve over time, but not in a statistically significant manner. At the same time, their evolution often leads code smell severity to be reduced, hence possibly contributing to improve code maintainability.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.**

I. Introduction

Software reusability refers to the development practice through which developers make use of existing code when implementing new functionalities [1], [2]. This is widely considered as a best practice, as it leads developers to save time, energy, and maintenance costs, other than relying on source code that has been previously tested [3], [4].

Contemporary Object-Oriented (OO) programming languages, *e.g.,* Java, provide developers with various mechanisms supporting code reusability: examples are design patterns [5], [6], the use of third-party libraries [7], [8], and programming abstractions [9]. These latter, in particular, have caught the attention of researchers since the rise of object-orientation and were found to be a valuable element to increase software quality and reusability [10], [11], [12], [13], [14].

When focusing on Java, there are two well-known abstraction mechanisms such as *inheritance* and *delegation* [15].

Inheritance is the process by which one class takes the property of another class: the new classes, known as derived or children classes, inherit the attributes and/or the behavior of the pre-existing classes, which are referred to as base, super, or parent classes. Delegation is, instead, the mechanism through which a class uses an object instance of another class by forwarding it messages and letting it performing actions [15].

The importance of inheritance and delegation has been remarked multiple times by the research community. In 1994, Chidamber and Kemerer [16] included in their Object-Oriented metric suite the Depth of the Inheritance Tree (DIT) metric, a measure of the number of classes that inherit from one another. Later on, various metric catalogs proposed variations of DIT as well as other inheritance metrics [17], [18], [19]. In addition, the sub-optimal adoption of inheritance and delegation mechanisms had led to the definition and investigation of reusability-specific code smells [20], [21], [22], [23]: as an example, Fowler [24] defined the *Refused Bequest* and *Middle Man* code smells, which refer to the poor use of inheritance and delegation in Object-Oriented programs that might lead to deteriorate their code quality [22], [25], [26], [27]. These studies have also led to the definition of automated code smell detection and refactoring approaches [28], [29], [30].

Still from an empirical standpoint, a number of studies targeted the role of inheritance and delegation mechanisms for monitoring software quality. In particular, researchers devoted extensive effort on the understanding of the potential impact of those mechanisms on software metrics [31], [32], [33], maintainability effort and costs [34], [35], [36], [37], design patterns [38], [39], change-proneness [40], [41], [42], [43], and source code defectiveness [44], [45], [46], [47].

While the current body of knowledge provides compelling evidence of the value of reusability mechanisms for the analysis of source code quality properties, we can still identify a noticeable research gap: as Mens and Demeyer [48] already reported in the early 2000s, the *long-term* evolution of source code quality metrics might provide a different perspective of the nature of a software project, possibly revealing complementary or even contrasting findings with respect to the studies that investigated code metrics in a fixed point of software evolution. To the best of our knowledge, Nasseri *et al.* [49] were the only researchers studying the evolution of reusability metrics. They specifically focused on the size of the inheritance hierarchies and aimed at assessing whether

Software analysis, evolution, and Reengineering (SANER)

Delegation and Inheritance do not influence the defect proneness of source code

The reusability metrics positively influence the decrease of maintenance effort

On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

Giammaria Giordano · **Gerardo Festa** · **Gemma Catolino** · **Fabio Palomba** · **Filomena Ferrucci** · **Carmine Gravino**

Received: date / Accepted: date

Abstract Software reusability mechanisms, like inheritance and delegation in Object-Oriented programming, are widely recognized as key instruments of software design that reduce the risks of source code being affected by defects, other than to reduce the effort required to maintain and evolve source code. Previous work has traditionally employed source code reuse metrics for prediction purposes, e.g., in the context of defect prediction. However, our research identifies two noticeable limitations of the current literature. First, still little is known about the extent to which developers actually employ code reuse mechanisms over time. Second, it is still unclear how these mechanisms may contribute to explaining defect-proneness and maintenance effort during software evolution. We aim at bridging this gap of knowledge, as an improved understanding of these aspects might provide insights into the actual support provided by these mechanisms, e.g., by suggesting whether and how to use them for prediction purposes. We propose an exploratory study, conducted on 12 JAVA projects—over 9000 commits—of the DEFECTS4J dataset, aiming at (1) assessing how developers use inheritance and delegation during software evolution; and (2) statistically analyzing the impact of inheritance and delegation on fault proneness and maintenance effort. Our results let emerge various usage patterns that describe the way inheritance and delegation vary over time. In addition, we find out that inheritance and delegation are statistically significant factors that influence both source code defect-proneness and maintenance effort.

Keywords Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.

Giammaria Giordano, Gerardo Festa, Fabio Palomba, Filomena Ferrucci, Carmine Gravino
Software Engineering (SeSa) Lab - University of Salerno (Italy) — E-mail: giagiordano@unisa.it, g.festa22@studenti.unisa.it, fpalomba@unisa.it, fferucci@unisa.it

Gemma Catolino
Jheronimus Academy of Data Science & Tilburg University, The Netherlands — E-mail: g.catolino@tilburguniversity.edu

Under Review-Empirical Software Engineering (EMSE)

Different tools can detect different security-related concerns with different frequencies

There are security-related concerns never detected (e.g., Improper Access Control)

A Preliminary Conceptualization and Analysis on Automated Static Analysis Tools for Vulnerability Detection in Android Apps

Giammaria Giordano, Fabio Palomba, Filomena Ferrucci
Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy
giagiordano@unisa.it, fpalomba@unisa.it, fferrucci@unisa.it

***Abstract*—The availability of dependable mobile apps is a crucial need for over three billion people who use apps daily for any social and emergency connectivity. A key challenge for mobile developers concerns the detection of security-related issues. While a number of tools have been proposed over the years—especially for the ANDROID operating system—we point out a lack of empirical investigations on the actual support provided by these tools; these might guide developers in selecting the most appropriate instruments to improve their apps. In this paper, we propose a preliminary conceptualization of the vulnerabilities detected by three automated static analysis tools such as ANDROBUGS2, TRUESEEING, and INSIDER. We first derive a taxonomy of the issues detectable by the tools. Then, we run the tools against a dataset composed of 6,500 ANDROID apps to investigate their detection capabilities in terms of frequency of detection of vulnerabilities and complementarity among tools. Key findings of the study show that current tools identify similar concerns, but they use different naming conventions. Perhaps more importantly, the tools only partially cover the most common vulnerabilities classified by the Open Web Application Security Project (OWASP) Foundation.**

***Index Terms*—Software Vulnerabilities; Android Apps; Automated Static Analysis Tools; Empirical Software Engineering.**

I. INTRODUCTION

The last decades and, most notably, the recent years have seen a drastic change in the way people communicate and interact among them. Around 80% of the global population owns a smartphone [2] and about 70% of these smartphones rely on the ANDROID operating system [6]. The diffusion of this operating system (OS) is favored by multiple factors, including the availability and marketing of mobile apps through the online app store [26]. In this context, previous research has pointed out that ANDROID apps can be affected by severe vulnerabilities that can impact both user privacy and security [16], [28], [34]. For this reason, several automated static analysis tools have been proposed to detect security concerns and assist mobile developers in improving their applications [21]. Nevertheless, in our research, we observed a lack of knowledge about the real support provided by these tools. In particular, it is unclear the set of problems that these tools can detect and how they behave when detecting vulnerabilities, e.g., whether their analysis fails in certain cases, the most common vulnerabilities identified, and to what extent different tools cover different vulnerabilities. An improved understanding of these aspects is crucial to let developers be aware of what kind of problems can be currently detected, other than letting them (1) more wisely select the tools to employ, (2) evaluate on complementing more tools, or (3) even understand whether current tools can actually identify vulnerabilities that are becoming more and more popular and harmful nowadays.

This paper performs the first step toward enlarging the body of empirical knowledge on the matter. We focus on three automated static analysis tools, i.e., ANDROBUGS2,[1] TRUESEEING,[2] and INSIDER,[3] to elicit a taxonomy of security-related concerns detectable with these tools. Afterward, we execute the tools on 6,500 free apps to assess the number of vulnerabilities the tools can detect and the complementarity among them.

The main results of the study indicate that in most cases the tools can detect the same vulnerabilities but using a different vocabulary, causing possible misunderstandings. The tools are also complementary, which implies that developers should select tools based on the specific categories of vulnerabilities they would detect. Lastly, the considered tools only partially cover the most widespread vulnerabilities classified by the Open Web Application Security Project (OWASP) Foundation.

To sum up, our paper provides the following contributions:

1) A large-scale empirical investigation into the support provided by three state-of-the-practice tools for the detection of security-related concerns;
2) An empirical analysis of the complementarity among the three considered tools, which might open new research directions connected to their combination;
3) A publicly available replication package [7], which contains all data and scripts employed to address and extend our research questions.

II. RELATED WORK

Researchers have been focusing on the ANDROID platform, as its open-source nature eased the definition of empirical investigations on the matter [14], [15]. For similar reasons, our study revolves around ANDROID; in the remainder of the section, we discuss the literature connected to that.

[1] https://github.com/androbugs2/androbugs2
[2] https://github.com/alterakey/trueseeing
[3] https://github.com/insidersec/insider

Euromicro

Research Gaps

There is a **lack of empirical knowledge** on how **reusability** mechanisms **vary** in **complex systems**

Most studies on the impact of reusability on **code quality** and **reliability** focused only on **traditional systems**

Research Gaps

There is a **lack of empirical knowledge** on how **reusability** mechanisms **vary** in **complex systems**

On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

Giammaria Giordano · Gerardo Festa · Gemma Catolino · Fabio Palomba · Filomena Ferrucci · Carmine Gravino

Received: date / Accepted: date

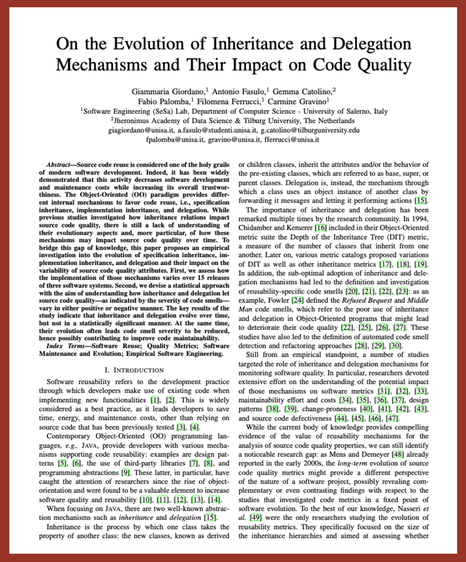
Abstract Software reusability mechanisms, like inheritance and delegation in Object-Oriented programming, are widely recognized as key instruments of software design that reduce the risks of source code being affected by defects, other than to reduce the effort required to maintain and evolve source code. Previous work has traditionally employed source code reuse metrics for prediction purposes, e.g., in the context of defect prediction. However, our research identifies two noticeable limitations of the current literature. First, still little is known about the extent to which developers actually employ code reuse mechanisms over time. Second, it is still unclear how these mechanisms may contribute to explaining defect-proneness and maintenance effort during software evolution. We aim at bridging this gap of knowledge, as an improved understanding of these aspects might provide insights into the actual support provided by these mechanisms, e.g., by suggesting whether and how to use them for prediction purposes. We propose an exploratory study, conducted on 12 JAVA projects—over 9000 commits—of the DEFECTS4J dataset, aiming at (1) assessing how developers use inheritance and delegation during software evolution; and (2) statistically analyzing the impact of inheritance and delegation on fault proneness and maintenance effort. Our results let emerge various usage patterns that describe the way inheritance and delegation vary over time. In addition, we find out that inheritance and delegation are statistically significant factors that influence both source code defect-proneness and maintenance effort.

Keywords Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.

Giammaria Giordano, Gerardo Festa, Fabio Palomba, Filomena Ferrucci, Carmine Gravino Software Engineering (SeSa) Lab - University of Salerno (Italy) — E-mail: giagiordano@unisa.it, g.festa229@studenti.unisa.it, fpalomba@unisa.it, fferucci@unisa.it

Gemma Catolino Jheronimus Academy of Data Science & Tilburg University, The Netherlands — E-mail: g.catolino@tilburguniversity.edu

EMSE



On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Giammaria Giordano,[1] Antonio Fasulo,[1] Gemma Catolino,[2] Fabio Palomba,[1] Filomena Ferrucci,[1] Carmine Gravino[1]

[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
giagiordano@unisa.it, a.fasulo@studenti.unisa.it, g.catolino@tilburguniversity.edu
fpalomba@unisa.it, gravino@unisa.it, fferucci@unisa.it

***Abstract*—Source code reuse is considered one of the holy grails of modern software development. Indeed, it has been widely demonstrated that this activity decreases software development and maintenance costs while increasing its overall trustworthiness. The Object-Oriented (OO) paradigm provides different internal mechanisms to favor code reuse, i.e., specification inheritance, implementation inheritance, and delegation. While previous studies investigated how inheritance relations impact source code quality, there is still a lack of understanding of their evolutionary aspects and, more particular, of how these mechanisms may impact source code quality over time. To bridge this gap of knowledge, this paper proposes an empirical investigation into the evolution of specification inheritance, implementation inheritance, and delegation and their impact on the variability of source code quality attributes. First, we assess how the implementation of those mechanisms varies over 15 releases of three software systems. Second, we devise a statistical approach with the aim of understanding how inheritance and delegation let source code quality—as indicated by the severity of code smells—vary in either positive or negative manner. The key results of the study indicate that inheritance and delegation evolve over time, but not in a statistically significant manner. At the same time, their evolution often leads code smell severity to be reduced, hence possibly contributing to improve code maintainability.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.**

I. Introduction

Software reusability refers to the development practice through which developers make use of existing code when implementing new functionalities [1], [2]. This is widely considered as a best practice, as it leads developers to save time, energy, and maintenance costs, other than relying on source code that has been previously tested [3], [4].

Contemporary Object-Oriented (OO) programming languages, *e.g.*, JAVA, provide developers with various mechanisms supporting code reusability: examples are design patterns [5], [6], the use of third-party libraries [7], [8], and programming abstractions [9]. These latter, in particular, have caught the attention of researchers since the rise of object-orientation and were found to be a valuable element to increase software quality and reusability [10], [11], [12], [13], [14].

When focusing on JAVA, there are two well-known abstraction mechanisms such as *inheritance* and *delegation* [15].

Inheritance is the process by which one class takes the property of another class: the new classes, known as derived or children classes, inherit the attributes and/or the behavior of the pre-existing classes, which are referred to as base, super, or parent classes. Delegation is, instead, the mechanism through which a class uses an object instance of another class by forwarding it messages and letting it performing actions [15].

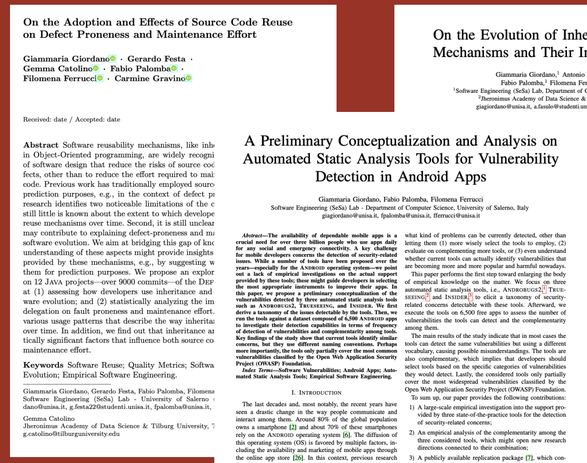
The importance of inheritance and delegation has been remarked multiple times by the research community. In 1994, Chidamber and Kemerer [16] included in their Object-Oriented metric suite the Depth of the Inheritance Tree (DIT) metric, a measure of the number of classes that inherit from one another. Later on, various metric catalogs proposed variations of DIT as well as other inheritance metrics [17], [18], [19]. In addition, the sub-optimal adoption of inheritance and delegation mechanisms had led to the definition and investigation of reusability-specific code smells [20], [21], [22], [23]: as an example, Fowler [24] defined the *Refused Bequest* and *Middle Man* code smells, which refer to the poor use of inheritance and delegation in Object-Oriented programs that might lead to deteriorate their code quality [22], [25], [26], [27]. These studies have also led to the definition of automated code smell detection and refactoring approaches [28], [29], [30].

Still from an empirical standpoint, a number of studies targeted the role of inheritance and delegation mechanisms for monitoring software quality. In particular, researchers devoted extensive effort on the understanding of the potential impact of those mechanisms on software metrics [31], [32], [33], maintainability effort and costs [34], [35], [36], [37], design patterns [38], [39], change-proneness [40], [41], [42], [43], and source code defectiveness [44], [45], [46], [47].

While the current body of knowledge provides compelling evidence of the value of reusability mechanisms for the analysis of source code quality properties, we can still identify a noticeable research gap: as Mens and Demeyer [48] already reported in the early 2000s, the *long-term* evolution of source code quality metrics might provide a different perspective of the nature of a software project, possibly revealing complementary or even contrasting findings with respect to the studies that investigated code metrics in a fixed point of software evolution. To the best of our knowledge, Nasseri *et al.* [49] were the only researchers studying the evolution of reusability metrics. They specifically focused on the size of the inheritance hierarchies and aimed at assessing whether

SANER

Most studies on the impact of reusability on **code quality** and **reliability** focused only on **traditional systems**



On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

Giammaria Giordano · Gerardo Festa · Gemma Catolino · Fabio Palomba · Filomena Ferrucci · Carmine Gravino

Received: date / Accepted: date

Abstract Software reusability mechanisms, like inh in Object-Oriented programming, are widely recogni of software design that reduce the risks of source cod fects, other than to reduce the effort required to mai code. Previous work has traditionally employed sourc prediction purposes, e.g., in the context of defect p research identifies two noticeable limitations of the c still little is known about the extent to which develope reuse mechanisms over time. Second, it is still unclear may contribute to explaining defect-proneness and m software evolution. We aim at bridging this gap of kn understanding of these aspects might provide insights provided by these mechanisms, e.g., by suggesting w them for prediction purposes. We propose an explor on 12 JAVA projects—over 9000 commits—of the DEF at (1) assessing how developers use inheritance and ware evolution; and (2) statistically analyzing the im delegation on fault proneness and maintenance effort. various usage patterns that describe the way inheritan over time. In addition, we find out that inheritance a tically significant factors that influence both source co maintenance effort.

Keywords Software Reuse; Quality Metrics; Softwa Evolution; Empirical Software Engineering.

Giammaria Giordano, Gerardo Festa, Fabio Palomba, Filomena Software Engineering (SeSa) Lab - University of Salerno dano@unisa.it, g.festa229@studenti.unisa.it, fpalomba@unisa.it,

Gemma Catolino Jheronimus Academy of Data Science & Tilburg University, g.catolino@tilburguniversity.edu

EMSE

On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Giammaria Giordano,[1] Antonio Fasulo,[1] Gemma Catolino,[2] Fabio Palomba,[1] Filomena Ferrucci,[1] Carmine Gravino[1]

[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
giagiordano@unisa.it, a.fasulo@studenti.unisa.it, g.catolino@tilburguniversity.edu
a.it, fferucci@unisa.it

A Preliminary Conceptualization and Analysis on Automated Static Analysis Tools for Vulnerability Detection in Android Apps

Giammaria Giordano, Fabio Palomba, Filomena Ferrucci
Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy
giagiordano@unisa.it, fpalomba@unisa.it, fferrucci@unisa.it

***Abstract*—The availability of dependable mobile apps is a crucial need for over three billion people who use apps daily for any social and emergency connectivity. A key challenge for mobile developers concerns the detection of security-related issues. While a number of tools have been proposed over the years—especially for the ANDROID operating system—we point out a lack of empirical investigations on the actual support provided by these tools; these might guide developers in selecting the most appropriate instruments to improve their apps. In this paper, we propose a preliminary conceptualization of the vulnerability detection capabilities of three popular tools available for ANDROID, i.e., ANDROBUGS, TRUESEEING, and INSIDER. We first derive a taxonomy of issues detectable by the tools. Then, we run the tools against a dataset composed of 6,500 ANDROID apps to investigate their detection capabilities in terms of frequency of detection of vulnerabilities and complementarity among tools. Key findings of the study show that current tools identify similar concerns, but they use different naming conventions. Perhaps more importantly, the tools only partially cover the most common vulnerabilities classified by the Open Web Application Security Project (OWASP) Foundation.**

***Index Terms*—Software Vulnerabilities; Android Apps; Automated Static Analysis Tools; Empirical Software Engineering.**

I. Introduction

The last decades and, most notably, the recent years have seen a drastic change in the way people communicate and interact among them. Around 80% of the global population owns a smartphone [2] and about 70% of these smartphones rely on the ANDROID operating system [6]. The diffusion of this operating system (OS) is favored by multiple factors, including the availability and marketing of mobile apps through the online app store [26]. In this context, previous research has pointed out that ANDROID apps can be affected by severe vulnerabilities that can impact both user privacy and security [16], [28], [34]. For this reason, several automated static analysis tools have been proposed to detect security concerns and assist mobile developers in improving their applications [21]. Nevertheless, in our research, we observed a lack of knowledge about the real support provided by these tools. In particular, it is unclear the set of problems that these tools can detect and how they behave when detecting vulnerabilities, e.g., whether their analysis fails in certain cases, the most common vulnerabilities identified, and to what extent different tools cover different vulnerabilities. An improved understanding of these aspects is crucial to let developers be aware of what kind of problems can be currently detected, other than letting them (1) more wisely select the tools to employ, (2) evaluate on complementing more tools, or (3) even understand whether current tools can actually identify vulnerabilities that are becoming more and more popular and harmful nowadays.

This paper performs the first step toward enlarging the body of empirical knowledge on the matter. We focus on three automated static analysis tools, i.e., ANDROBUGS,[1] TRUESEEING,[2] and INSIDER,[3] to elicit a taxonomy of security-related concerns detectable with these tools. Afterward, we execute the tools on 6,500 free apps to assess the number of vulnerabilities the tools can detect and the complementarity among them.

The main results of the study indicate that in most cases the tools can detect the same vulnerabilities but using a different vocabulary, causing possible misunderstandings. The tools are also complementary, which implies that developers should select tools based on the specific categories of vulnerabilities they would detect. Lastly, the considered tools only partially cover the most widespread vulnerabilities classified by the Open Web Application Security Project (OWASP) Foundation.

To sum up, our paper provides the following contributions:

1) A large-scale empirical investigation into the support provided by three state-of-the-practice tools for the detection of security-related concerns;
2) An empirical analysis of the complementarity among the three considered tools, which might open new research directions connected to their combination;
3) A publicly available replication package [7], which contains all data and scripts employed to address and extend our research questions.

II. Related Work

Researchers have been focusing on the ANDROID platform, as its open-source nature eased the definition of empirical investigations on the matter [14], [15]. For similar reasons, our study revolves around ANDROID; in the remainder of the section, we discuss the literature connected to that.

[1]https://github.com/androbugs2/androbugs2
[2]https://github.com/aboutsecurity/seeing
[3]https://github.com/insidersec/insider

hildren classes, inherit the attributes and/or the behavior of pre-existing classes, which are referred to as base, super, or nt classes. Delegation is, instead, the mechanism through ch a class uses an object instance of another class by varding it messages and letting it performing actions [15]. he importance of inheritance and delegation has been arked multiple times by the research community. In 1994, damber and Kemerer [16] included in their Object-Oriented ric suite the Depth of the Inheritance Tree (DIT) metric, sasure of the number of classes that inherit from one ther. Later on, various metric catalogs proposed variations DIT as well as other inheritance metrics [17], [18], [19]. ddition, the sub-optimal adoption of inheritance and dele- on mechanisms had led to the definition and investigation reusability-specific code smells [20], [21], [22], [23]: as an mple, Fowler [24] defined the *Refused Bequest* and *Middle* n code smells, which refer to the poor use of inheritance delegation in Object-Oriented programs that might lead leteriorate their code quality [22], [25], [26], [27]. These ies have also led to the definition of automated code smell ction and refactoring approaches [28], [29], [30].

till from an empirical standpoint, a number of studies ted the role of inheritance and delegation mechanisms for storing software quality. In particular, researchers devoted nsive effort on the understanding of the potential impact hose mechanisms on software metrics [31], [32], [33], stainability effort and costs [34], [35], [36], [37], design erns [38], [39], change-proneness [40], [41], [42], [43], source code defectiveness [44], [45], [46], [47].

hile the current body of knowledge provides compelling lence of the value of reusability mechanisms for the lysis of source code quality properties, we can still identify ticeable research gap: as Mens and Demeyer [48] already rted in the early 2000s, the *long-term* evolution of source e quality metrics might provide a different perspective he nature of a software project, possibly revealing com- nentary or even contrasting findings with respect to the ies that investigated code metrics in a fixed point of ware evolution. To the best of our knowledge, Nasseri *et* [49] were the only researchers studying the evolution of ability metrics. They specifically focused on the size of inheritance hierarchies and aimed at assessing whether

SANER

Euromicro

Closed works

On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality
Saner

A Preliminary Conceptualization and Analysis on Automated Static Analysis Tools for Vulnerability Detection in Android Apps
Euromicro

On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort
Submitted at EMSE

Open works

Survey with developers on the role of reusability mechanisms

Other works

On the Use of Artificial Intelligence to Deal with Privacy in IoT Systems: A Systematic Literature Review
Journal of Systems and Software (JSS)
supported with Business Solution S.L.R.

Next Steps

Survey with developers on the role of reusability mechanisms

February

An empirical evaluation of reusability mechanisms in complex systems

April

On the role of reusability mechanisms and their impact on code quality and reliability

May

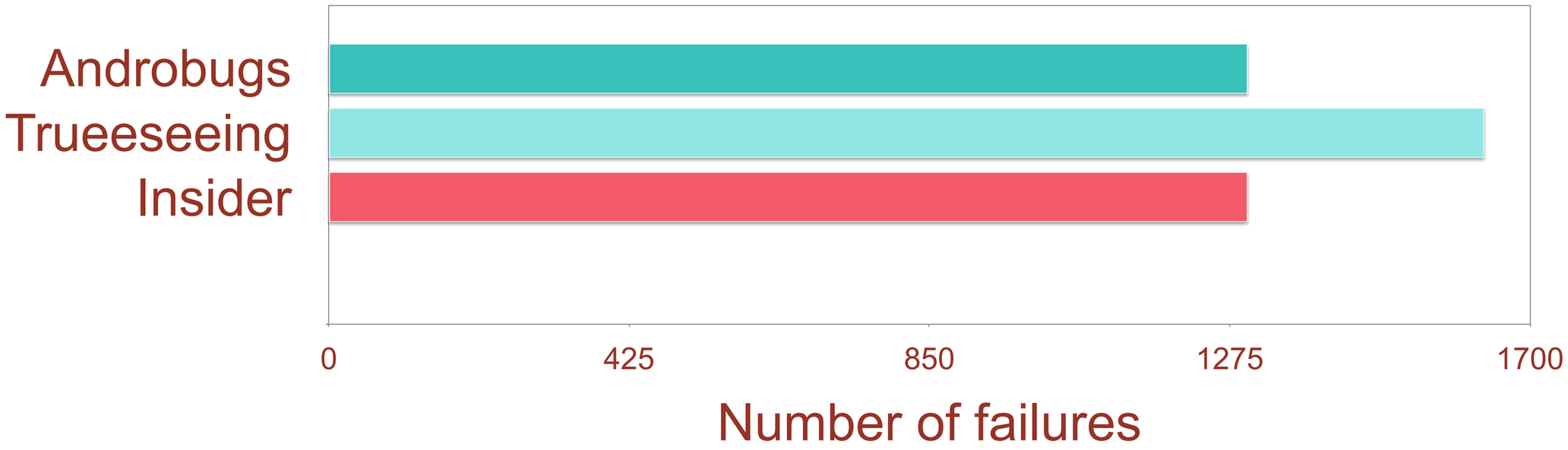
Backup slides

Third-Party Libraries

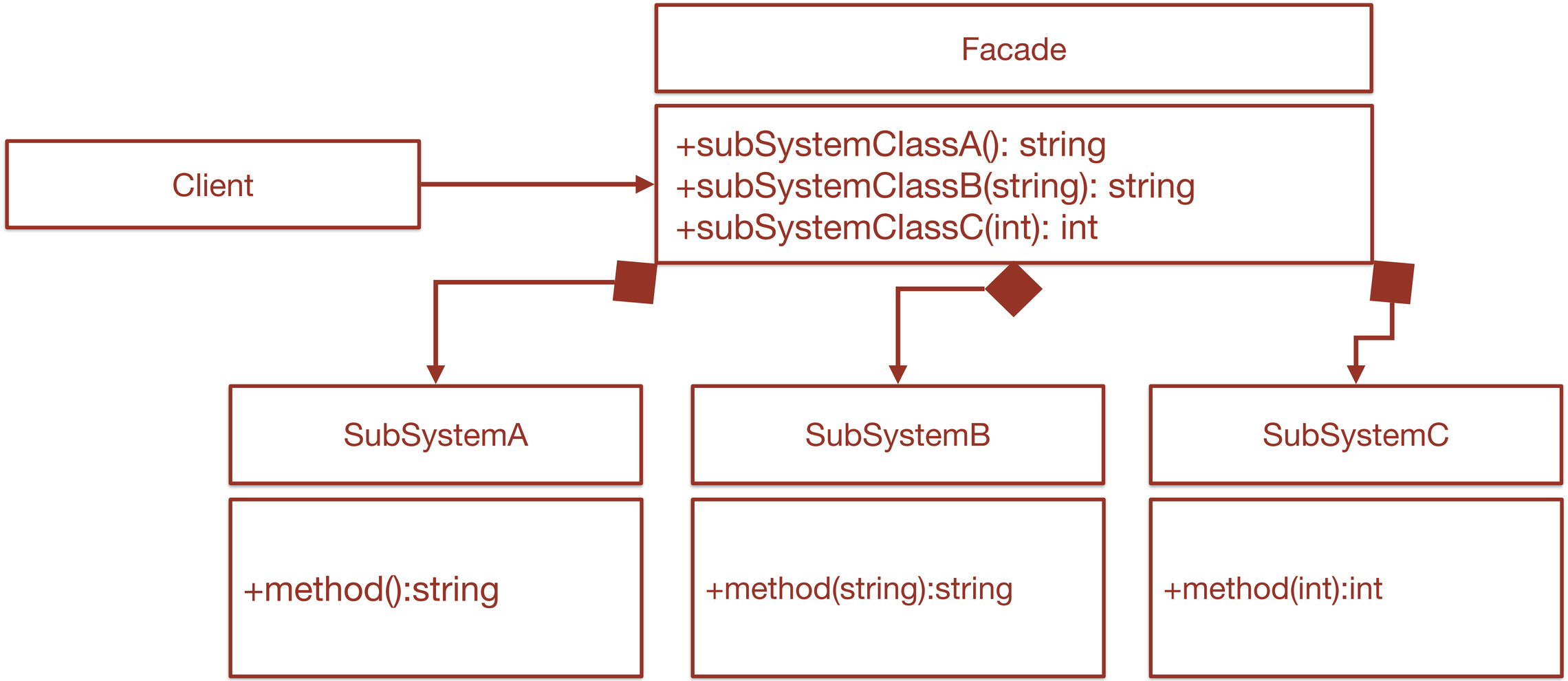
```
from qiskit_machine_learning.algorithms import QSVC


 sqvc = QSVC(quantum_kernel = adhoc_kernel)
 qsvc.fot(train_features, train_labels)
 qsvc_score = qsvc.score(test_features, test_labels)
```

RQ2 - How static analysis tools can be used to detect reliability issues in mobile apps?

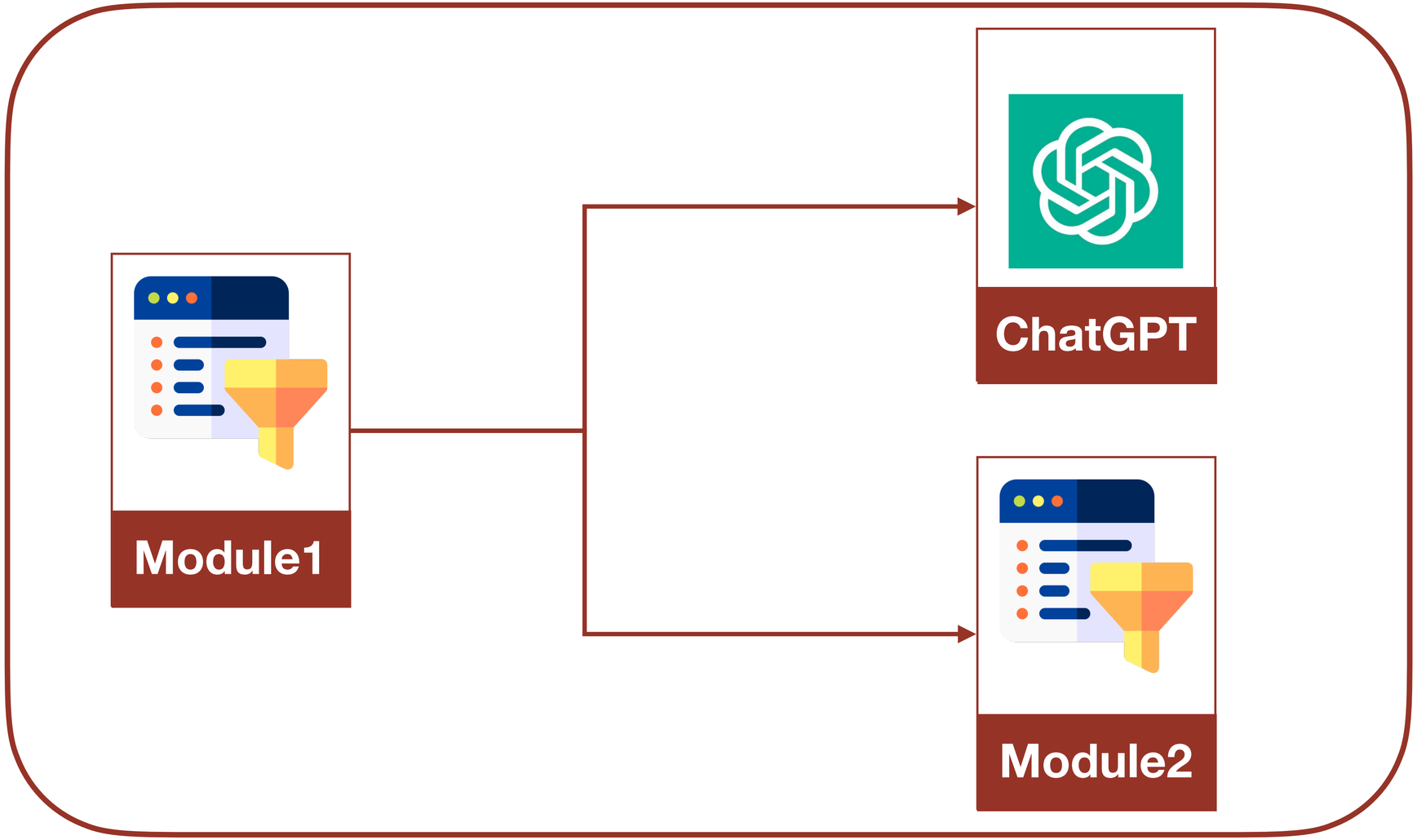
The preliminary results indicated that in most cases different tools can detect different vulnerabilities with different frequencies.



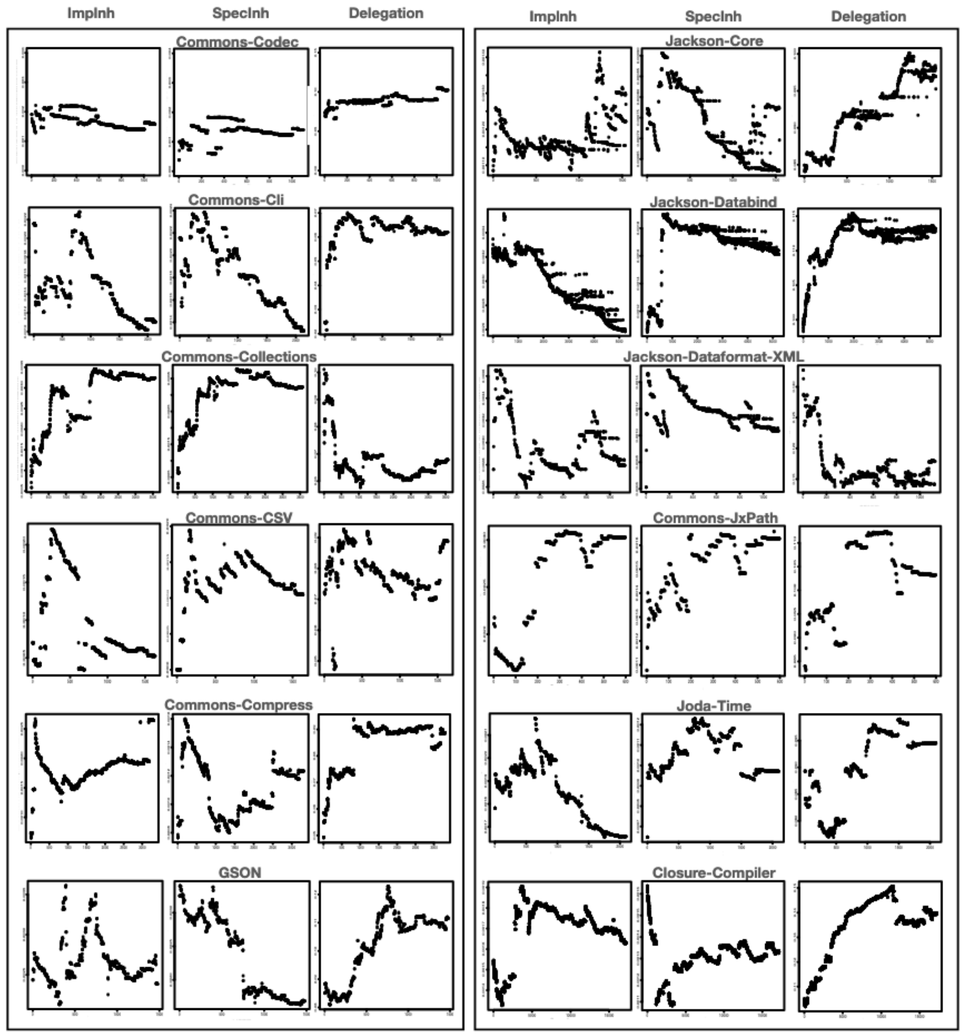
Design Pattern



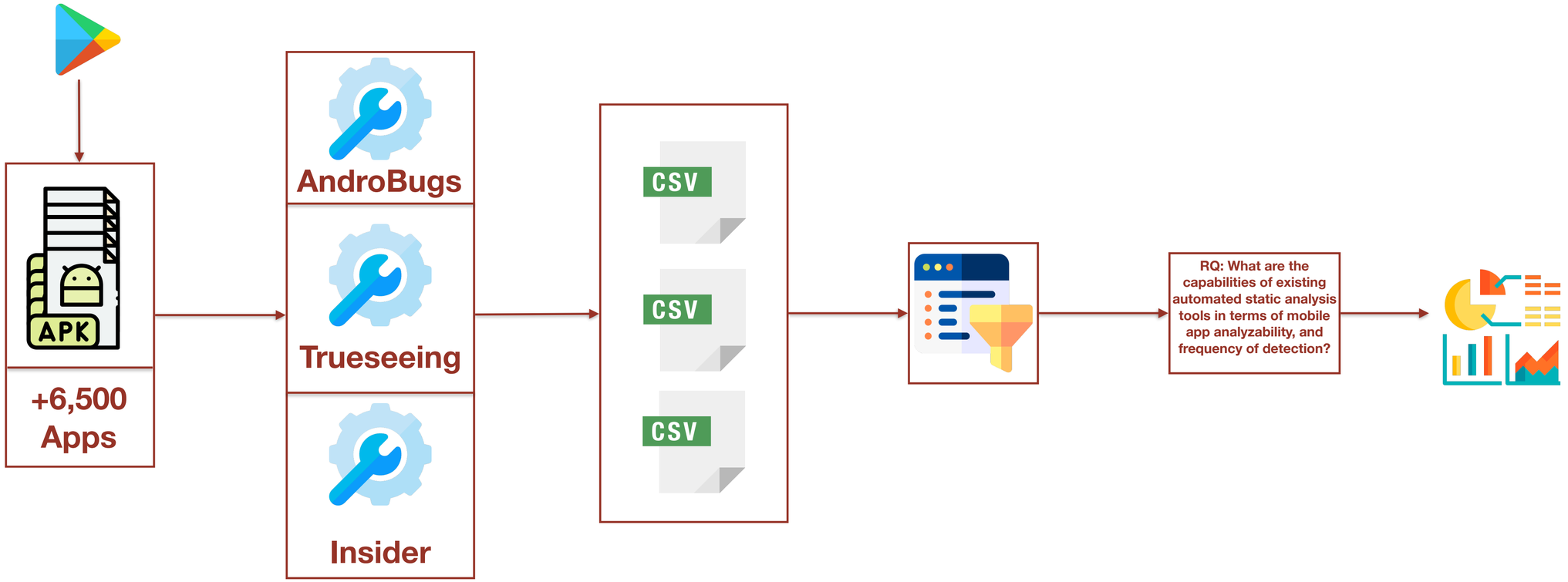
System Wrapping



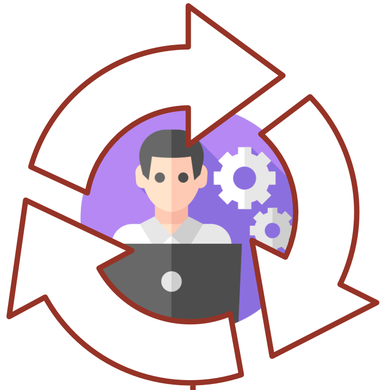
RQ1 - How


ImpInh
SpecInh
Delegation
Commons-Codec
Commons-Cli
Commons-Collections
Commons-CSV
Commons-Compress
GSON
ImpInh
SpecInh
Delegation
Jackson-Core
Jackson-Databind
Jackson-Dataformat-XML
Commons-JxPath
Joda-Time
Closure-Compiler

RQ2 - An empirical analysis of the impact of vulnerabilities in mobile apps



RQ2 - On the impact of reusability mechanisms in traditional systems



- Code Quality
  - Code Smells
- Reliability
  - Defect-Proneness
  - Maintenance Effort

RQ2 - On the impact of reusability mechanisms in traditional systems

To address the **RQ2** from a **defect proneness and maintenance effort** standpoint, we selected over 9,000 commits of 12 Java projects provided by Defects4J and extracted information on commits using PyDriller

Dependent Variables
#Bugs
Code Churns

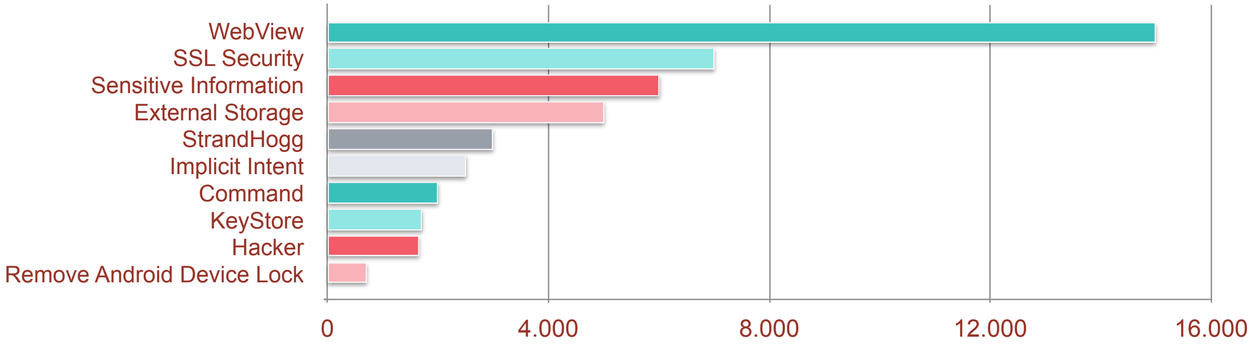
Independent Variables
Inheritance
Delegation

Control Variables
CK Metrics

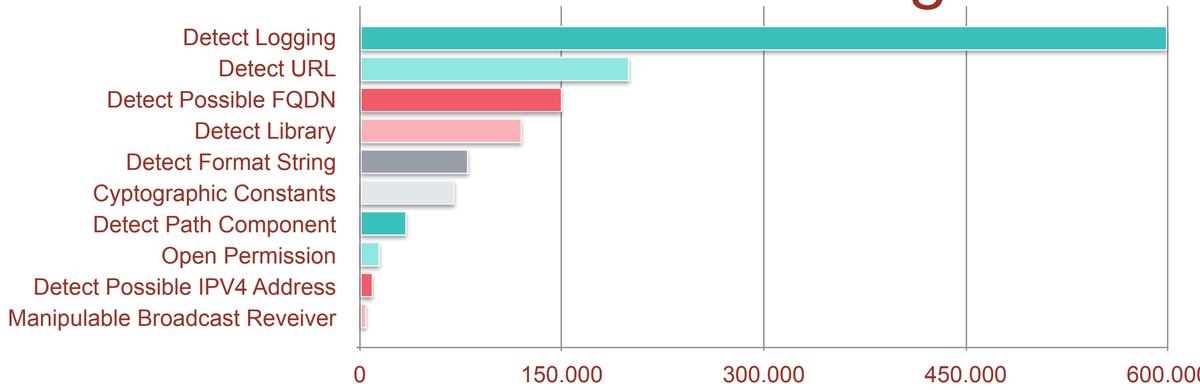
To assess our study, we used the **Multinomial log-linear model,** and the **generalized Linear Model**

RQ2 - How static analysis tools can be used to detect reliability issues in mobile apps?

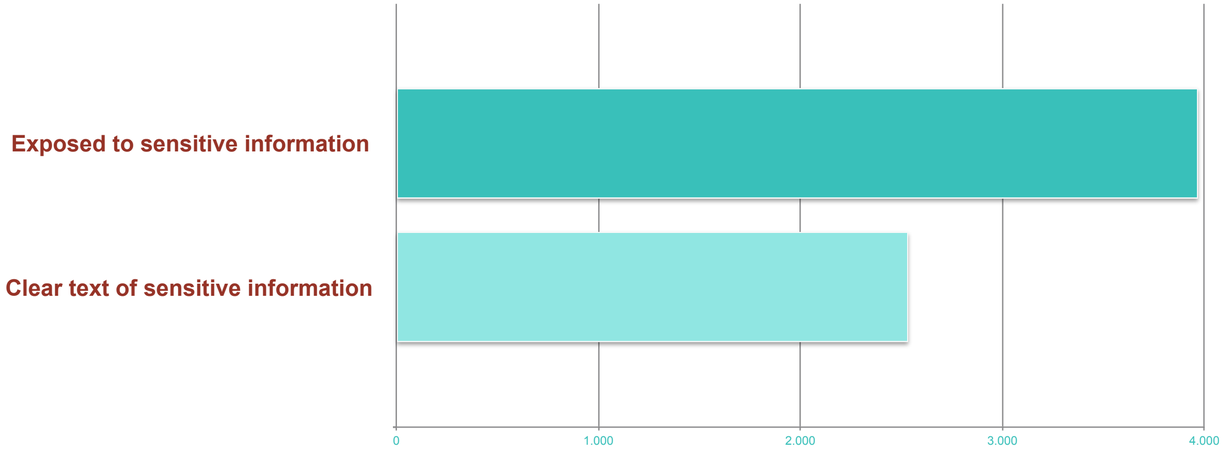
Androbugs



Trueeseeing



Insider



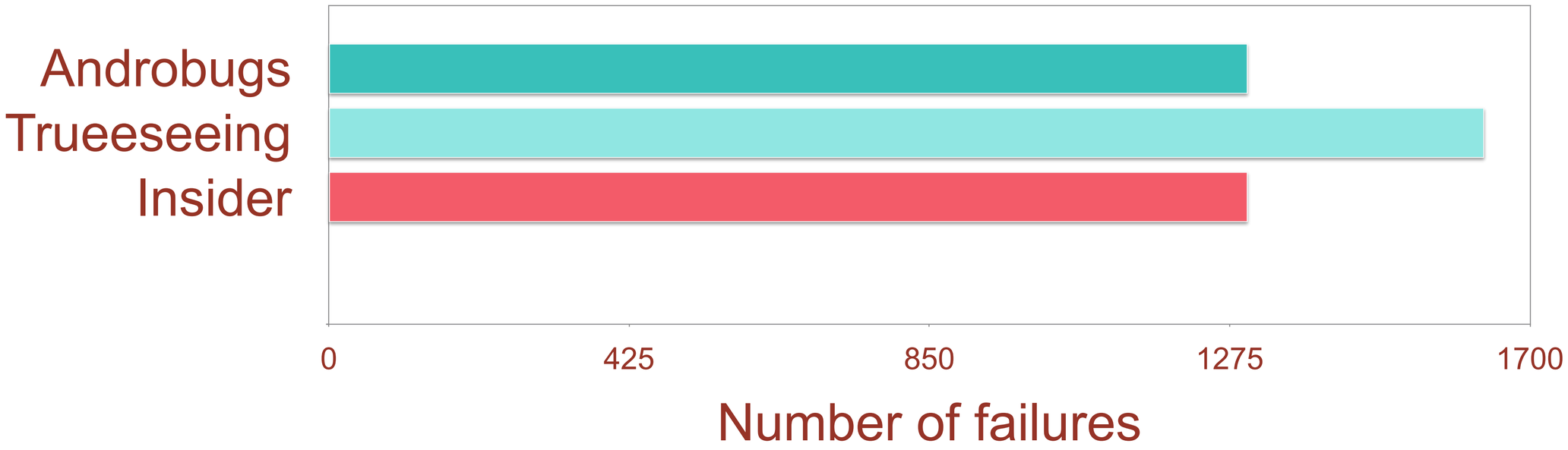
Different tools can detect different security-related concerns with different frequencies

There are security-related concerns never detected (e.g., Improper Access Control)

A deeper analysis of the actual support provided by these tools could be necessary

RQ2 - On the role of Static Analyses Tools to detect Vulnerabilities in Android Applications

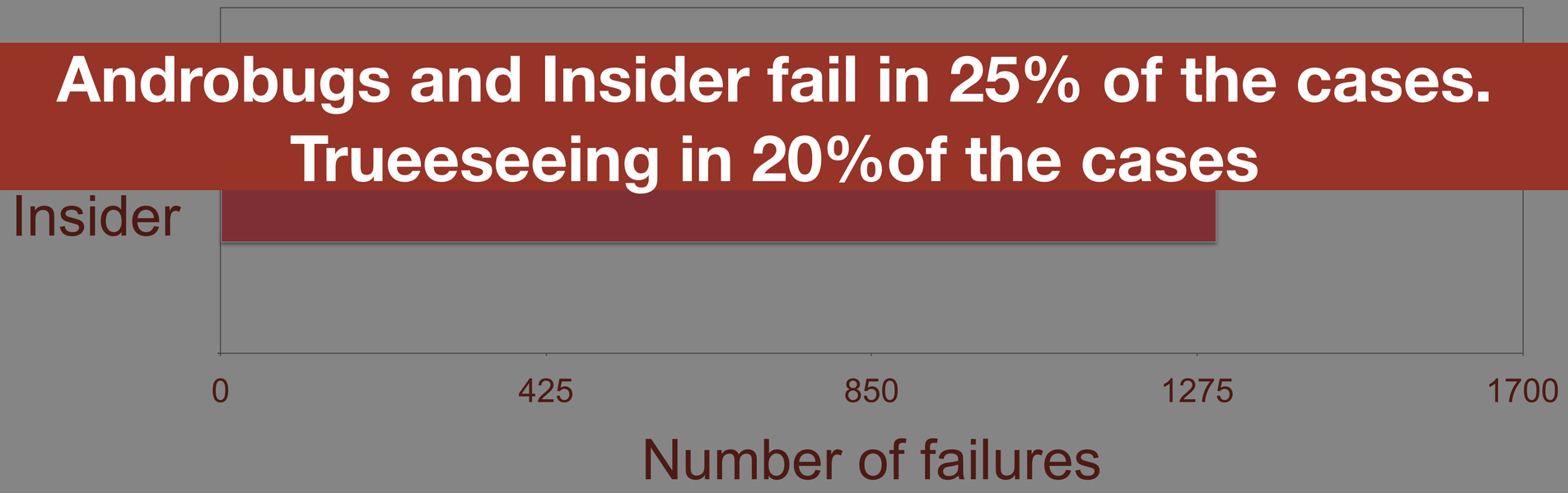
The preliminary results indicated that in most cases different tools can detect different vulnerabilities with different frequencies.



RQ2 - On the role of Static Analyses Tools to detect Vulnerabilities in Android Applications

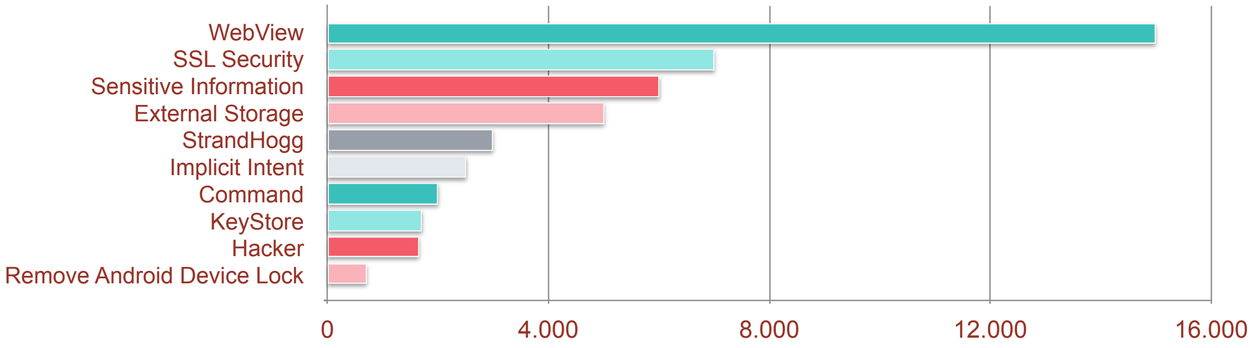
The preliminary results indicated that in most cases different tools can detect different vulnerabilities with different frequencies.

Androbugs and Insider fail in 25% of the cases.
Trueeseeing in 20%of the cases

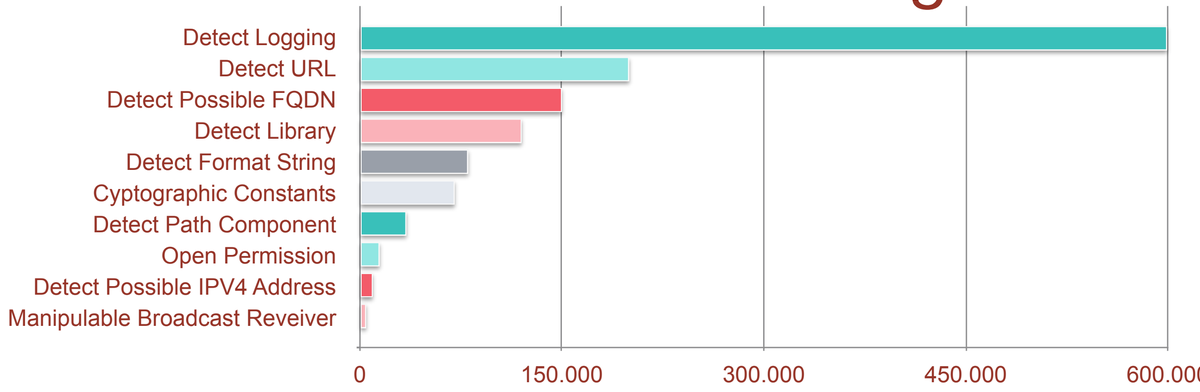


RQ2 - On the role of Static Analyses Tools to detect Vulnerabilities in Android Applications

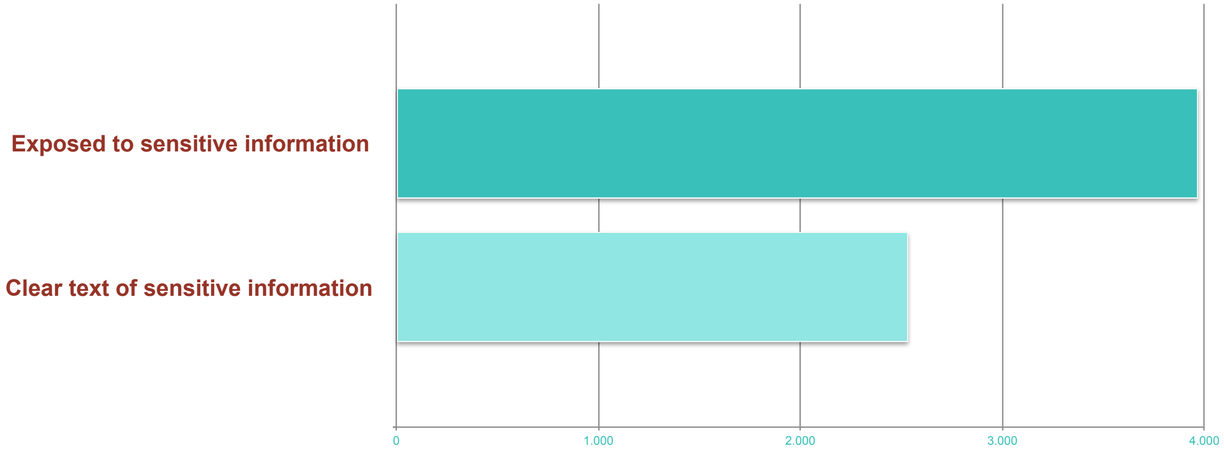
Androbugs



Trueeseeing



Insider



RQ2 - On the impact of reusability mechanisms in traditional systems

	Spaghetti Code		God Class		Class Data Should Be Private		Complex Class	
Variables	**Decrease**	**Increase**	**Decrease**	**Increase**	**Decrease**	**Increase**	**Decrease**	**Increase**
Delegation	0.011***	-0.358***	1.054***	-1.363***	0.016***	0.330***	0.011***	-0.358***
Implementation Inheritance	-0.094***	-0.061***	0.048***	0.003***	-0.016***	-0.008***	-0.094***	-0.061***
Specification Inheritance	0.002***	-0.023***	0.028***	0.036***	0.022***	-0.010***	0.002***	-0.023***
DIT	0.060***	0.180***	-0.274***	0.108***	-0.133***	0.004***	0.060***	0.180***
NOC	-0.009***	0.007***	-0.053***	0.002***	0.032***	0.004***	-0.009***	0.007***
LOC	-0.000	-0.000	-0.000	-0.000	0.000**	-0.000	-0.000	-0.000
LCOM	0.910***	-0.101***	-0.177***	-3.218***	0.408***	0.230***	0.910***	-0.101***
WMC	0.0005	-0.0004	-0.002	-0.001	0.001	-0.001	0.0005	-0.0004
CBO	-0.327***	0.201***	-0.679***	0.870***	0.023***	-0.453***	-0.327***	0.201***
RFC	0.001	0.003**	0.008***	0.003	0.002	0.005***	0.001	0.003***

*p<0.1; **p<0.05; ***p<0.01.

RQ2 - On the impact of reusability mechanisms in traditional systems

	Spaghetti Code		God Class		Class Data Should Be Private		Complex Class	
Variables	**Decrease**	**Increase**	**Decrease**	**Increase**	**Decrease**	**Increase**	**Decrease**	**Increase**
Delegation	0.011***	-0.358***	1.054***	-1.363***	0.016***	0.330***	0.011***	-0.358***
Implementation Inheritance	-0.094***	-0.061***	0.048***	0.003***	-0.016***	-0.008***	-0.094***	-0.061***
Specification Inheritance	0.002***	-0.023***	0.028***	0.036***	0.022***	-0.010***	0.002***	-0.023***
DIT	0.060***	0.180***	-0.274***	0.108***	-0.133***	0.004***	0.060***	0.180***

Delegation and inheritance positively correlate to the decrease of the code smell severity

RQ2 - On the impact of reusability mechanisms in traditional systems

	Com.-Codec N=2,134	Com.-Cli N=1,099	Com.-Col. N=3,560	Com.-CSV N=1,634	Comp. N=3,305	Gson N=1,478
DiffWMC	163.951 (105.295)	26.263 (227.457)	-20.375 (56.965)	-20.375 (56.965)	-1,988.919*** (210.722)	-377.039 (269.203)
DiffNOC			10,213.080*** (2,341.143)	-132.074 (1,357.614)	-10,740.970*** (1,699.369)	17,827.570*** (3,288.230)
DiffLCOM	1.383 (2.713)	-12.154 (16.169)	7.799* (4.680)	10.673*** (1.905)	26.285*** (6.021)	-15.488 (12.955)
DiffDIT	3,341.228*** (903.732)	-2,378.489 (1,497.270)	-1,787.167 (1,192.199)		52,852.530*** (2,813.141)	-6,958.231** (2,826.673)
DiffCBO	1,021.357*** (150.161)	-108.063 (134.420)	6,717.225*** (652.191)	-56.282 (94.958)	5,529.115*** (307.003)	1,916.428*** (145.944)
DiffRFC		192.611*** (57.094)	4.682 (60.012)			
DiffLOC	1.293 (2.158)	-9.992* (5.254)	-58.840*** (10.760)	2.994 (2.693)	5.769 (5.471)	46.604*** (10.894)
Delegation	0.017 (0.045)	-0.697*** (0.229)	-0.003 (0.057)	0.165 (0.124)	-0.080*** (0.022)	-0.119** (0.050)
SpecInh	-1.217 (3.145)	39.595*** (11.915)	1.143 (1.211)	-6.889 (6.984)	-0.710 (1.950)	1.415 (1.301)
ImpInh	-0.131 (1.747)	-1.026 (0.870)	-0.386 (4.387)		3.653*** (1.093)	2.080 (2.226)
BugDecrease	1.433 (37.641)	-74.159 (102.978)	-106.139 (620.995)	-1.272 (69.685)	-26.208 (85.330)	-6.128 (90.985)
BugIncrease	-20.191 (37.620)	-70.799 (101.593)	-111.854 (620.996)	-14.892 (69.652)	-15.856 (86.311)	-14.031 (96.408)
Constant	52.948*** (15.918)	126.288** (51.413)	103.349 (77.155)	11.161 (20.003)	91.271*** (25.693)	111.100* (63.054)
	Jack.-Core N=1,543	**Jack.-Datab. N=5,228**	**Jack.-XML N=1,128**	**Com.-JxPath N=598**	**Joda-Time N=2,094**	**Clo.-Compiler N=17,171**
DiffWMC		853.627*** (71.347)		-889.089 (1,208.343)		-35,485.190 *** (1,024.124)
DiffNOC	21,588.520*** (1,212.595)	22,830.430*** (1,564.318)	333.786 (509.649)	24,786.920*** (5,760.288)	54,104.760*** (3,864.815)	204,776.100*** (25,377.820)
DiffLCOM	1.241 (0.765)	-28.862*** (1.208)	-8.269*** (0.992)	21.501*** (5.505)	189.720*** (23.536)	454.243*** (9.841)
DiffDIT		50,782.460*** (1,712.723)				305,846.900*** (27,225.780)
DiffCBO	1,682.687*** (131.899)	3,147.449*** (74.440)	239.229*** (19.892)	3,504.462*** (363.997)	-31,929.670*** (2,814.595)	-472.842 (643.145)
DiffRFC				1,358.532*** (363.735)		
DiffLOC	28.585*** (1.371)	-8.353*** (3.029)	8.568*** (0.946)	-158.255*** (12.875)	344.715*** (42.978)	1,028.922*** (38.580)
Delegation	-0.012 (0.017)	0.010* (0.006)	-0.096** (0.044)	-0.598** (0.294)	-0.580*** (0.107)	-0.014* (0.008)
SpecInh	2.701 (3.677)	-1.281*** (0.305)	-2.847 (2.324)	6.773 (14.133)	-156.445*** (22.026)	0.868 (0.737)
ImpInh		-0.140 (0.499)	3.001** (1.408)	0.942 (3.559)	179.745*** (20.294)	0.406 (0.407)
BugDecrease	83.885* (48.156)	28.236 (19.492)	19.831 (23.738)	-41.147 (149.800)	-625.949 (602.612)	-73.094 (91.940)
BugIncrease	-51.820 (48.181)	-7.043 (20.359)	-20.624 (26.005)	14.086 (149.626)	-690.321 (618.444)	-52.403 (91.954)
Constant	25.537 (47.505)	182.664*** (57.987)	57.931*** (9.413)	1,561.449*** (387.616)	-6,729.363*** (706.442)	89.824 (76.824)

Significance codes: *p<0.1; **p<0.05; ***p<0.01.

	Com.-Codec N=2,134		Com.-Cli N=1,099		Com.-Col. N=3,560		Com.-CSV N=1,634		Comp. N=3,305		Gson N=1,478	
	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
DiffWMC	-10.098 (7.495)	2.280 (10.981)	-0.691 (3.434)	2.416 (3.602)			-3.539 (2.559)	0.627 (5.136)	-5.248*** (0.033)	-1.903 (4.387)	3.261 (30.899)	-1.305 (25.907)
DiffNOC					-0.038 (0.050)	0.004 (0.013)	-4.413*** (0.156)	1.052*** (0.176)	10.188*** (0.002)	-5.653*** (0.051)	-0.159 (0.275)	1.536*** (0.415)
DiffLCOM	0.092 (0.140)	0.054 (0.261)	0.166 (0.256)	-0.744*** (0.244)	0.422 (0.808)	0.476 (22.615)	-0.056 (0.066)	-0.040 (0.130)	-0.066 (0.335)	0.046 (0.125)	0.013 (1.242)	-0.159 (1.157)
DiffDIT	11.927*** (0.269)	−0.183*** (0.033)	0.012 (5.830)	-0.0003 (0.023)	0.012 (5.830)	-0.0003 (0.023)	-4.526*** (0.238)	0.661*** (0.169)	12.511*** (0.002)	-5.151*** (0.125)	0.696 (0.466)	1.896** (0.798)
DiffCBO	-5.434 (5.898)	-9.729*** (0.243	-0.645 (3.617)	-5.947 (3.821)	-0.878 (58.069)	-0.495*** (0.103)	-0.994 (4.485)	-3.484 (8.728)	-4.163*** 0.021)	1.467 (2.717)	-17.977 (12.462)	-3.472 (12.553)
DiffRFC	4.123 (5.784)	-0.030 (1.106)	-0.014 (1.027)	4.123 (5.784)	1.013 (6.087)	1.924 (14.548)						
DiffLOC	0.005 (0.302)	0.075 (0.346)	0.002 (0.139)	0.056 (0.200)	-0.611 (1.053)	-0.099 (11.153)	0.175 0.121)	0.045 (0.236)	0.149* (0.090)	0.024 (0.104)	0.115 (1.273)	0.689 (1.373)
DiffDelegations	0.058 (0.049)	-0.060 (0.077	0.017 (0.022)	0.001 (0.025	-0.069 (0.137)	0.004 (0.654)	0.031 (0.076)	-0.058 (0.147)	0.013 (0.017)	-0.003 (0.013)	0.068 (0.059)	-0.018 (0.078)
DiffSpecInh	-1.791 (1.685)	-1.510 (3.395	0.070 (0.618)	1.382*** (0.542)	1.013 (6.087)	1.924 (14.548)	-0.571 (5.267)	-1.495 (10.178)	-0.187 (0.862)	-0.148 (0.637)	-0.356 (3.632)	0.226 (1.865)
DiffimpInh	-0.060 (0.940)	0.046 (2.134			0.767 (3.457)	0.771 (13.557)	-1.141 (2.923)	-0.094 (4.432)	-0.337 (0.488)	0.154 (0.383)	0.047 (2.105)	0.267 (1.713)
Churns	-0.002 (0.003)	-0.002 0.004)	-0.002 (0.002)	-0.005 (0.004)	-0.026 (0.038)	-0.100 (0.127)	-0.004 (0.007)	-0.009 (0.013)	-0.003 (0.002)	-0.0003 (0.001)	-0.015 (0.014)	-0.007 (0.009)
Constant	-4.762*** (0.248)	-4.717*** (0.246)	-3.472*** (0.187)	-3.448*** (0.187)	6.429*** (0.536)	-6.230*** (0.527)	4.605*** (0.265)	4.520*** 0.264)	-4.236*** (0.160)	-4.327*** (0.159)	-4.910*** (0.369)	-5.051*** (0.372)
	Jack.-Core N=1,543		**Jack.-Datab. N=5,228**		**Jack.-XML N=1,128**		**Com.-JxPath N=598**		**Joda-Time N=2,094**		**Clo.-Compiler N=17,171**	
	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
DiffWMC			-2.330 (1.546)	3.344** (1.619)			-14.452*** (2.914)	41.577*** (3.503)			-8.302*** (0.006)	-3.841*** (0.004)
DiffNOC	-37.085*** (0.359)	-93.807*** (0.372)	-58.836*** (0.037)	-152.598*** (0.043)	-0.066*** (0.020)	-0.235*** (0.007)	4.203*** (0.113)	-1.798*** (0.040)	-1.001*** (0.007)	-0.723*** (0.021)	0.617*** (0.0003)	2.750*** (0.0003)
DiffLCOM	-0.024 (0.029)	-0.008 (0.031)	0.155*** (0.049)	0.179*** (0.045)	-0.016 (0.244)	-0.420 (0.478)	0.217 (1.161)	-1.167 (1.509)	-0.835 (0.867)	-0.255 (2.234)	0.076 (0.086)	0.034 (0.069)
DiffDIT			-70.763*** (0.028)	-124.104*** (0.029)							0.391 *** (0.0003)	-0.665*** (0.0002)
DiffCBO	-2.840 (5.329)	-8.386 (5.859)	1.062 (0.979)	2.261* (1.194)	-1.162 (3.528)	18.673* (10.142)	-11.521 (12.928)	-28.867*** (9.482)	5.642*** (0.319)	-7.746*** (0.028)	-7.895*** (0.003)	3.860*** (0.002)
DiffRFC							8.152 (7.483)	-47.303*** (8.690)				
DiffLOC	-0.039 (0.053)	0.039 (0.047)	-0.045 (0.147)	-0.013 (0.141)	-0.009 (0.228)	0.077 (0.639)	-3.004* (1.736)	5.674*** (1.428)	0.645 (0.699)	1.564 (2.364)	0.615*** (0.153)	0.035 (0.116)
DiffDelegations	0.001 (0.003)	0.003 (0.003)	-0.005 (0.003)	-0.010*** (0.003)	0.027 (0.043)	0.005 (0.066)	0.201* (0.104)	0.399*** (0.100)	0.032 (0.051)	-0.074 (0.117)	-0.003 (0.002)	0.002 (0.002)
DiffSpecInh	-0.371 (0.439)	0.491 (0.407)	0.109 (0.095)	-0.065 (0.100)	-0.489 (2.984)	1.159 (5.170)	-4.686* (2.742)	0.161 (1.719)	-1.633 (3.875)	-6.170*** (0.250)	0.002 (0.278)	-0.048 (0.268)
DiffimpInh			0.018 (0.099)	0.341*** (0.090)	-0.461 (1.068)	-0.430 (2.108)	-2.314 (1.718)	-15.482*** (4.800)	-1.313 (2.317)	1.200 (4.509)	0.133 (0.118)	-0.034 (0.096)
Churns	-0.001 (0.002)	-0.004 (0.004)	-0.001 (0.001)	-0.004** (0.002)	0.002 (0.006)	-8.872*** (0.001)	-0.015* (0.008)	-0.026*** (0.007)	-0.006(0.006)	-0.019 (0.013)	-0.001 (0.0004)	-0.00001 (0.0001)
Constant	-4.183*** (0.237)	-4.082*** (0.237)	-4.048*** (0.118)	-4.043*** (0.127)	-5.312*** (0.440)	-5.346*** (0.518)	-3.345*** (0.262)	-3.323*** (0.264)	-4.545*** (0.243)	-4.462*** (0.251)	-4.624*** (0.081)	-4.654*** (0.080)

Significance codes: *p<0.1; **p<0.05; ***p<0.01.

Delegation and Inheritance do not influence the defect proneness of source code

The reusability metrics positively influence the decrease of maintenance effort

Conclusion

The main objective of this Ph.D. project is to understand how reusability mechanisms evolve over time in complex systems and their impact on code quality and reliability

To assess this study, we identified two research questions

- How software reusability evolves over time?
- How can reusability mechanisms be used to predict code quality and reliability in complex systems?

To archive these results, we developed a tool to extract reusability metrics in Traditional Systems performed two preliminary analyses to understand how reusability impacts code smells, defect proneness, and maintenance effort, and analyzed the impact on vulnerability in complex systems

List of Publications

Journal

- "*On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort*",

  G. Giordano, G. Festa, G. Catolino, F. Palomba, F. Ferrucci, and C. Gravino,

  (**Submitted at Empirical Software Engineering - EMSE)**

- "*On the use of artificial intelligence to deal with privacy in IoT systems: A systematic literature review*",

  G. Giordano, F. Palomba, and F. Ferrucci,

  Journal of Systems and Software (JSS)

International Conferences

- "*On the adoption and effects of source code reuse on defect proneness and maintenance effort*," (Registered Report)

  G. Giordano, G. Festa, G. Catolino, F. Palomba, F. Ferrucci, and C. Gravino, 2022.

  IEEE **International Conference on Software Maintenance and Evolution (ICSME)**. Limassol, Cyprus.

- "A preliminary conceptualization and analysis on automated static analysis tools for vulnerability detection in android apps."

  G. Giordano, F. Palomba, and F. Ferrucci,

  Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Maspalomas, Gran Canaria, Spain.

- "On the evolution of inheritance and delegation mechanisms and their impact on code quality",

  G. Giordano, A. Fasulo, G. Catolino, F. Palomba, F. Ferrucci, and C. Gravino,

  IEEE **International Conference on Software Analysis, Evolution and Reengineering (SANER),** 2022,

Learning and Teaching Activities

Learning Activities

- Winter School on Software Engineering. Virtual
- 5th International Summer School on Software Engineering (ISSSE). Virtual.
- 20th Belgium-Netherlands Software Evolution Workshop - BENEVOL 2021.
- Introduction to Machine Learning Course held by Prof. Dario Di Nucci and Prof. Gemma Catolino - Jheronimus Academy of Data Science (JADS), Netherlands - 2021
- Virtual - Facebook Testing and Verification Symposium - Facebook TAV 2020.

Teaching Activities

I served as a teaching fellow for the courses: Computer Science Education, Fundamentals of Artificial Intelligence, Software Dependability, Software Engineering, Software Engineering for A.I., Software Metrics and Quality, Software Project Management

External Collaborations

Abroad Period

I collaborated with the Jheronimus Academy of Data Science (JADS) Netherlands for 9 months (starting on 3 April).

During this period, I worked on software reuse in traditional systems.

We published a registered report paper on software reusability and its impact on code quality metrics. This work was published at **ICSME**.

In addition, we have extended the paper and submitted it to **EMSE** Journal.

Industrial Period

I collaborated with an Italian Company (Business Solution S.L.R.) for three months (starting on 27 July). During this period, I worked on vulnerabilities in mobile applications.

We published a full paper at **Euromicro Conference (SEAA).**

Professional Activities

I served as Program Committee Member or publicity chair at several International Conferences and workshops.

- 20th International Conference on Mining Software Repositories (**MSR** 2022) **Program Committee Member**, Melbourne, Australia
- 7th International Conference on Software Engineering Advances (**ICSEA** 2022) **Program Committee Member**, Lisbon, Portugal
- 1st Workshop on Software Quality Assurance for Artificial Intelligence **Web and Publicity Chair** SQA4AI 2022, Virtual
- 6th International Conference on Software Engineering Advances (**ICSEA 2021**) **Program Committee Member**, Barcelona, Spain

I reviewed more than 20 papers for International Conferences and Journals.

RQ1₂ - On the role of Static Analyses Tools to detect Vulnerabilities in Android Applications

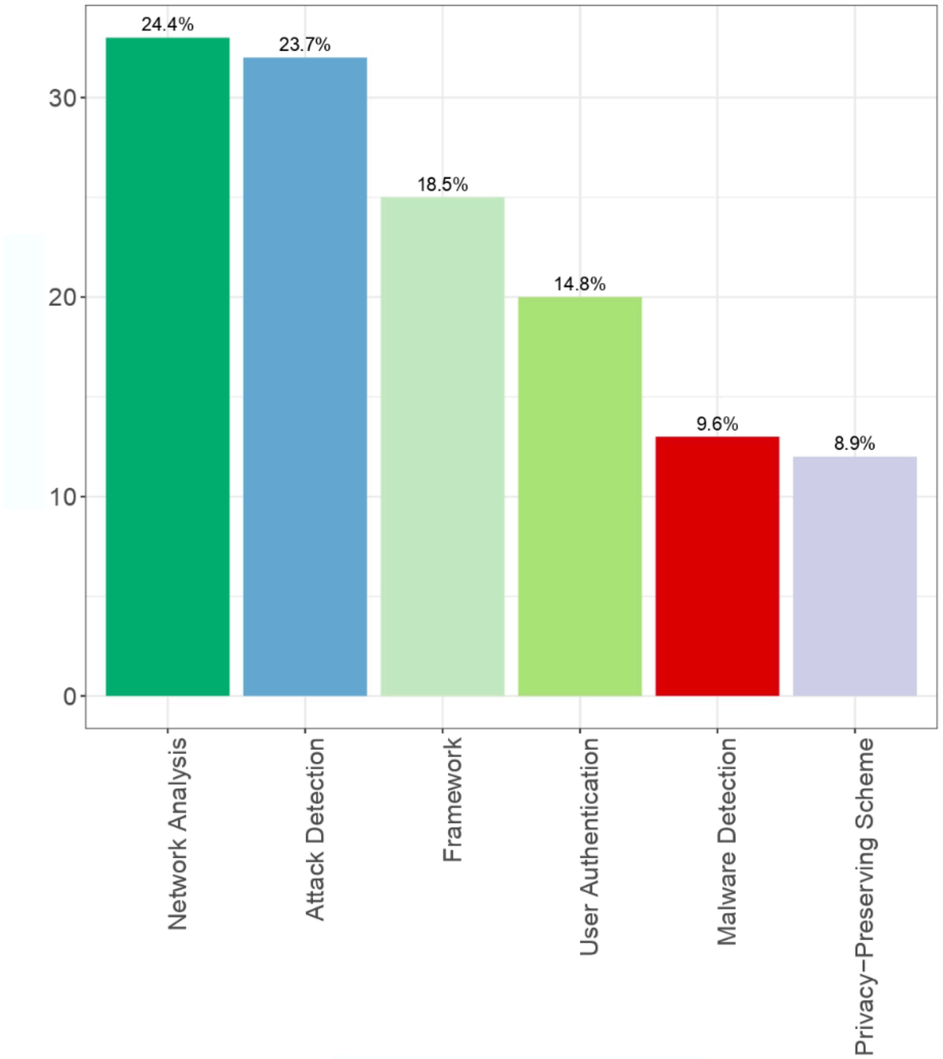
	Com.-Codec N=2,134	Com.-Cli N=1,099	Com.-Col. N=3,560	Com.-CSV N=1,634	Comp. N=3,305	Gson N=1,478
DiffWMC	163.951 (105.295)	26.263 (227.457)	-20.375 (56.965)	-20.375 (56.965)	-1,988.919*** (210.722)	-377.039 (269.203)
DiffNOC			10,213.080*** (2,341.143)	-132.074 (1,357.614)	-10,740.970*** (1,699.369)	17,827.570*** (3,288.230)
DiffLCOM	1.383 (2.713)	-12.154 (16.169)	7.799* (4.680)	10.673*** (1.905)	26.285*** (6.021)	-15.488 (12.955)
DiffDIT	3,341.228*** (903.732)	-2,378.489 (1,497.270)	-1,787.167 (1,192.199)		52,852.530*** (2,813.141)	-6,958.231** (2,826.673)
DiffCBO	1,021.357*** (150.161)	-108.063 (134.420)	6,717.225*** (652.191)	-56.282 (94.958)	5,529.115*** (307.003)	1,916.428*** (145.944)
DiffRFC		192.611*** (57.094)	4.682 (60.012)			
DiffLOC	1.293 (2.158)	-9.992* (5.254)	-58.840*** (10.760)	2.994 (2.693)	5.769 (5.471)	46.604*** (10.894)
Delegation	0.017 (0.045)	-0.697*** (0.229)	-0.003 (0.057)	0.165 (0.124)	-0.080*** (0.022)	-0.119** (0.050)
SpecInh	-1.217 (3.145)	39.595*** (11.915)	1.143 (1.211)	-6.889 (6.984)	-0.710 (1.950)	1.415 (1.301)
ImpInh	-0.131 (1.747)	-1.026 (0.870)	-0.386 (4.387)		3.653*** (1.093)	2.080 (2.226)
BugDecrease	1.433 (37.641)	-74.159 (102.978)	-106.139 (620.995)	-1.272 (69.685)	-26.208 (85.330)	-6.128 (90.985)
BugIncrease	-20.191 (37.620)	-70.799 (101.593)	-111.854 (620.996)	-14.892 (69.652)	-15.856 (86.311)	-14.031 (96.408)
Constant	52.948*** (15.918)	126.288** (51.413)	103.349 (77.155)	11.161 (20.003)	91.271*** (25.693)	111.100* (63.054)
	Jack.-Core N=1,543	**Jack.-Datab. N=5,228**	**Jack.-XML N=1,128**	**Com.-JxPath N=598**	**Joda-Time N=2,094**	**Clo.-Compiler N=17,171**
DiffWMC		853.627*** (71.347)		-889.089 (1,208.343)		-35,485.190 *** (1,024.124)
DiffNOC	21,588.520*** (1,212.595)	22,830.430*** (1,564.318)	333.786 (509.649)	24,786.920*** (5,760.288)	54,104.760*** (3,864.815)	204,776.100*** (25,377.820)
DiffLCOM	1.241 (0.765)	-28.862*** (1.208)	-8.269*** (0.992)	21.501*** (5.505)	189.720*** (23.536)	454.243*** (9.841)
DiffDIT		50,782.460*** (1,712.723)				305,846.900*** (27,225.780)
DiffCBO	1,682.687*** (131.899)	3,147.449*** (74.440)	239.229*** (19.892)	3,504.462*** (363.997)	-31,929.670*** (2,814.595)	-472.842 (643.145)
DiffRFC				1,358.532*** (363.735)		
DiffLOC	28.585*** (1.371)	-8.353*** (3.029)	8.568*** (0.946)	-158.255*** (12.875)	344.715*** (42.978)	1,028.922*** (38.580)
Delegation	-0.012 (0.017)	0.010* (0.006)	-0.096** (0.044)	-0.598** (0.294)	-0.580*** (0.107)	-0.014* (0.008)
SpecInh	2.701 (3.677)	-1.281*** (0.305)	-2.847 (2.324)	6.773 (14.133)	-156.445*** (22.026)	0.868 (0.737)
ImpInh		-0.140 (0.499)	3.001** (1.408)	0.942 (3.559)	179.745*** (20.294)	0.406 (0.407)
BugDecrease	83.885* (48.156)	28.236 (19.492)	19.831 (23.738)	-41.147 (149.800)	-625.949 (602.612)	-73.094 (91.940)
BugIncrease	-51.820 (48.181)	-7.043 (20.359)	-20.624 (26.005)	14.086 (149.626)	-690.321 (618.444)	-52.403 (91.954)
Constant	25.537 (47.505)	182.664*** (57.987)	57.931*** (9.413)	1,561.449*** (387.616)	-6,729.363*** (706.442)	89.824 (76.824)

Significance codes: *p<0.1; **p<0.05; ***p<0.01.

	Com.-Codec N=2,134		Com.-Cli N=1,099		Com.-Col. N=3,560		Com.-CSV N=1,634		Comp. N=3,305		Gson N=1,478	
	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
DiffWMC	-10.098 (7.495)	2.280 (10.981)	-0.691 (3.434)	2.416 (3.602)			-3.539 (2.559)	0.627 (5.136)	-5.248*** (0.033)	-1.903 (4.387)	3.261 (30.899)	-1.305 (25.907)
DiffNOC					-0.038 (0.050)	0.004 (0.013)	-4.413*** (0.156)	1.052*** (0.176)	10.188*** (0.002)	-5.653*** (0.051)	-0.159 (0.275)	1.536*** (0.415)
DiffLCOM	0.092 (0.140)	0.054 (0.261)	0.166 (0.256)	-0.744*** (0.244)	0.422 (0.808)	0.476 (22.615)	-0.056 (0.066)	-0.040 (0.130)	-0.066 (0.335)	0.046 (0.125)	0.013 (1.242)	-0.159 (1.157)
DiffDIT	11.927*** (0.269)	−0.183*** (0.033)	0.012 (5.830)	-0.0003 (0.023)	0.012 (5.830)	-0.0003 (0.023)	-4.526*** (0.238)	0.661*** (0.169)	12.511*** (0.002)	-5.151*** (0.125)	0.696 (0.466)	1.896** (0.798)
DiffCBO	-5.434 (5.898)	-9.729*** (0.243	-0.645 (3.617)	-5.947 (3.821)	-0.878 (58.069)	-0.495*** (0.103)	-0.994 (4.485)	-3.484 (8.728)	-4.163*** 0.021)	1.467 (2.717)	-17.977 (12.462)	-3.472 (12.553)
DiffRFC	4.123 (5.784)	-0.030 (1.106)	-0.014 (1.027)	4.123 (5.784)	1.013 (6.087)	1.924 (14.548)						
DiffLOC	0.005 (0.302)	0.075 (0.346)	0.002 (0.139)	0.056 (0.200)	-0.611 (1.053)	-0.099 (11.153)	0.175 0.121)	0.045 (0.236)	0.149* (0.090)	0.024 (0.104)	0.115 (1.273)	0.689 (1.373)
DiffDelegations	0.058 (0.049)	-0.060 (0.077	0.017 (0.022)	0.001 (0.025	-0.069 (0.137)	0.004 (0.654)	0.031 (0.076)	-0.058 (0.147)	0.013 (0.017)	-0.003 (0.013)	0.068 (0.059)	-0.018 (0.078)
DiffSpecInh	-1.791 (1.685)	-1.510 (3.395	0.070 (0.618)	1.382*** (0.542)	1.013 (6.087)	1.924 (14.548)	-0.571 (5.267)	-1.495 (10.178)	-0.187 (0.862)	-0.148 (0.637)	-0.356 (3.632)	0.226 (1.865)
DiffimpInh	-0.060 (0.940)	0.046 (2.134			0.767 (3.457)	0.771 (13.557)	-1.141 (2.923)	-0.094 (4.432)	-0.337 (0.488)	0.154 (0.383)	0.047 (2.105)	0.267 (1.713)
Churns	-0.002 (0.003)	-0.002 0.004)	-0.002 (0.002)	-0.005 (0.004)	-0.026 (0.038)	-0.100 (0.127)	-0.004 (0.007)	-0.009 (0.013)	-0.003 (0.002)	-0.0003 (0.001)	-0.015 (0.014)	-0.007 (0.009)
Constant	-4.762*** (0.248)	-4.717*** (0.246)	-3.472*** (0.187)	-3.448*** (0.187)	6.429*** (0.536)	-6.230*** (0.527)	4.605*** (0.265)	4.520*** 0.264)	-4.236*** (0.160)	-4.327*** (0.159)	-4.910*** (0.369)	-5.051*** (0.372)
	Jack.-Core N=1,543		**Jack.-Datab. N=5,228**		**Jack.-XML N=1,128**		**Com.-JxPath N=598**		**Joda-Time N=2,094**		**Clo.-Compiler N=17,171**	
	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
DiffWMC			-2.330 (1.546)	3.344** (1.619)			-14.452*** (2.914)	41.577*** (3.503)			-8.302*** (0.006)	-3.841*** (0.004)
DiffNOC	-37.085*** (0.359)	-93.807*** (0.372)	-58.836*** (0.037)	-152.598*** (0.043)	-0.066*** (0.020)	-0.235*** (0.007)	4.203*** (0.113)	-1.798*** (0.040)	-1.001*** (0.007)	-0.723*** (0.021)	0.617*** (0.0003)	2.750*** (0.0003)
DiffLCOM	-0.024 (0.029)	-0.008 (0.031)	0.155*** (0.049)	0.179*** (0.045)	-0.016 (0.244)	-0.420 (0.478)	0.217 (1.161)	-1.167 (1.509)	-0.835 (0.867)	-0.255 (2.234)	0.076 (0.086)	0.034 (0.069)
DiffDIT			-70.763*** (0.028)	-124.104*** (0.029)							0.391 *** (0.0003)	-0.665*** (0.0002)
DiffCBO	-2.840 (5.329)	-8.386 (5.859)	1.062 (0.979)	2.261* (1.194)	-1.162 (3.528)	18.673* (10.142)	-11.521 (12.928)	-28.867*** (9.482)	5.642*** (0.319)	-7.746*** (0.028)	-7.895*** (0.003)	3.860*** (0.002)
DiffRFC							8.152 (7.483)	-47.303*** (8.690)				
DiffLOC	-0.039 (0.053)	0.039 (0.047)	-0.045 (0.147)	-0.013 (0.141)	-0.009 (0.228)	0.077 (0.639)	-3.004* (1.736)	5.674*** (1.428)	0.645 (0.699)	1.564 (2.364)	0.615*** (0.153)	0.035 (0.116)
DiffDelegations	0.001 (0.003)	0.003 (0.003)	-0.005 (0.003)	-0.010*** (0.003)	0.027 (0.043)	0.005 (0.066)	0.201* (0.104)	0.399*** (0.100)	0.032 (0.051)	-0.074 (0.117)	-0.003 (0.002)	0.002 (0.002)
DiffSpecInh	-0.371 (0.439)	0.491 (0.407)	0.109 (0.095)	-0.065 (0.100)	-0.489 (2.984)	1.159 (5.170)	-4.686* (2.742)	0.161 (1.719)	-1.633 (3.875)	-6.170*** (0.250)	0.002 (0.278)	-0.048 (0.268)
DiffimpInh			0.018 (0.099)	0.341*** (0.090)	-0.461 (1.068)	-0.430 (2.108)	-2.314 (1.718)	-15.482*** (4.800)	-1.313 (2.317)	1.200 (4.509)	0.133 (0.118)	-0.034 (0.096)
Churns	-0.001 (0.002)	-0.004 (0.004)	-0.001 (0.001)	-0.004** (0.002)	0.002 (0.006)	-8.872*** (0.001)	-0.015* (0.008)	-0.026*** (0.007)	-0.006(0.006)	-0.019 (0.013)	-0.001 (0.0004)	-0.00001 (0.0001)
Constant	-4.183*** (0.237)	-4.082*** (0.237)	-4.048*** (0.118)	-4.043*** (0.127)	-5.312*** (0.440)	-5.346*** (0.518)	-3.345*** (0.262)	-3.323*** (0.264)	-4.545*** (0.243)	-4.462*** (0.251)	-4.624*** (0.081)	-4.654*** (0.080)

Significance codes: *p<0.1; **p<0.05; ***p<0.01.

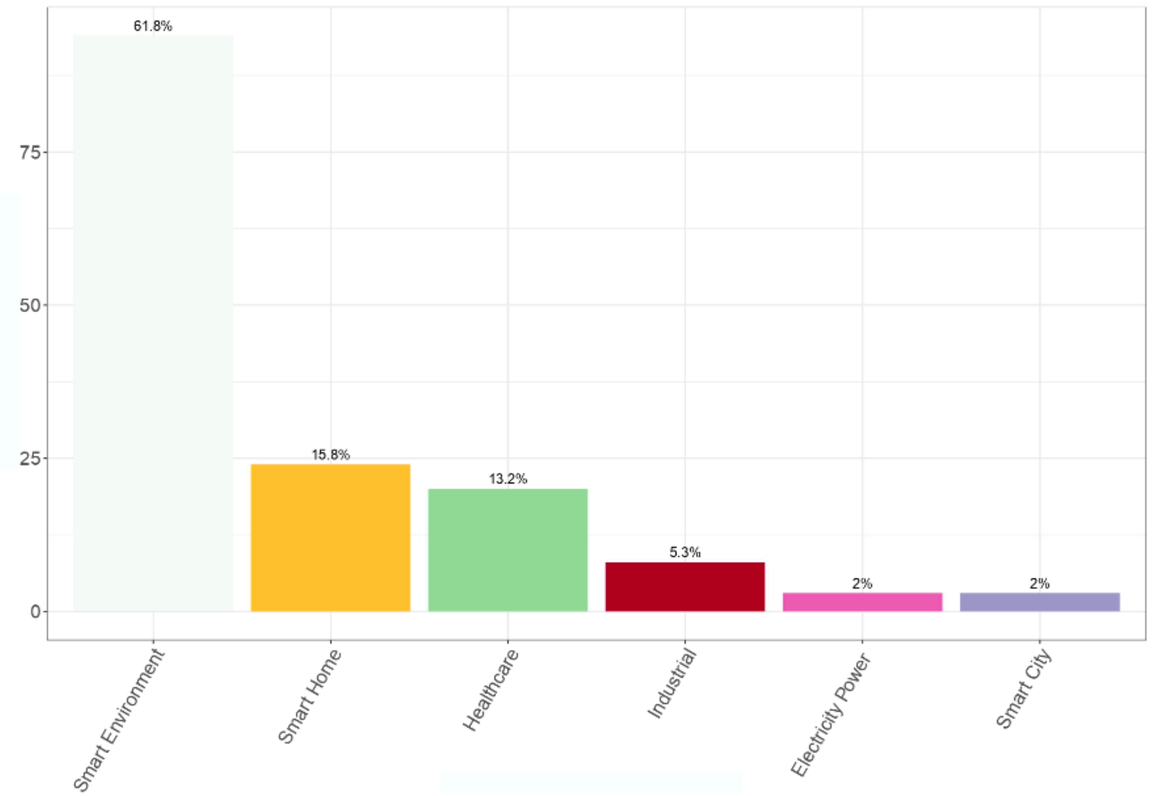
RQ1 - On the privacy tasks tackled with the use of artificial intelligence technique



To respond of the **RQ1,** we manually classified each paper

Most papers use artificial intelligence algorithms to analyze network traffic, or to detect possible malware or attacks

RQ2 - On the IoT domains where artificial intelligence techniques have been applied



In 62% of the cases, the authors do not specify the domain where artificial intelligence techniques have been applied.

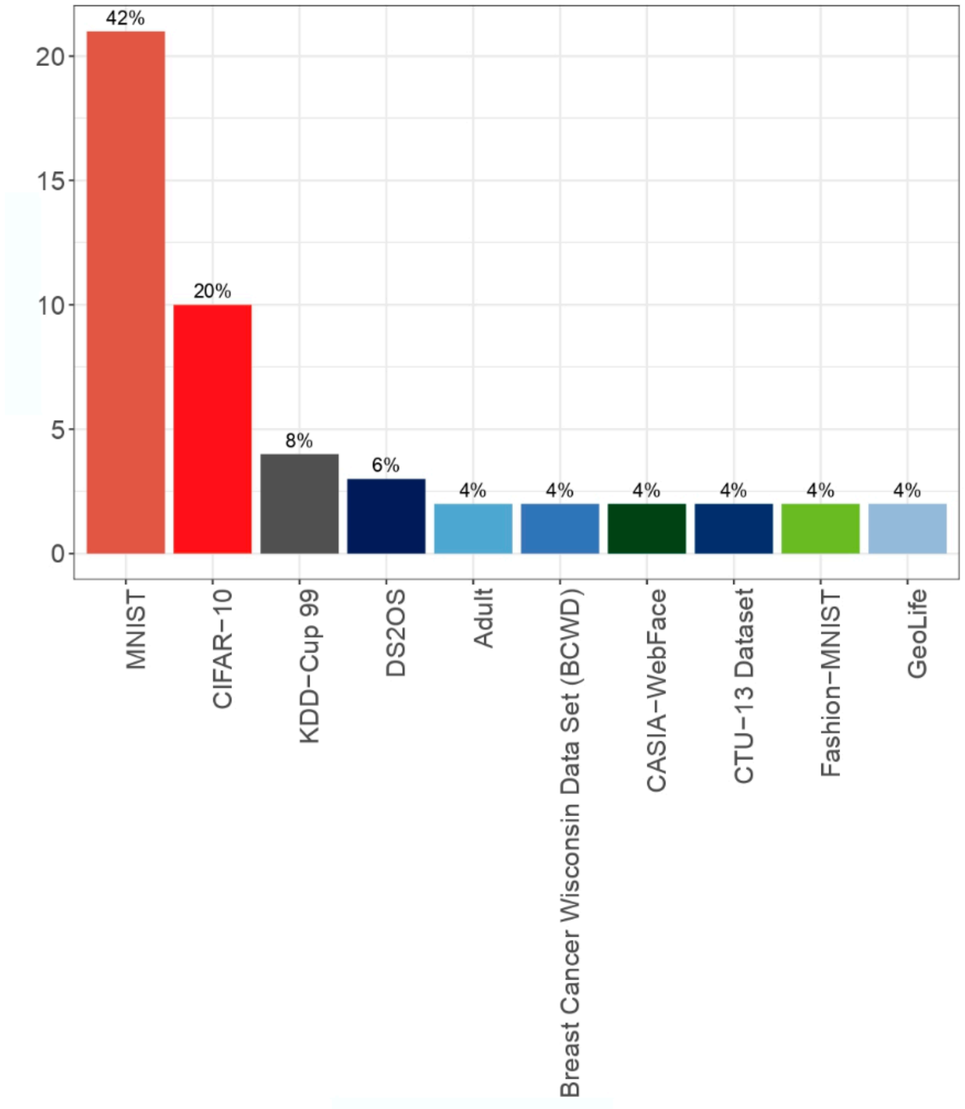
This outcome suggests that artificial intelligence algorithms could be considered “context-independent”.

RQ3 - On the families of artificial intelligence algorithms used to deal with privacy

Most of the proposed approaches to deal with privacy in IoT systems use **supervised learning** techniques. In particular, according to the literature, **Random Forest** is the most applied.

However, we identified a lack of analyses on other type of artificial intelligence approaches, especially regards **deep learning** approaches

RQ4 - On the dataset employed by artificial intelligence algorithms



In almost half of the nearly, papers use **MNIST** dataset to train artificial intelligence algorithms

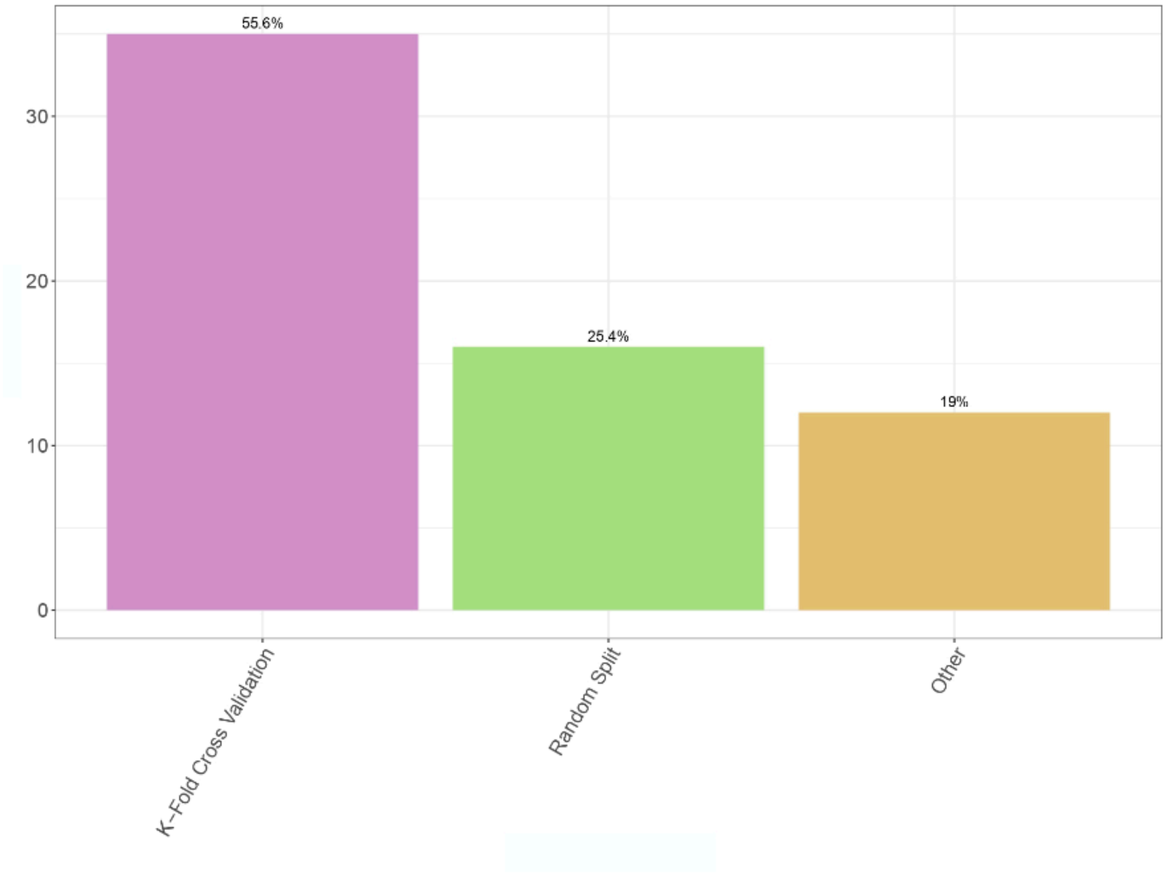
RQ4 - On the dataset employed by artificial intelligence algorithms

Ian Goodfellow

Instead of moving on to harder datasets than MNIST, the ML community is studying it more than ever.
Even proportional to other datasets

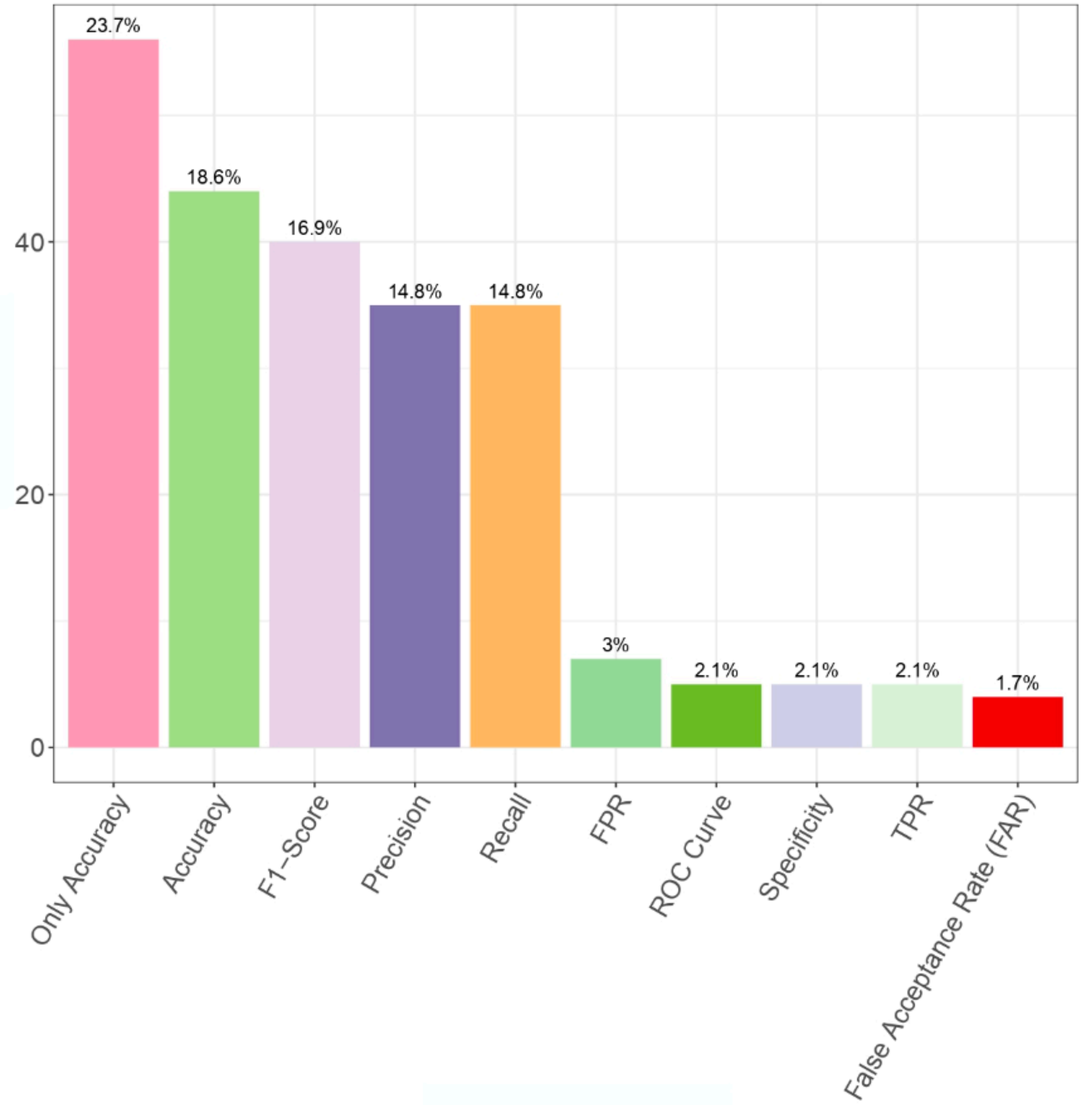
Most researchers indicated that the use of **MNIST** is too easy to train artificial intelligence algorithms, and for this, the results obtained could be affected by biases

RQ5 - On the validation strategies employed to assess the artificial intelligence



In 56% of the cases, the strategy applied to validate artificial intelligence algorithms is the **K-Fold Cross Validation;** however, in several cases, the use of this strategy is unsuitable

RQ6 - On the evaluation metrics employed to assess the artificial intelligence



Most of the proposed studies used to evaluate artificial intelligence algorithms' **accuracy** metric.

However, the characteristics of the datasets might make them biased toward accuracy, implying a biased interpretation of the real capabilities of the proposed techniques

Sum Up

sesa lab
SOFTWARE ENGINEERING
SALERNO

Scan Me!

giagiordano@unisa.it

@GiammariaGiord1

https://broke31.github.io/giammaria-giordano/

Researchers propose artificial intelligence algorithms to deal with privacy, but in most cases, the artificial intelligence pipelines have several concerns.


MIND THE GAP
Fill

Research Gaps

Fill The Gap

Based on the State-of-The-Art, we identified several concerns on the use of artificial intelligence to deal with privacy in IoT systems

The state-of-the-art does not provide information on how third-party libraries used to implement artificial intelligence algorithms can be used to predict privacy or security attributes.

The artificial intelligence techniques used to deal with privacy in IoT Systems do not follow the SE4AI principles, and for this, the results could be affected by biases.

The frameworks and the best practices proposed in the literature to deal with privacy are never applied, which suggests that these frameworks could not be applied in IoT contexts.

State-of-The-Art

What is the impact of third-party libraries on non-functionality requirements in IoT systems?

Are the existing artificial intelligence techniques able to detect security and privacy issues during the information exchange?

Are the proposed frameworks usable to deal with privacy in IoT systems?

State-of-The-Art

What is the impact of software reusability on non-functionality requirements in IoT systems?

Are the existing artificial intelligence techniques able to detect security and privacy issues during the information exchange?

Are the proposed frameworks usable to deal with privacy in IoT systems?

RQ1- On the use of third-party libraries to implement artificial intelligence algorithms

To better address the **RQ1,** we conducted two preliminary steps.
We analyzed the impact of the principal mechanisms of reuse for traditional systems on code quality and reliability during the software evolution

What is the impact of software reusability on non-functionality requirements in traditional systems?

Take into account that in most cases, vulnerabilities in mobile applications depend on the use of unsafe third-party libraries; we conducted a preliminary investigation on the capabilities of Static Analysis Tools to detect vulnerabilities in mobile apps.

How do third-party libraries impact the security and privacy in mobile applications?

State-of-The-Art

What is the impact of software reusability on non-functionality requirements in IoT systems?

Are the existing artificial intelligence techniques able to detect security and privacy issues during the information exchange?

Are the proposed frameworks usable to deal with privacy in IoT systems?

State-of-The-Art

What is the impact of software reusability on non-functionality requirements in IoT systems?

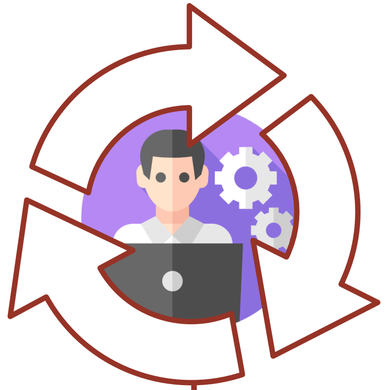
Are the existing artificial intelligence techniques able to detect security and privacy issues during the information exchange?

Are the proposed frameworks usable to deal with privacy in IoT systems?

Methodology

$RQ1_1$- On the use of reusability mechanisms in traditional systems

To address the **$RQ1_1$** we conducted two empirical investigations.



- Code Quality
  - Code Smells
- Reliability
  - Defect-Proneness
  - Maintenance Effort

RQ1₁- On the use of reusability mechanisms in traditional systems

From the code smells perspective, we selected multiple versions of three Java Projects: ANT, JHotDraw, and JEdit

Dependent Variables
Spaghetti Code
Data Should be Private
Complex Class
God Class

Independent Variables
Inheritance
Delegation

Control Variables
CK Metrics

To extract information on Code smells, we used Decor Tool, while for the Independent Variables and the Control Variables, we built a tool “ResueMetrics”

RQ1$_1$- On the use of reusability mechanisms in traditional systems

We labeled as “increase”, “decrease”, or “stable” the difference of reusability between versions I and I+1 and applied the *Multinomial Log-Linear Model*

$RQ1_1$- On the use of reusability mechanisms in traditional systems

To address the **$RQ1_1$** from a reliability standpoint, we selected over 9,000 commits of 12 Java projects provided by Defects4J and extracted information on commits using PyDriller

Dependent Variables
#Bugs
Code Churns

Independent Variables
Inheritance
Delegation

Control Variables
CK Metrics