From Correlation to Causation: Applying Cohort Studies to Mining Software Repositories

Sira Vegas

SIESTA 2026
September 2, 2026
Madrid, Spain

RESEARCH CONDUCTED BY

Sira Vegas

Natalia Juristo

Universidad Politécnica de Madrid,
Madrid, Spain

Sabato Nocera

Giuseppe Scanniello

University of Salerno,
Fisciano, Italy

BASED ON

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3713676

Investigating Causality in Software Repositories Using Cohort-Based Methods

Sira Vegas, Sabato Nocera, Giuseppe Scanniello and Natalia Juristo

Causal or Correlational? A Cohort Study on the Effects of Code Smells on Class Change- and Fault-Proneness

Sabato Nocera*, Sira Vegas†, Giuseppe Scanniello*, Massimiliano Di Penta‡, and Natalia Juristo†
*University of Salerno, †Universidad Politécnica de Madrid, ‡ University of Sannio

ICSE '26, Rio de Janeiro, Brazil

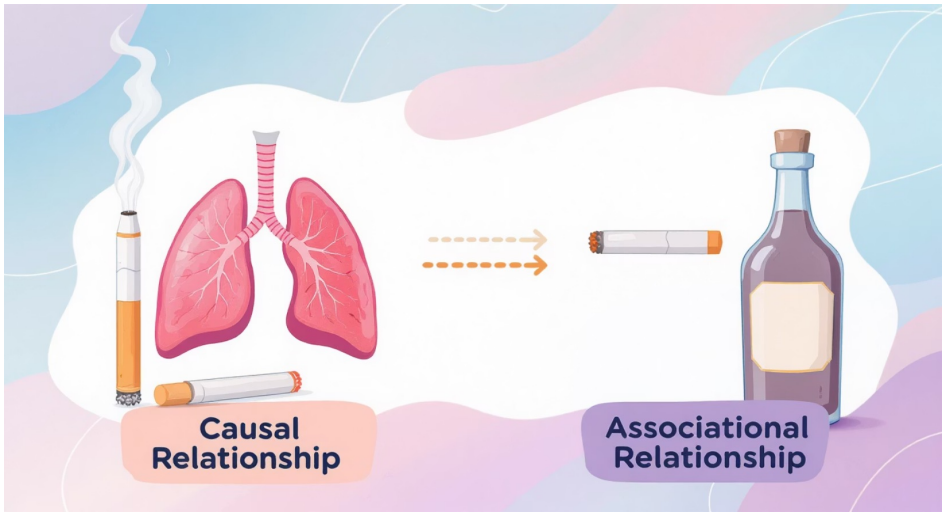
ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787786

CONTENTS

- Introduction
- Conditions for causality
- Strategies for causality
- Cohort studies
- Examples used
- Heuristics
- Conclusions

Introduction
Understanding Causal Relationships

- Not all relationships involve causality
- Causal relationship:
  - A change in one variable results in a change in another



Some SonarQube issues have a significant but small effect on faults and changes. A large-scale empirical study

Valentina Lenarduzzi [a,*], Nyyti Saarimäki [b], Davide Taibi [b]

[a] LUT University, Lahti, Fir
[b] Tampere University, Tam

ARTICLE IN

Article history:
Received 30 August 2019
Received in revised form
Accepted 20 July 2020
Available online 23 July 2

Keywords:
Change-proneness
Fault-proneness
SonarQube
Empirical study

Internal Validity. This threat concerns the internal factors of the study that may have affected the results. We are aware that static analysis tools detect a non-negligible amount of false positives (Johnson et al., 2013), and also the SonarQube's detection accuracy for some rules might not be perfect. However, our goal was to replicate the same conditions adopted by practitioners when using SonarQube, analyzing the change-proneness and fault-proneness of the Sonar issues detected by the default ruleset. Therefore, we did not modify or remove any possible false positives, to accurately reflect the results that developers can obtain running SonarQube on the same projects

Some issues detected by SonarQube were duplicated, reporting the issue violated in the same class and in the same position but with different resolution times. We are aware of this fact, but we did not remove such issues from the analysis since we wanted to report the results without modifying the output provided by SonarQube. We are aware that we cannot claim a direct cause–effect relationship between the presence of Sonar issues and the fault-proneness and change-proneness of classes, that can be influenced by other factors, as this type of investigation would require a controlled experiment. We are also aware that classes with different roles (e.g., classes controlling the business logic) can be more frequently modified than others.

nove technical issues believed to impact software
styles violations.
narQube issues in software systems and to assess
considering also their different types and severities.
va projects from the Apache Software Foundation

faults and 12M changes in Java files. The projects
95K SonarQube issues in more than 200K classes.
hange-prone than affected ones, but the difference
re slightly more change-prone than classes affected
y Vulnerability. As for fault-proneness, there is no
ses. Moreover, we found incongruities in the type

rs to understand which SonarQube issues should be
gaps. Moreover, results can also support companies
s accurately as possible.

An experimental investigation on the innate relationship between quality and refactoring

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

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

Threats to *internal validity* concern factors that could influence our observations. In particular, the fact that code smells disappear, may or may not be related to refactoring activities occurred between the observed releases. In other words, other changes could have produced such effects. However, although the performed analyses and the obtained results allow us to claim correlation and not causation, we corroborate our quantitative results by means of some qualitative analysis, aimed at illustrating examples in which specific kinds of refactorings helped to remove some code smells.

ARTICLE INF

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

Keywords:
Refactoring
Code smells
Empirical study

rs, and
esence
obser-
opers.
object
ed the
cur on
code components for which certain indicators—such as quality metrics or the presence of smells as detected by tools—suggest there might be need for refactoring operations. Results indicate that, more often than not, quality metrics do not show a clear relationship with refactoring. In other words, refactoring operations are generally focused on code components for which quality metrics do not suggest there might be need for refactoring operations. Finally, 42% of refactoring operations are performed on code entities affected by code smells. However, only 7% of the performed operations actually remove the code smells from the affected class.

An experime
quality and r

Gabriele Bavota

[a] Free University of Bozen-B
[b] University of Salerno, Fisci
[c] University of Sannio, Bene
[d] University of Molise, Pesch

ARTICLE IN

Article history:
Received 8 April 2015
Revised 8 May 2015
Accepted 12 May 2015
Available online 21 May 2

Keywords:
Refactoring
Code smells
Empirical study

An exploratory study of the impact of antipatterns on class change- and fault-proneness

Foutse Khomh · Massimiliano Di Penta · Yann-Gaël Guéhéneuc · Giuliano Antoniol

Published online: 6 August 2011

Editor: Jim Whitehead

Threats to *internal validity* do not affect this study, being an exploratory study (Yin 2002). We do not claim causation, but relate the presence of antipatterns with the occurrences of changes, faults, and issues. Nevertheless, we tried to explain—by looking at specific changes, commit notes, and change histories—why some antipatterns could have been the cause of changes/issues/faults. We are aware that antipatterns can be dependent to each other and relied on the logistic regression model-building procedure to select the subset of non-correlated antipatterns. When

be subject to fault-fixing than other classes, (2) to what extent these odds (if higher) are due to the sizes of the classes or to the presence of antipatterns, and (3) what kinds of changes affect classes participating in antipatterns. We show that, in almost all releases of the four systems, classes participating in antipatterns are more change- and fault-prone than others. We also show that size alone cannot explain the higher odds of classes with antipatterns to underwent a (fault-fixing) change than other

l by developers, and
, or on the presence
erature lacks obser-
rmed by developers.
nces of being object
cally, we mined the
activities occur on
f smells as detected
nore often than not,
toring operations are
e might be need for
ities affected by code
lls from the affected

When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)

Michele Tufano[1], Fabio Palomba[2,3], Gabriele Bavota[4]
Rocco Oliveto[5], Massimiliano Di Penta[6], Andrea De Lucia[3], Denys Poshyvanyk[1]

[1]The College of W… The Netherlands
[3]University of… Switzerland,
[5]Univer…), Italy

rocco.ol… …vm.edu

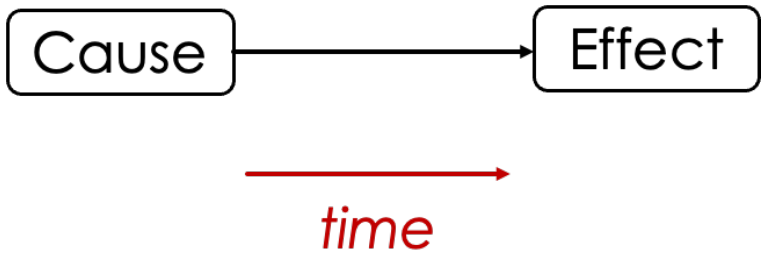
RQ$_2$ we studied tags related to different aspects of a software project lifetime—characterizing commits, developers, and the project status itself—we are aware that there could be many other factors that could have influenced the introduction of smells. In any case, it is worth noting that it is beyond the scope of this work to make any claims related to causation of the relationship between the introduction of smells and product or process factors characterizing a software project.

Abstract—Technic… …he making it right". One noticeable syn… …nd implementation choices. Previous … …ty of code. While the repercussions of smells on code quality have been empirically assessed, there is still only anecdotal evidence on *when* and *why* bad smells are introduced, what is their *survivability*, and *how* they are *removed* by developers. To empirically corroborate such anecdotal evidence, we conducted a large empirical study over the change history of 200 open source projects. This study required the development of a strategy to identify smell-introducing commits, the mining of over half a million of commits, and the manual analysis and classification of over 10K of them. Our findings mostly contradict common wisdom, showing that most of the smell instances are introduced when an artifact is created and not as a result of its evolution. At the same time, 80% of smells survive in the system. Also, among the 20% of removed instances, only 9% is removed as a direct consequence of refactoring operations.

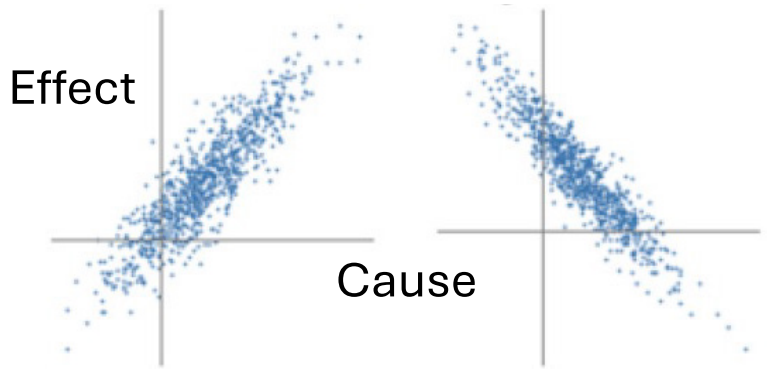
Index Terms—Code Smells, Empirical Study, Mining Software Repositories

Conditions for Causality

Temporal precedence	The value of the independent variable must precede the value of the dependent variable
Empirical association	
Non-spuriousness	

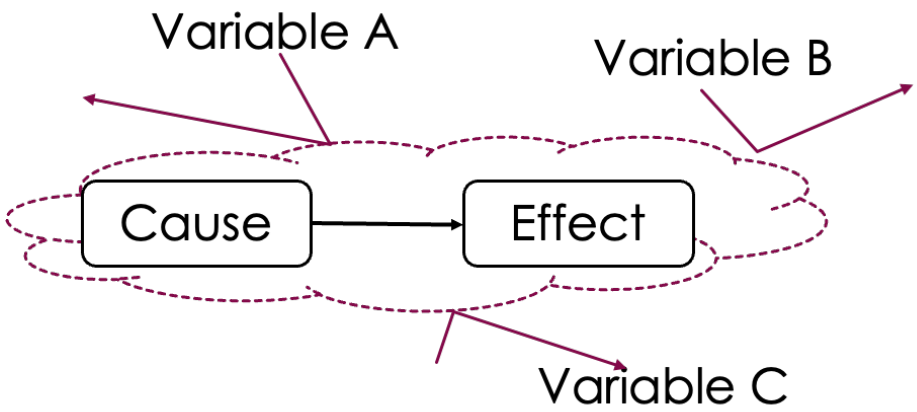


CONDITIONS FOR CAUSALITY



Empirical association	There is a statistical relationship between the independent and dependent variables

CONDITIONS FOR CAUSALITY

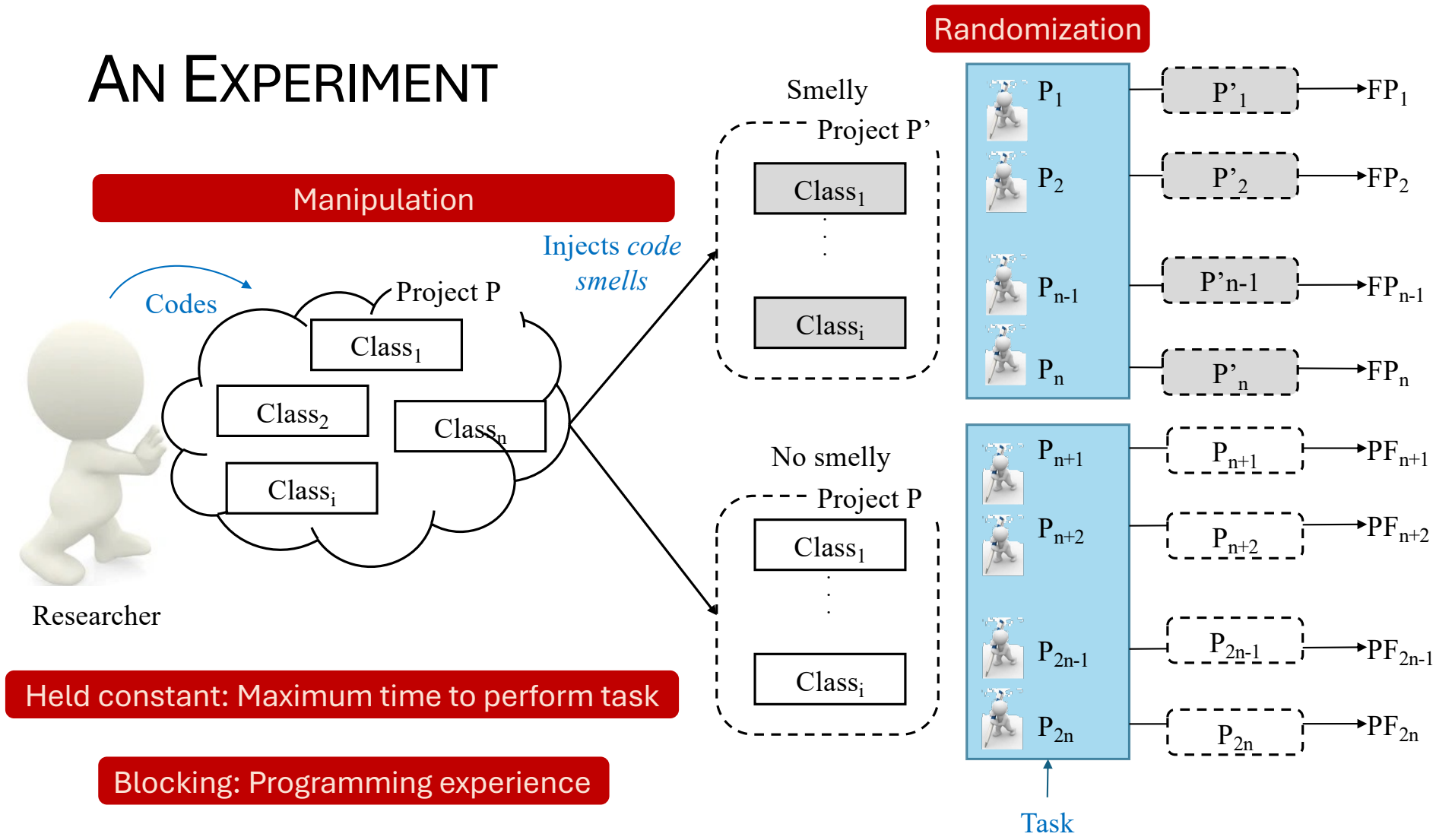


Non-spuriousness	The relationship between cause and effect must not be due to the influence of other variables (confounders)

Strategies for Causality in Controlled Experiments

Temporal precedence	Manipulation		
Non-spuriousness	Held constant	Blocking	Randomization
Empirical association	Statistical analysis		

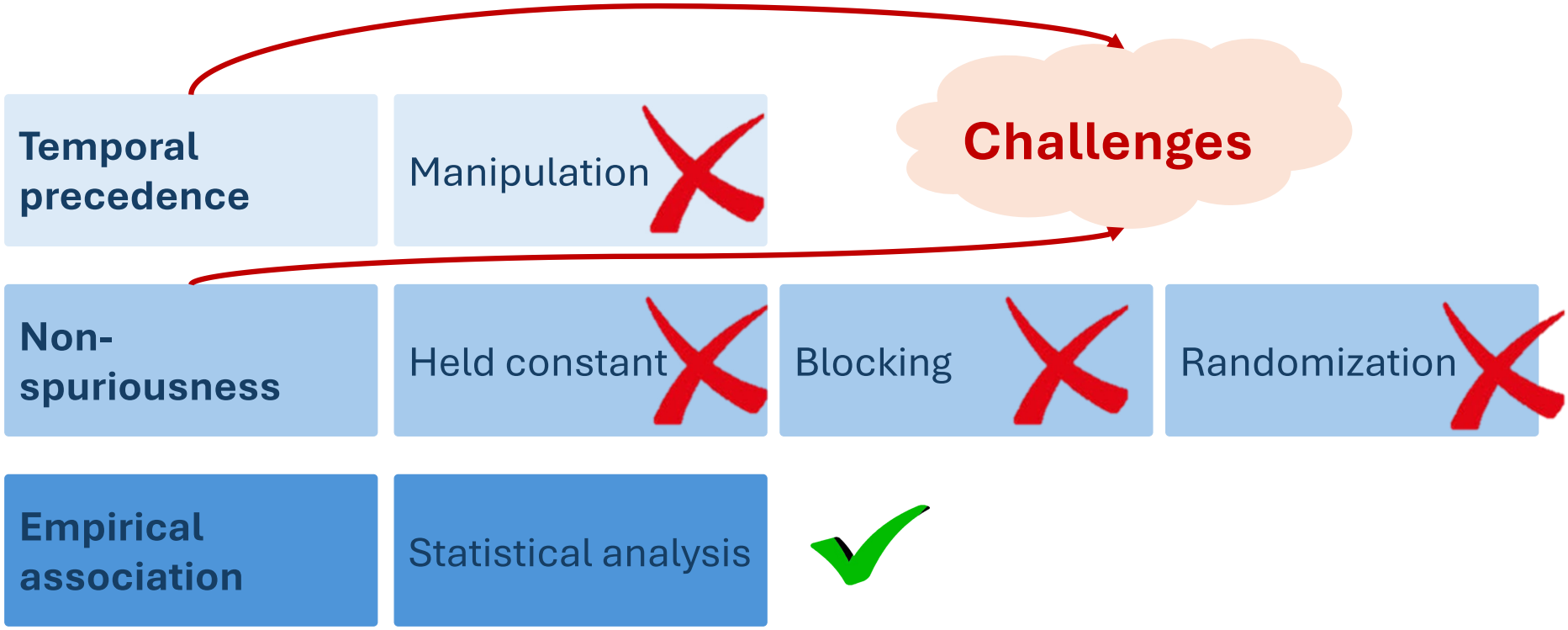
An Experiment



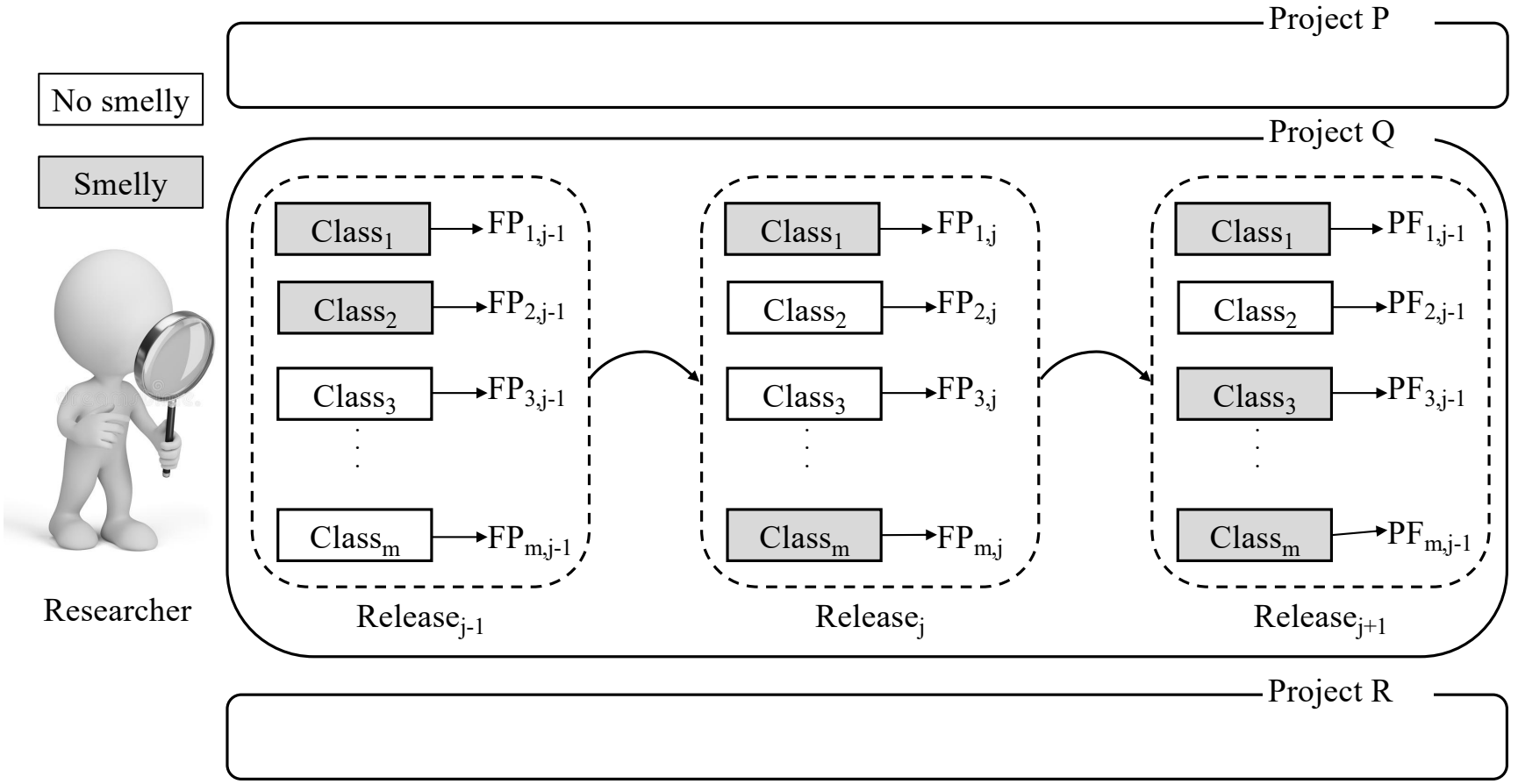
Held constant: Maximum time to perform task

Blocking: Programming experience

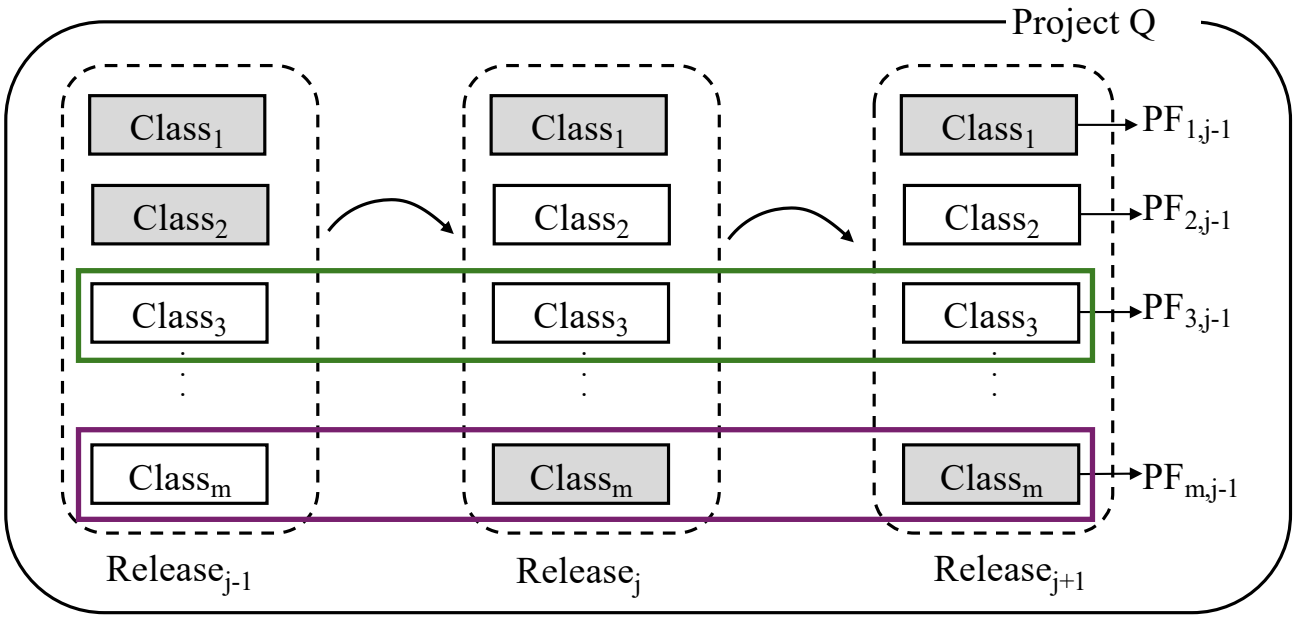
Strategies for Causality with Observational Data



AN OBSERVATIONAL STUDY

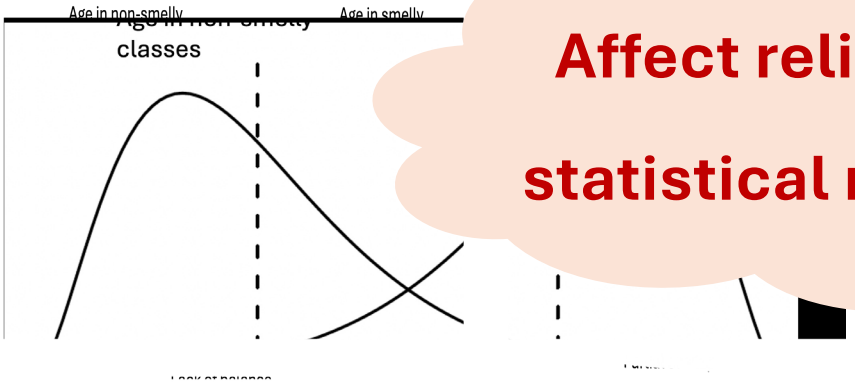


Using Observational Data
Challenge 1: Temporal Precedence



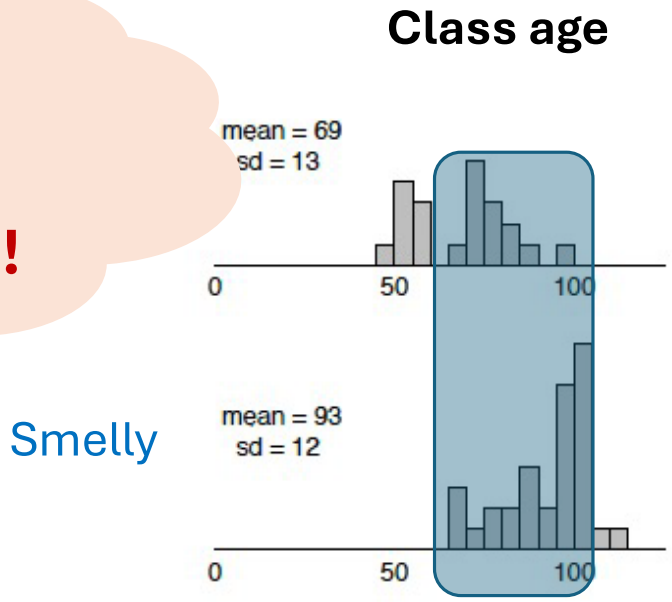
USING OBSERVATIONAL DATA
CHALLENGE 2: NON-SPURIOUSNESS

LACK OF BALANCE

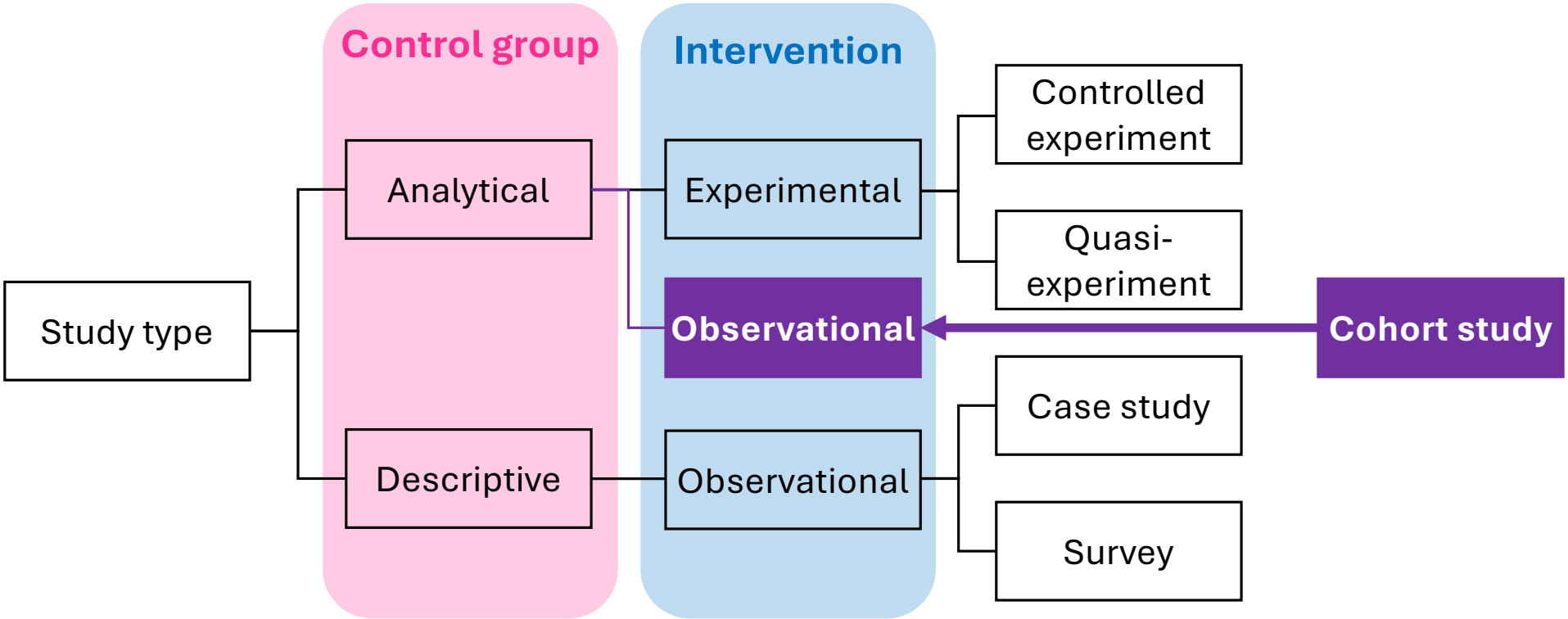


PARTIAL OVERLAP

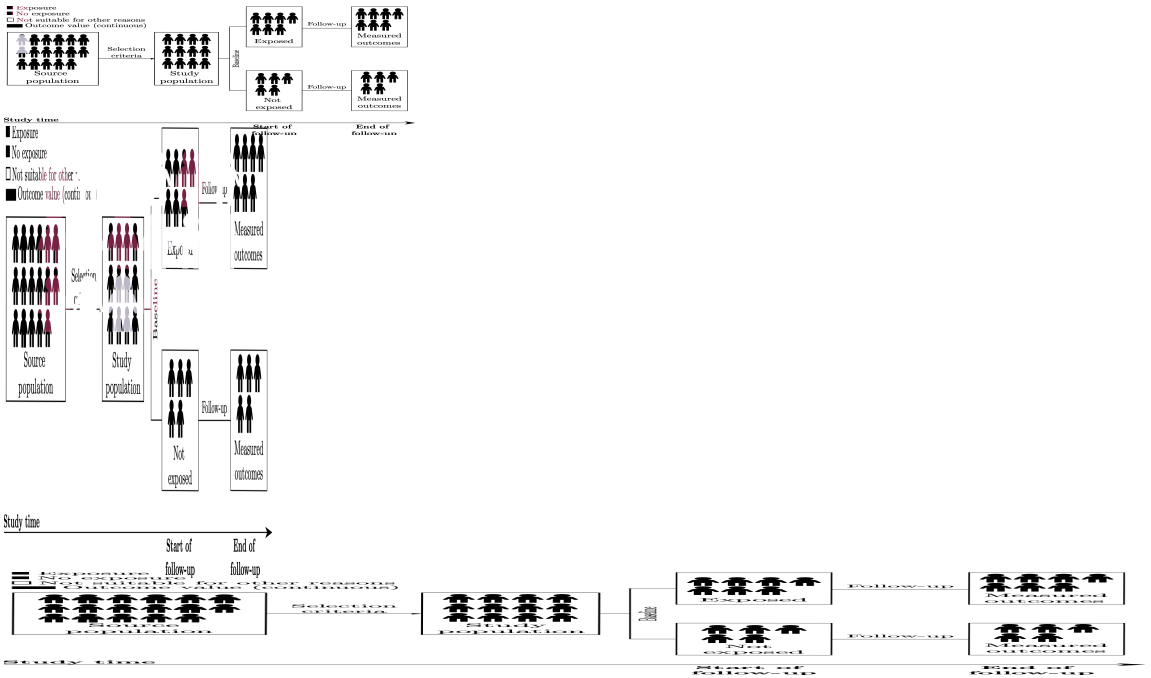
Affect reliability of statistical models!!!!



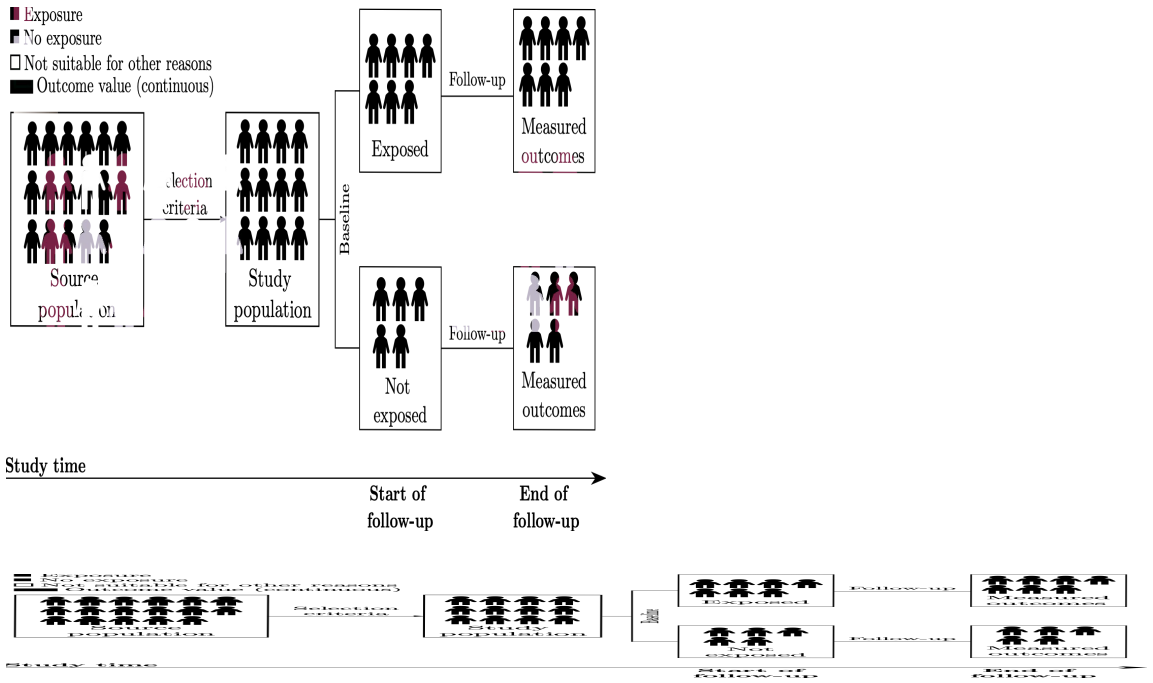
Taxonomy of Studies in SE



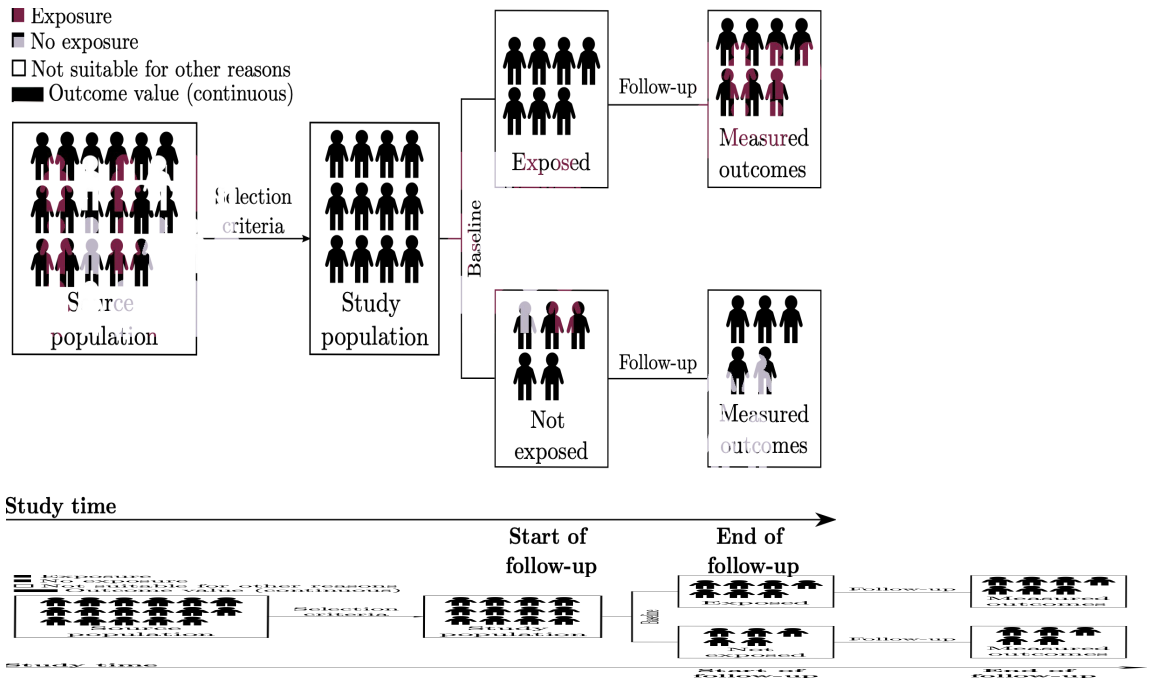
COHORT STUDIES



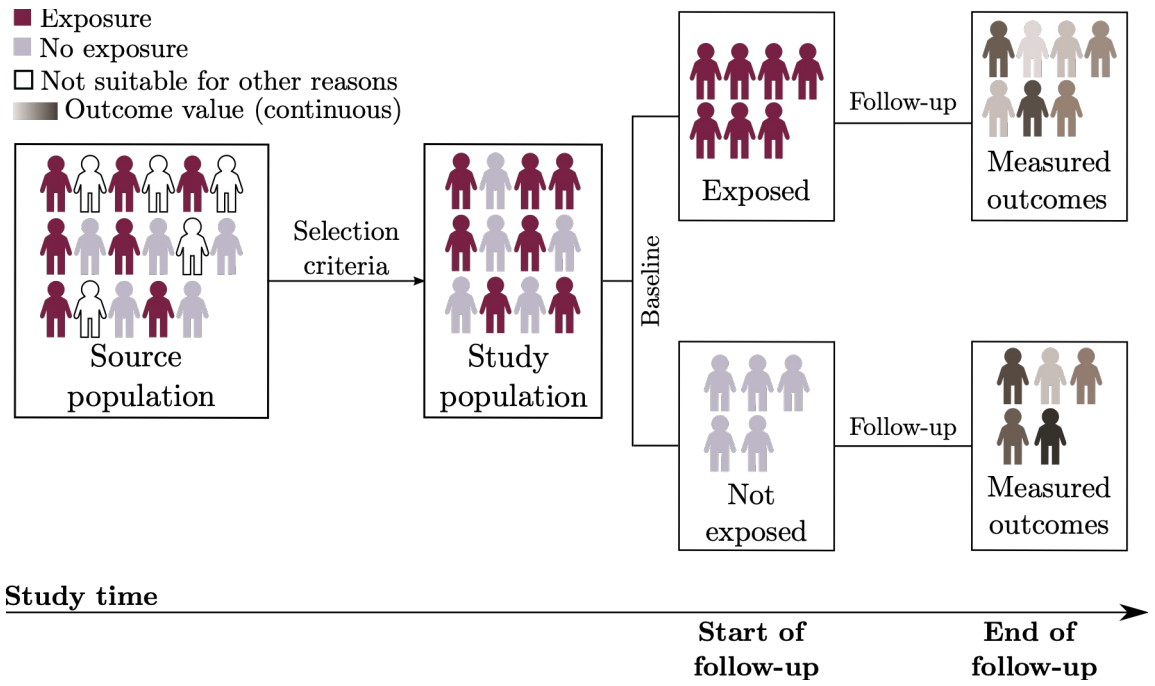
COHORT STUDIES



COHORT STUDIES



COHORT STUDIES



Examples Used

Lung cancer example

- Study on lung cancer
- Investigates the effect of smoking on the development of lung cancer

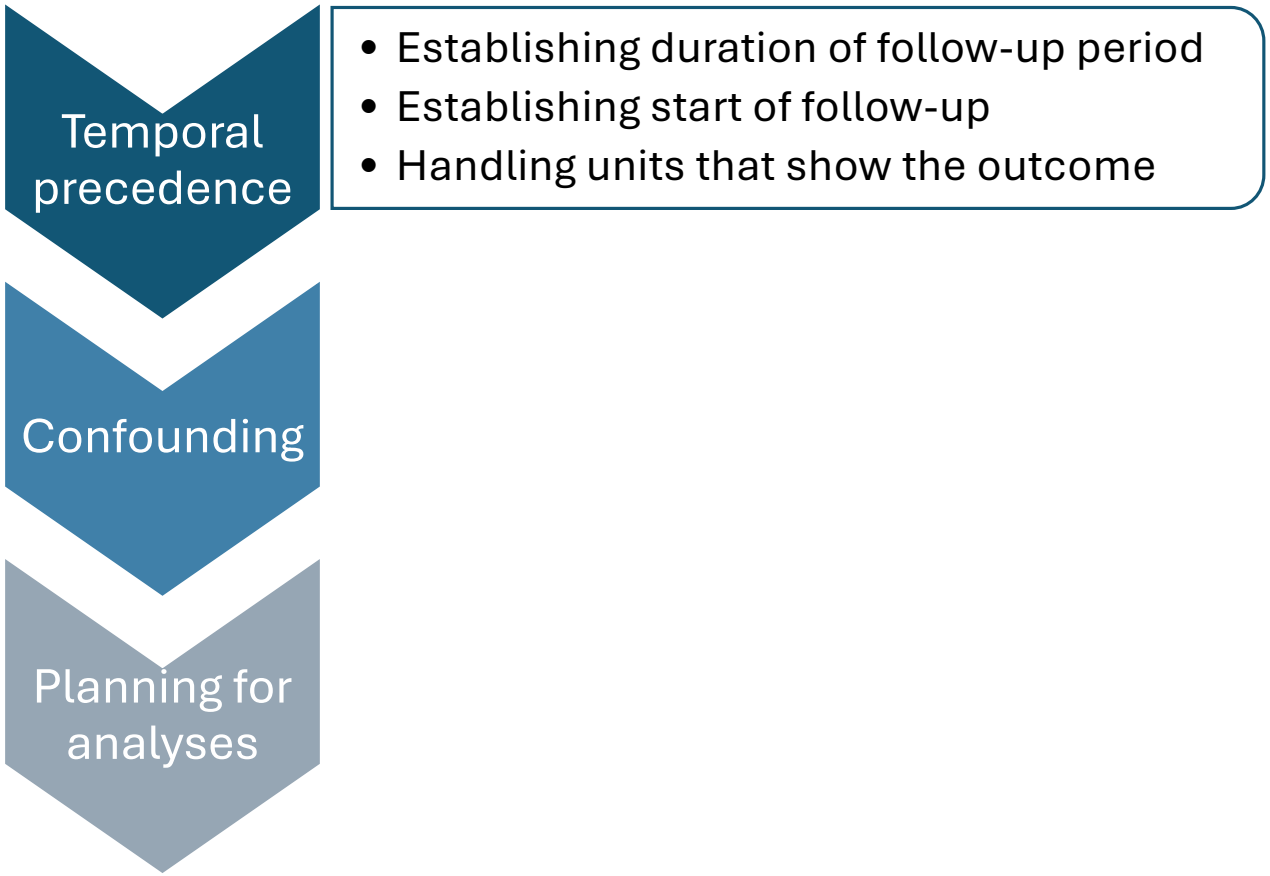
OWASP DC example

- Study on software supply chain security
- Investigates whether adopting a Software Composition Analysis (SCA) tool such as OWASP Dependency-Check (OWASP DC) affects the number of vulnerabilities affecting project components

Code smells example

- Study on code smells
- Investigates the effect of code smells on class change- and fault-proneness

HEURISTICS


Temporal precedence
Confounding
Planning for analyses
• Establishing duration of follow-up period
• Establishing start of follow-up
• Handling units that show the outcome

Establishing Duration of Follow-Up Period

Explanation

- Time interval during which study units are monitored to measure the response variables
- Depends on the estimated treatment induction time (period between treatment initiation and the observable onset of its effects)
- Must be consistent with the response variable measured for all units

Lung cancer example

- Follow-up: 20–40 years (20–30-year induction time)
- Study duration: identical for all units

OWASP DC example

- Follow-up: 264 days (several minutes induction time)
- Study duration: identical for all units

Guideline

A consistent follow-up period based on the expected treatment induction time must be defined. If too short, causal effects may be missed; if too long, resources may be wasted without improving causal inference

Establishing Start of Follow-Up (Time Zero)

Explanation

- Time zero must always be set to ensure temporal precedence
- Exposed: time zero typically aligns with treatment initiation, but may occur later and vary across units
- Controls: time zero is more challenging without treatment initiation; it can be randomly selected or aligned with a specific exposed unit

Lung cancer example

- Option 1:
  - Exposed: start of smoking
  - Controls: randomly selected within the window/aligned with an exposed unit
- Option 2: same calendar date for all units and record exposure status

OWASP DC example

- Exposed: immediately before OWASP DC adoption
- Controls: randomly selected within the window or aligned with the commit preceding adoption in an exposed project

Guideline

Time zero must be defined to ensure temporal precedence between exposure and outcome. It is typically aligned with treatment initiation and may vary across units within a defined window. Control groups require specific strategies for it

Handling Units that Show the Outcome

Explanation

- Discrete outcome: exclude units with the outcome at time zero, as they are no longer at risk of developing it
- Continuous outcome: units need not be excluded; account for the initial value through change from baseline or as a confounder

Lung cancer example

- Discrete outcome
- Participants with disease at time zero must be excluded

OWASP DC example

- Existing vulnerabilities need not be excluded, but baseline vulnerabilities must be accounted for

Guideline

Units with outcomes already present at time zero must be excluded to ensure temporality. For continuous outcomes, the baseline value should be accounted for as outcome change or as a confounder

Code Smells Example

Research questions

- RQ1. To what extent do code smells cause class change-proneness?
- RQ2. To what extent do code smells cause class fault-proneness?

Dataset

- F, Khomh, M. Di Penta, Y. G. Guéhéneuc, G. Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Engineering 17 (2012), 243–275

Variables

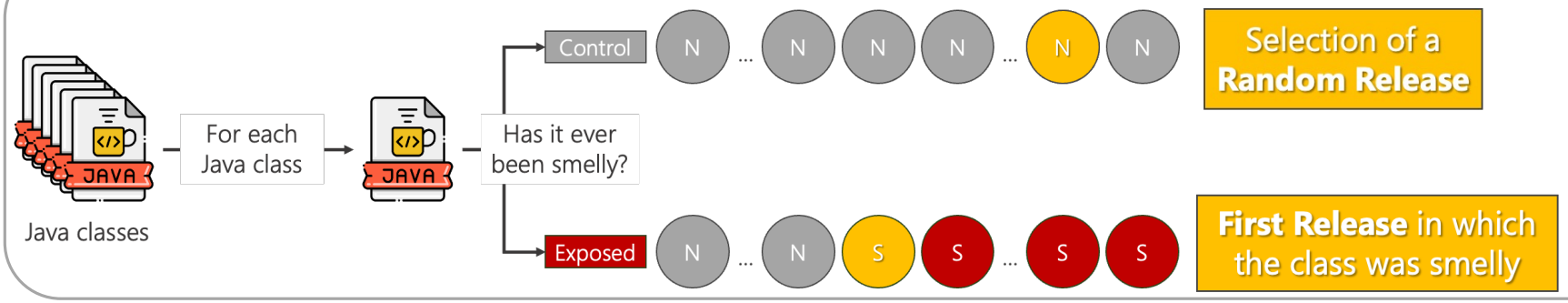
- Independent: Smelly (+11 more, one per each type of code smell)
- Dependent: Fault proneness, change proneness

Code Smells Example

Duration of follow-up period

- One release (as in the Khomh et al. study)

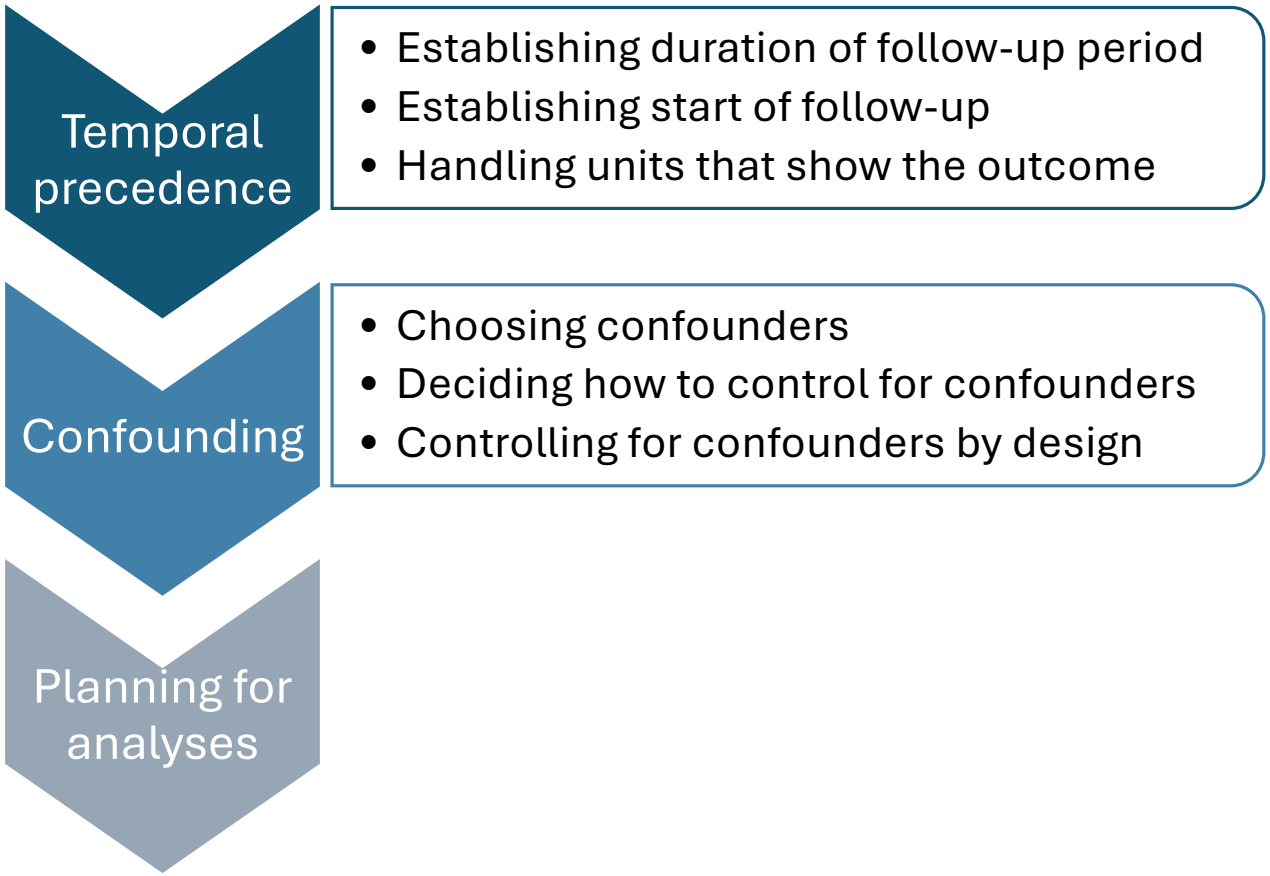
Establishing time-zero



Handling units that show outcome

- Not necessary

HEURISTICS


Temporal precedence
• Establishing duration of follow-up period
• Establishing start of follow-up
• Handling units that show the outcome
Confounding
• Choosing confounders
• Deciding how to control for confounders
• Controlling for confounders by design
Planning for analyses

Choosing Confounders

Explanation

- Confounders can be identified through: subject-matter knowledge or statistical methods
- Subject-matter knowledge rely on explicit knowledge of causal structure
- **Disjunctive cause**: control variables that influence the exposure, outcome, or both, based on causal knowledge and literature

Lung cancer example

- Smoking exposure: gender, socioeconomic status
- Lung cancer risk: age, gender

OWASP DC example

- Prior work suggests:
  - Confounders: project age, #components
  - No confounders: project activity level, popularity, developer experience

Guideline

A cohort study should identify and control for confounders influencing exposure, outcome, or both, based on the literature. Confounders should be measured for all units at baseline

Deciding How to Control for Confounders

Explanation

- Ensure that the control and exposed groups are comparable with respect to the confounders
- Analysis-based control: using statistical models (e.g., regression, ANCOVA, logistic models)
- Limitations: imbalance or partial overlap
- When present: prefer design-level approaches to control confounders

Lung cancer example

- Control by design: sex, age
- Control by analysis: socioeconomic status

OWASP DC example

- Control by design: age, #components
- Control by analysis: possible residual confounding

Guideline

Balance and overlap between treatment and control groups should be assessed. When imbalance or insufficient overlap, design-level approaches to control for confounders should be preferred to avoid extrapolation and preserve causal validity

Controlling for Confounders by Design

Explanation

- Create comparable groups by pruning units
- Restriction: select units with similar confounder values—limits generalizability
- Matching: pairs units to balance observed confounders
- Matching strategy: test different approaches and select the one best suited to the data (assessing balance/sample size)
- Statistical packages available

Lung cancer example

- Restrict by age
- Match by sex

OWASP DC example

- Restriction: projects with no components or newly created
- Matching: address residual confounding

Guideline

Confounders should be controlled by design using restriction and/or matching. The choice should consider study goals, balance and overlap between groups, and sample size

Code Smells Example

Choosing confounders

- Literature review:
- Correlated to the effort required to fix defects and reduce overall software maintainability:
  - Depth of Inheritance Tree (DIT)
  - Number of Children (NOC)
- Correlated with different types of code smells
  - Number of Methods (NOM)
  - Response for a Class (RFC)
  - Weighted Methods per Class (WMC)
  - Lack of Cohesion of Methods (LCOM)
- Correlated with difficulty to perform changes
  - Lines of Code (LOC)

Code Smells Example

Balance & overlap

- Lack of balance: differences in confounder distributions between groups can threaten validity
- Partial overlap: non-overlapping conf
  extrapolation

All confounders have issues

Assessm

- Stan
  - Me
    between group means
  - SMD < 0.20 indicates good balance
- Kolmogorov-Smirnov (KS) test:
  - Compares distributions
  - p > 0.05 indicates similar distributions

Code Smells Example

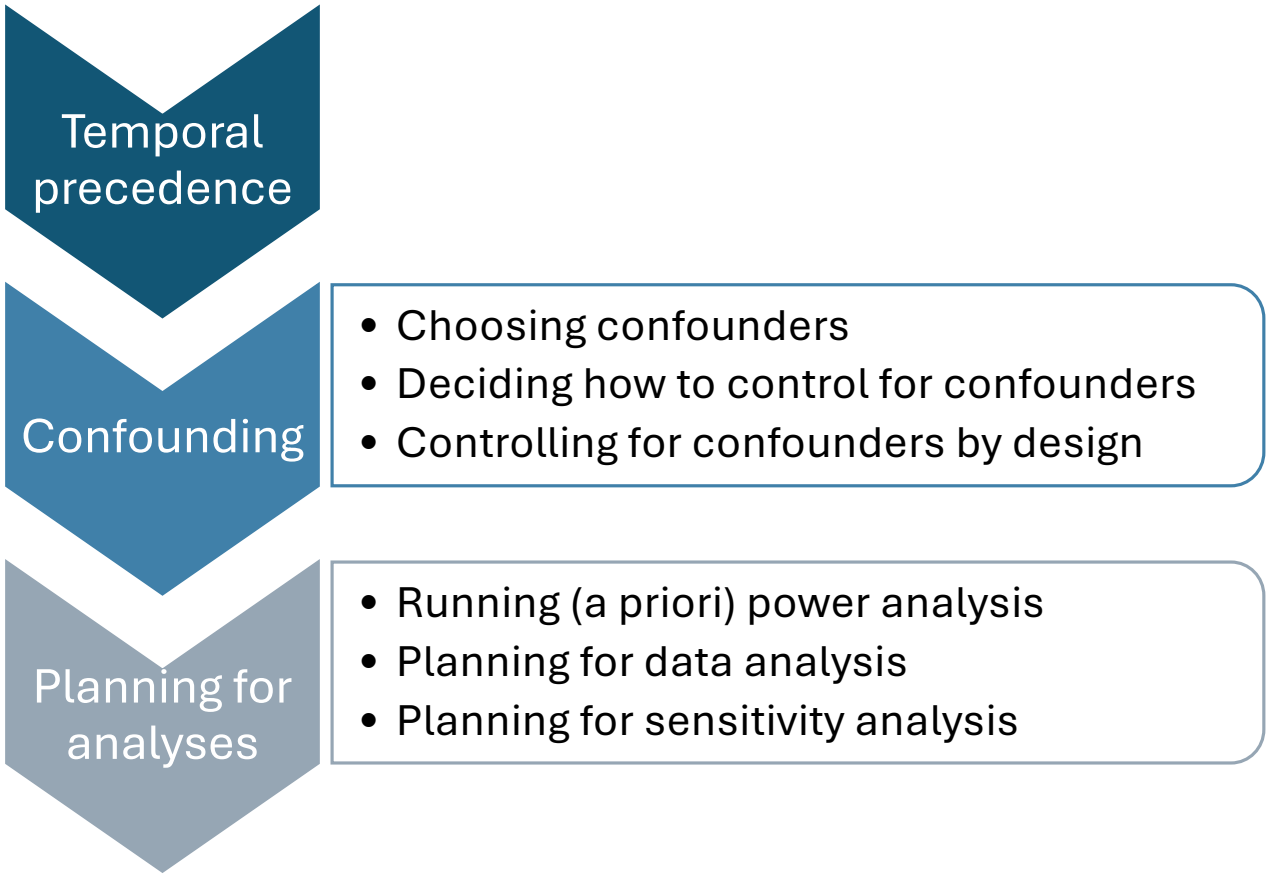
Deciding how to control for confounders

- Control by design for all confounders
- Control by analysis for possible residual confounding

Controlling for confounders by design

- Restriction: excluding Java objects that do not have type «class»
- Matching

HEURISTICS


Temporal precedence
Confounding
• Choosing confounders
• Deciding how to control for confounders
• Controlling for confounders by design
Planning for analyses
• Running (a priori) power analysis
• Planning for data analysis
• Planning for sensitivity analysis

Running (A Priori) Power Analysis

Explanation

- Determines the minimum sample size needed to detect significant effects
- Conducted before data analysis to avoid underpowered studies
- If the required sample size cannot be achieved, revise unit selection, restriction and/or matching

Lung cancer example

- Helps determine the required number of participants based on the effect size that the researchers intend to detect

OWASP DC example

- Helps determine the required number of projects based on the effect size that the researchers intend to detect

Guideline

An a priori power analysis should estimate the sample size required for valid inference, as restriction and matching affect sample size and statistical power. If insufficient, the unit selection strategy and design approaches should be revised

Planning for Data Analysis

Explanation

- Unadjusted analysis: quantify potential confounding bias
- Adjusted analysis: causal analysis (performed after matching/restriction
- Matched confounders: excluded from the model to avoid overadjustment
- Include them if: residual imbalance, complex matching, or potential interactions exist

Lung cancer example

- The statistical analysis will include:
  - Socioeconomic status
  - But not age or sex, unless one of the mentioned circumstances arises

OWASP DC example

- The statistical analysis will include:
  - Change in the number of vulnerabilities between the start and end of follow-up
  - But not age or #components, unless one of the mentioned circumstances arises

Guideline

An adjusted analysis should be planned after matching or restriction, incorporating confounders when needed and serving as basis for causal inference. An unadjusted analysis assesses confounding bias, and both results should be triangulated

Planning for Sensitivity Analysis

Explanation

- Observational studies cannot fully rule out residual confounding
- Sensitivity analysis assesses how robust causal estimates are to unmeasured confounders
- Known confounders: adjust estimates using prior information
- Unknown confounders: use measures E-value or robustness value to assess strength needed to nullify the effect

Lung cancer example

- Sensitivity analysis should be planned based on the availability of information about unmeasured confounders

OWASP DC example

- Sensitivity analysis should be planned based on the availability of information about unmeasured confounders

Guideline

Sensitivity analyses assess the robustness of causal relationships to unmeasured confounding. They do not establish its existence, but evaluate how strongly it would need to affect results to alter or nullify the observed effect

Code Smells Example

Running (a priory) power analysis

- No residual confounders:
  - Required sample sizes: 824 (small), 134 (medium), and 54 (large effects)
  - Confounder-adjusted sample size is sufficient to detect **medium effects** across all variables, and **potentially small effects**
  - Lazy Class: sufficient to detect small effects
- Residual confounders:
  - Required sample sizes: 395 (small), 55 (medium), and 25 (large effects)
  - Large Class: insufficient sample size to detect even large effects; limited units were already available before matching/restriction
  - Complex Class & SG: sufficient for medium and potentially small effects
  - Smelly, Long Method & RPB: sufficient for small effects

CODE SMELLS EXAMPLE

Planning for data analysis

Dependent Variable	Independent Variable	Unadjusted		Adjusted	
		p-value	Effect Size	p-value	Effect Size
Change-proneness	Smelly	**3.7E-76** ↑	*Small*	**1E-10** ↑	*Small*
	Anti-Singleton	**1.1E-18** ↑	*Small*	**5.1E-07** ↑	*Small*
	Blob	**1.7E-04** ↑	*Small*	0.388 ↑	Negligible
	CDSBP	**1.4E-06** ↑	*Small*	**0.031** ↑	*Small*
	Complex Class	0.065 ↑	Negligible	0.987 ↑	Negligible
	Large Class	**4E-08** ↑	*Large*	**0.009** ↑	*Large*
	Lazy Class	**4.3E-12** ↑	Negligible	0.176 ↑	Negligible
	Long Method	**2.4E-66** ↑	*Small*	**0.002** ↑	*Small*
	LPL	**1.2E-13** ↑	*Small*	**2E-05** ↑	*Small*
	Message Chains	**3.1E-14** ↑	*Small*	**8.9E-07** ↑	*Small*
	RPB	**4.8E-25** ↑	*Small*	**2.1E-14** ↑	*Small*
	SG	**0.017** ↑	*Small*	0.381 ↑	Negligible
Fault-proneness	Smelly	**2.9E-20** ↑	Negligible	**0.039** ↑	Negligible
	Anti-Singleton	**0.022** ↑	Negligible	0.332 ↑	Negligible
	Blob	0.285 ↑	Negligible	0.472 ↓	Negligible
	CDSBP	0.708 ↓	Negligible	0.472 ↓	Negligible
	Complex Class	0.926 ↓	Negligible	0.522 ↓	Negligible
	Large Class	**0.008** ↑	*Small*	0.522 ↓	*Small*
	Lazy Class	0.172 ↑	Negligible	0.51 ↑	Negligible
	Long Method	**2.7E-15** ↑	Negligible	0.552 ↓	Negligible
	LPL	**5.5E-07** ↑	Negligible	**0.002** ↑	Negligible
	Message Chains	**1E-12** ↑	Negligible	**1.8E-04** ↑	Negligible
	RPB	**3.5E-09** ↑	Negligible	**2E-05** ↑	*Small*
	SG	**0.03** ↑	Negligible	0.3 ↑	*Small*

Code Smells Example

Planning for sensitivity analysis

Dependent Variable	Independent Variable	Confounder Strength (× LOC)
Change-proneness	Smelly	20.551
	Anti-Singleton	509.588
	CDSBP	1.191
	Large Class	8.249
	Long Method	135.789
	LPL	14.158
	Message Chains	5,381.215
	RPB	108.100
Fault-proneness	Smelly	30.852
	LPL	9.777
	Message Chains	2,149.906
	RPB	44.229

CONCLUSIONS

- The premise of mining software repositories research is to enable data-driven insights to support software engineers
- Much of the research conducted in the past remains primarily descriptive or correlational, offering limited guidance for actionable improvement
- Advancing causal understanding within this type of study is therefore essential to unveil and explain the mechanisms that drive observed phenomena
- We propose the instantiation of the cohort study design to mining software repository studies
- We have showed how it can be applied to investigate causation