50% SEMINAR

AUTOMATED SOFTWARE DEBLOAT

Speaker: **César Soto Valero**
Supervisor: **Benoit Baudry**
Co-supervisors: **Martin Monperrus, Thomas Durieux**

AGENDA

AGENDA

1. INTRODUCTION AND STATE-OF-THE-ART

AGENDA

1. **INTRODUCTION AND STATE-OF-THE-ART**
2. **CONTRIBUTIONS**
   I. Detecting and removing bloated dependencies
   II. Longitudinal analysis of bloated dependencies
   III. Trace-based debloat for Java bytecode

AGENDA

1. **INTRODUCTION AND STATE-OF-THE-ART**
2. **CONTRIBUTIONS**
   I. Detecting and removing bloated dependencies
   II. Longitudinal analysis of bloated dependencies
   III. Trace-based debloat for Java bytecode
3. **SUMMARY AND FUTURE WORK**

AGENDA

1. **INTRODUCTION AND STATE-OF-THE-ART**
2. **CONTRIBUTIONS**
   I. Detecting and removing bloated dependencies
   II. Longitudinal analysis of bloated dependencies
   III. Trace-based debloat for Java bytecode
3. **SUMMARY AND FUTURE WORK**
4. **PHD PROGRESS**

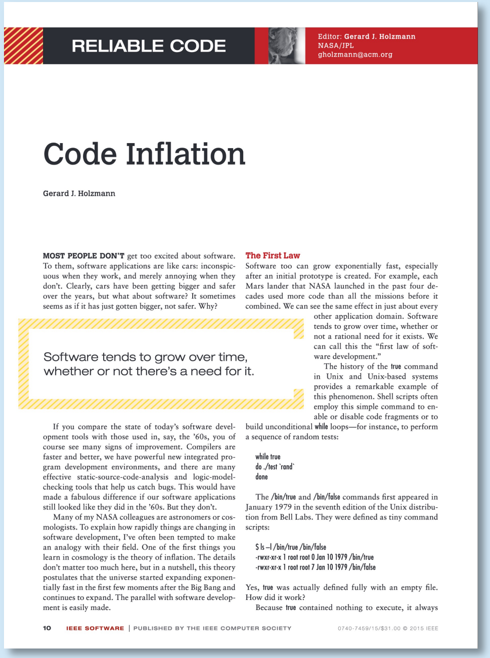
AGENDA

1. **INTRODUCTION AND STATE-OF-THE-ART**
2. **CONTRIBUTIONS**
   I. Detecting and removing bloated dependencies
   II. Longitudinal analysis of bloated dependencies
   III. Trace-based debloat for Java bytecode
3. **SUMMARY AND FUTURE WORK**
4. **PHD PROGRESS**
5. **Q&A**

“

Software tends to grow over time, whether or not there's a need for it.

Holzmann, G. J. (2015). Code inflation. IEEE Software, 32 (2).



RELIABLE CODE

Editor: **Gerard J. Holzmann**
NASA/JPL
gholzmann@acm.org

Code Inflation

Gerard J. Holzmann

MOST PEOPLE DON'T get too excited about software. To them, software applications are like cars: inconspicuous when they work, and merely annoying when they don't. Clearly, cars have been getting bigger and safer over the years, but what about software? It sometimes seems as if it has just gotten bigger, not safer. Why?

> Software tends to grow over time, whether or not there's a need for it.

If you compare the state of today's software development tools with those used in, say, the '60s, you of course see many signs of improvement. Compilers are faster and better, we have powerful new integrated program development environments, and there are many effective static-source-code-analysis and logic-model-checking tools that help us catch bugs. This would have made a fabulous difference if our software applications still looked like they did in the '60s. But they don't.

Many of my NASA colleagues are astronomers or cosmologists. To explain how rapidly things are changing in software development, I've often been tempted to make an analogy with their field. One of the first things you learn in cosmology is the theory of inflation. The details don't matter too much here, but in a nutshell, this theory postulates that the universe started expanding exponentially fast in the first few moments after the Big Bang and continues to expand. The parallel with software development is easily made.

The First Law

Software too can grow exponentially fast, especially after an initial prototype is created. For example, each Mars lander that NASA launched in the past four decades used more code than all the missions before it combined. We can see the same effect in just about every other application domain. Software tends to grow over time, whether or not a rational need for it exists. We can call this the "first law of software development."

The history of the true command in Unix and Unix-based systems provides a remarkable example of this phenomenon. Shell scripts often employ this simple command to enable or disable code fragments or to build unconditional while loops—for instance, to perform a sequence of random tests:

```
while true
do ./test `rand`
done
```

The /bin/true and /bin/false commands first appeared in January 1979 in the seventh edition of the Unix distribution from Bell Labs. They were defined as tiny command scripts:

```
$ ls –l /bin/true /bin/false
-rwxr-xr-x 1 root root 0 Jan 10 1979 /bin/true
-rwxr-xr-x 1 root root 7 Jan 10 1979 /bin/false
```

Yes, true was actually defined fully with an empty file. How did it work?

Because true contained nothing to execute, it always

Software tends to grow over time, whether or not there's a need for it.

Holzmann, G. J. (2015). Code inflation. IEEE Software, 32 (2).

THE HISTORY OF THE `true` COMMAND

1979

```
$ ls -l /bin/true
-rwxr-xr-x 1 root root 0 Jan 10 1979 /bin/true
```

Holzmann, G. J. (2015). Code inflation. IEEE Software, 32 (2).

THE HISTORY OF THE `true` COMMAND

1984

```
$ ls -l /bin/true
-rwxr-xr-x 1 root root 276 May 14 1984 /bin/true
```

Holzmann, G. J. (2015). Code inflation. IEEE Software, 32 (2).

THE HISTORY OF THE `true` COMMAND

2010

```
$ ls -l /bin/true
-rwxr-xr-x 1 root root 8377 Sep 10 2010 /bin/true
```

Holzmann, G. J. (2015). Code inflation. IEEE Software, 32 (2).

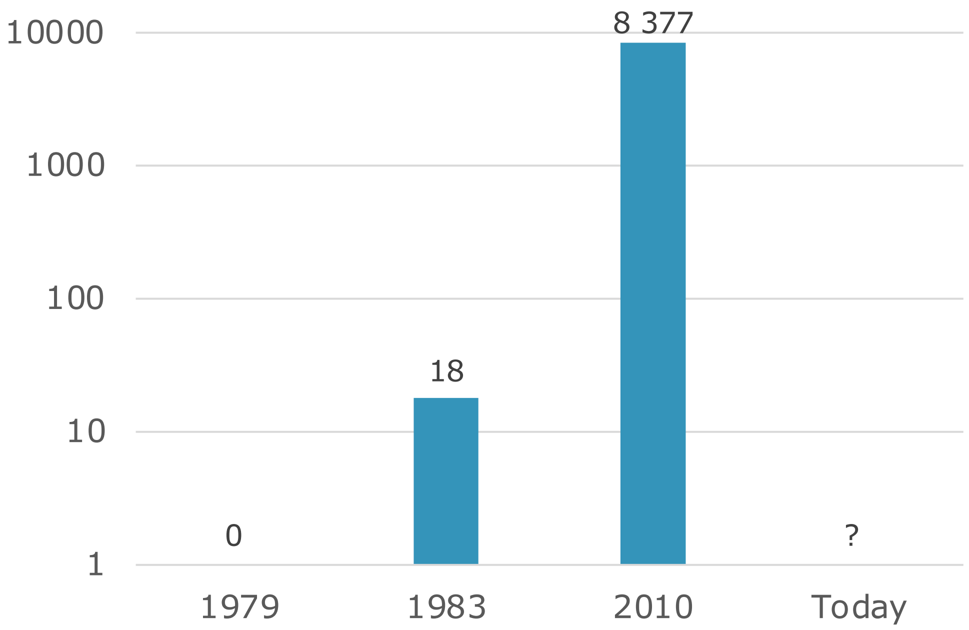
THE HISTORY OF THE `true` COMMAND

TODAY

```
$ type true
true is a shell builtin
```

Holzmann, G. J. (2015). Code inflation. IEEE Software, 32 (2).

Holzmann, G. J. (2015). Code inflation. IEEE Software, 32 (2).

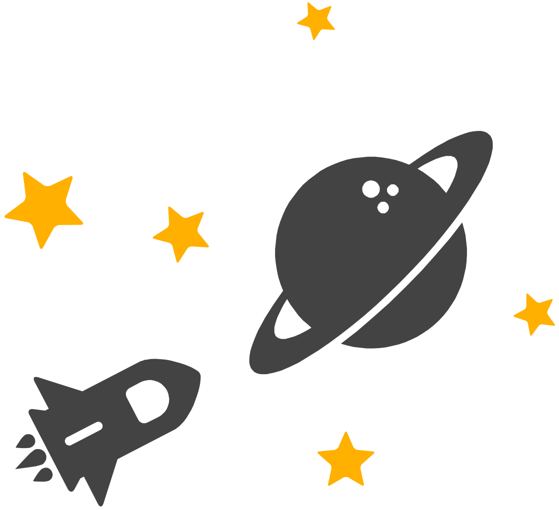


Size (in bytes) of the `true` command

SOFTWARE BLOAT

Code that is packaged in an application but that is not necessary for building and running the application.

IT IS A PROBLEM



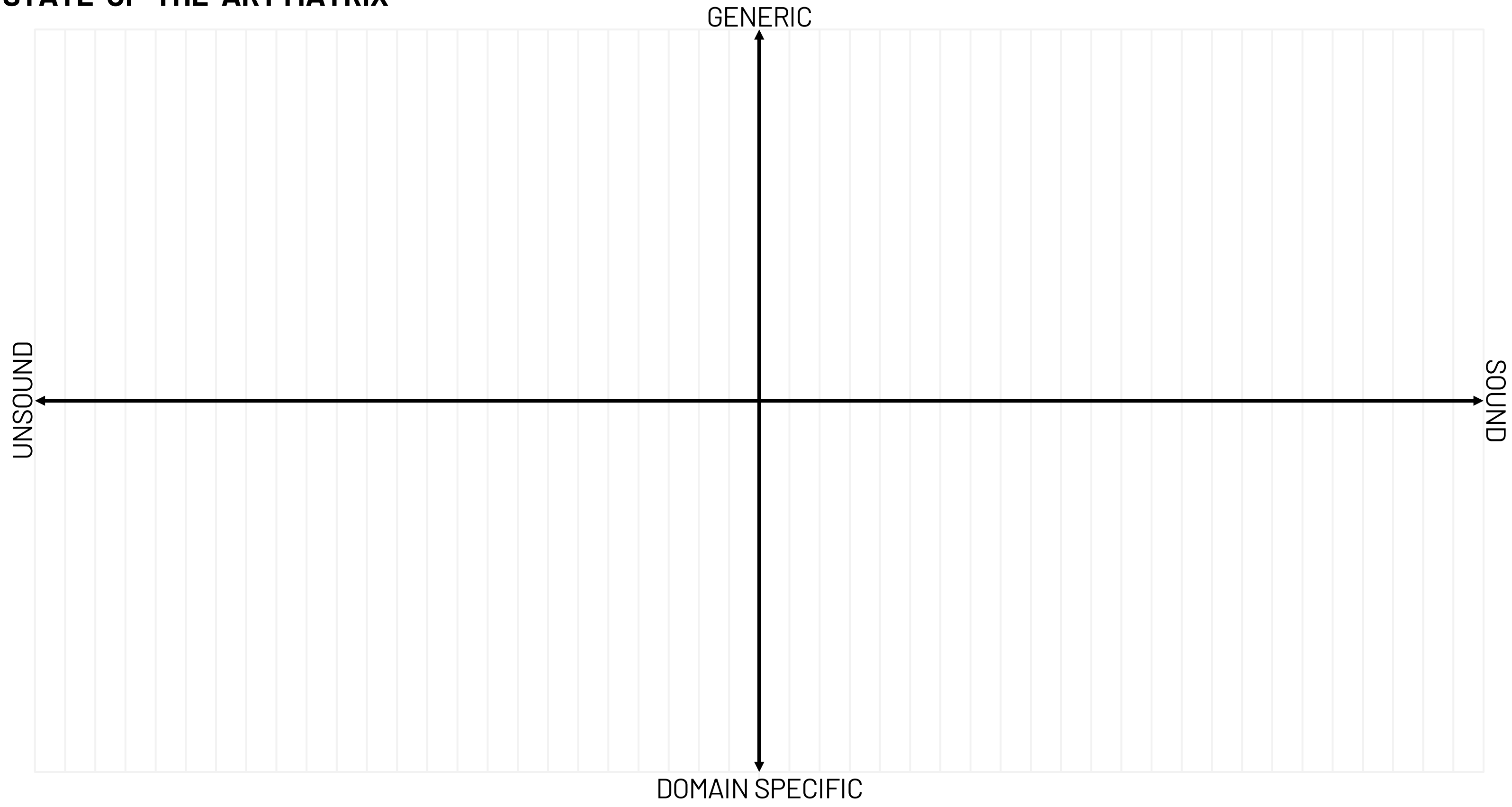
For size

For security

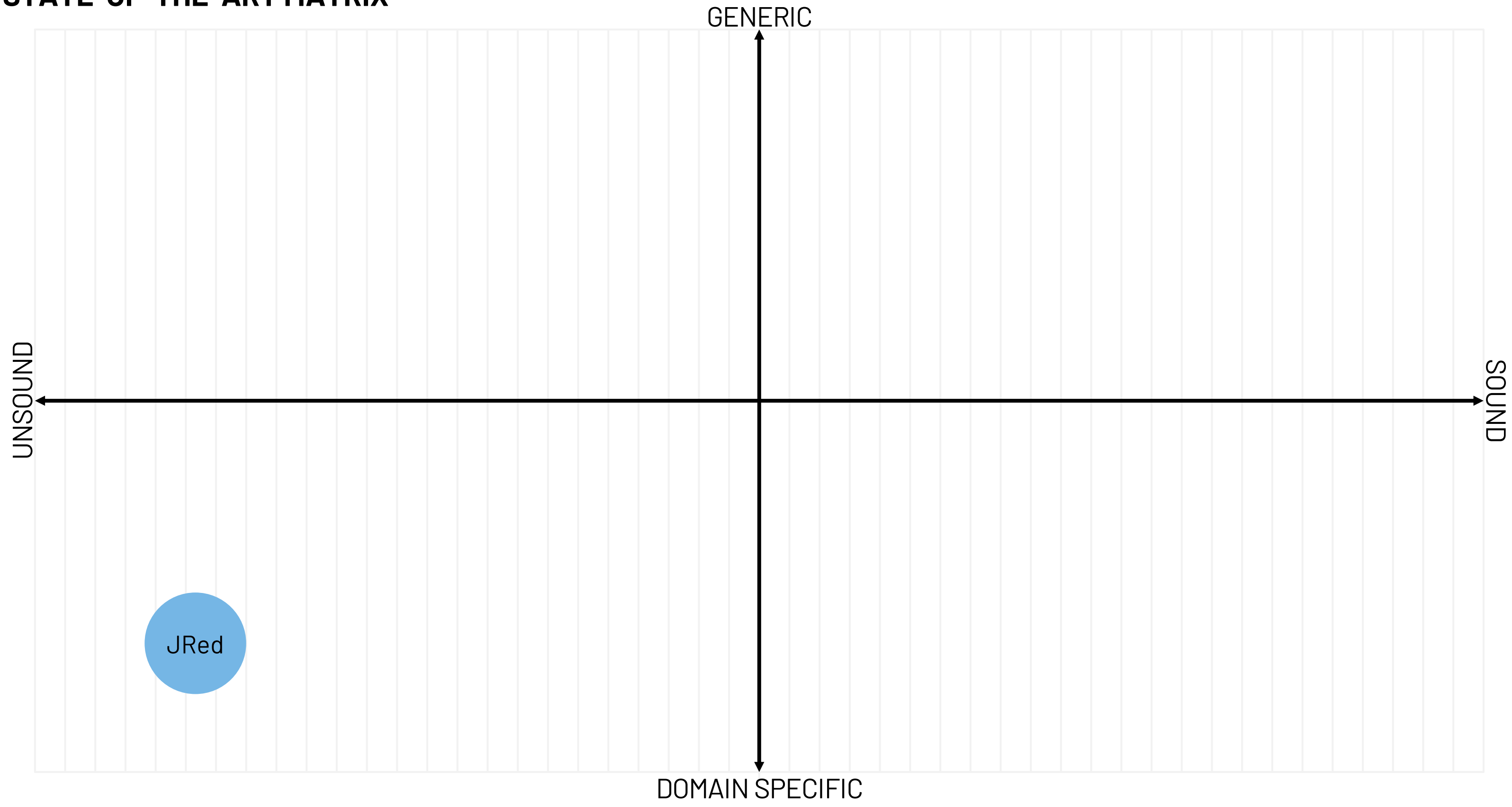
For maintenance

For performance

STATE-OF-THE-ART MATRIX

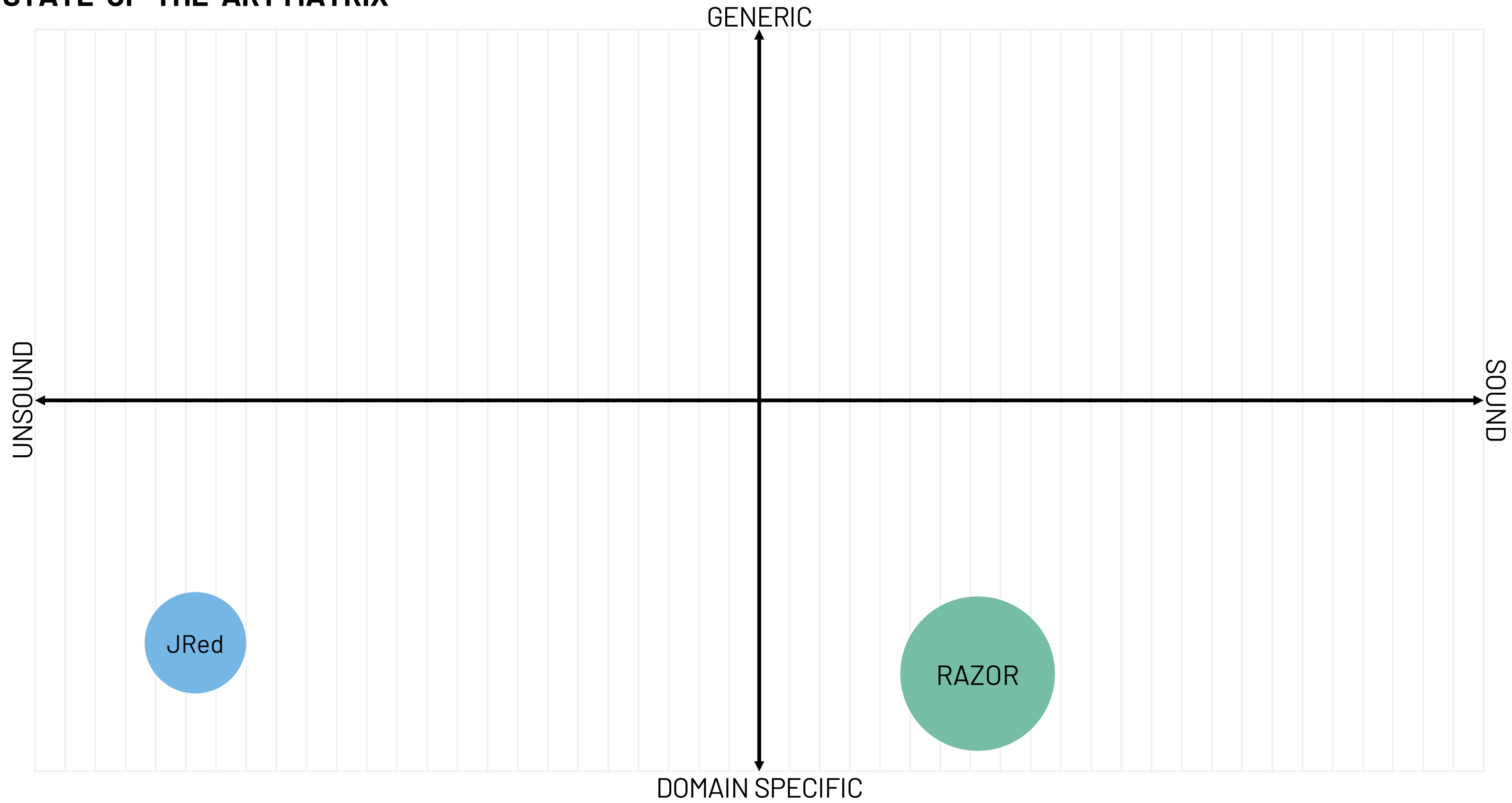


STATE-OF-THE-ART MATRIX



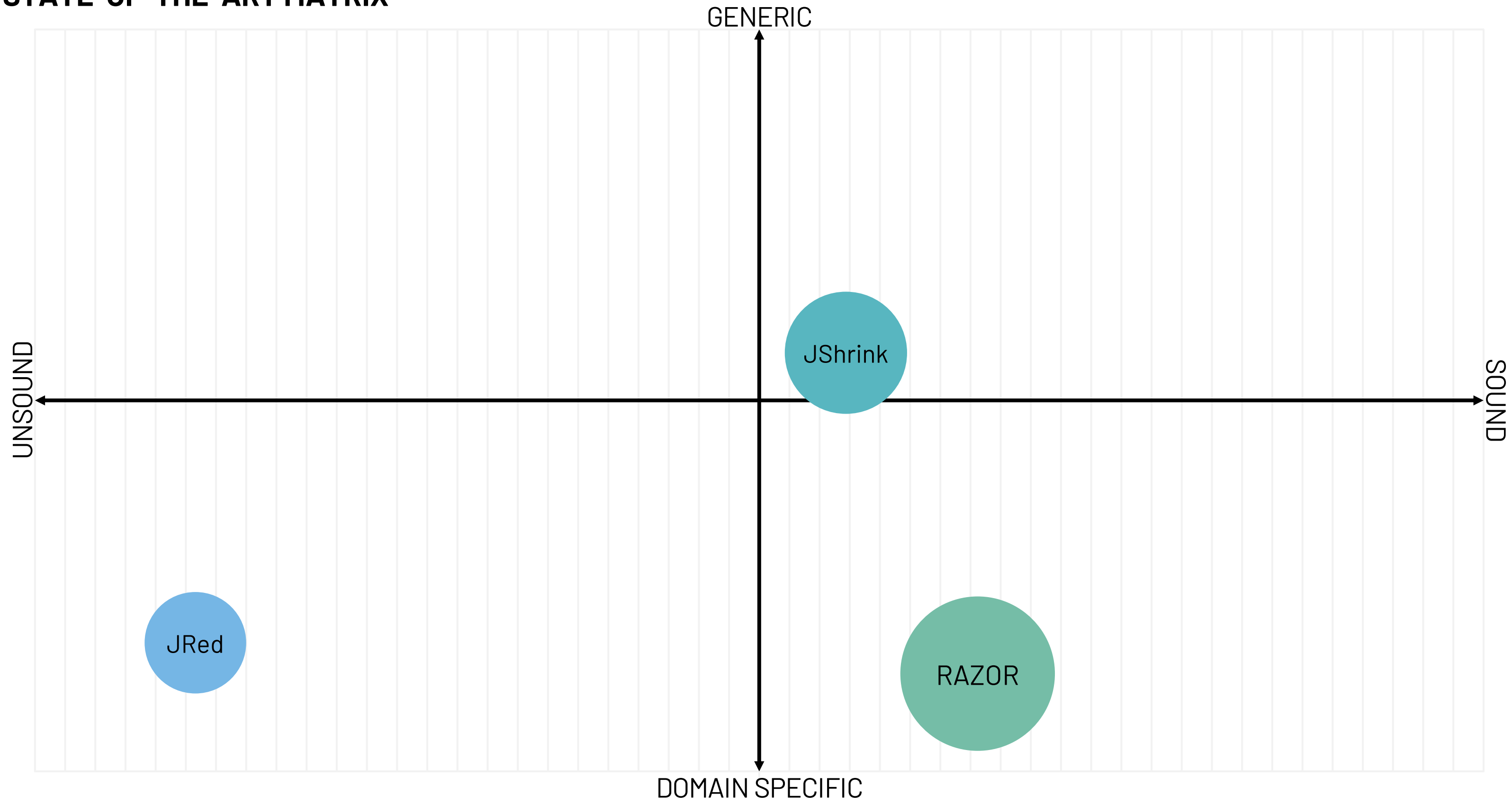
Jiang et. al., JRed: Program Customization and Bloatware Mitigation [COMSAC'16]

STATE-OF-THE-ART MATRIX



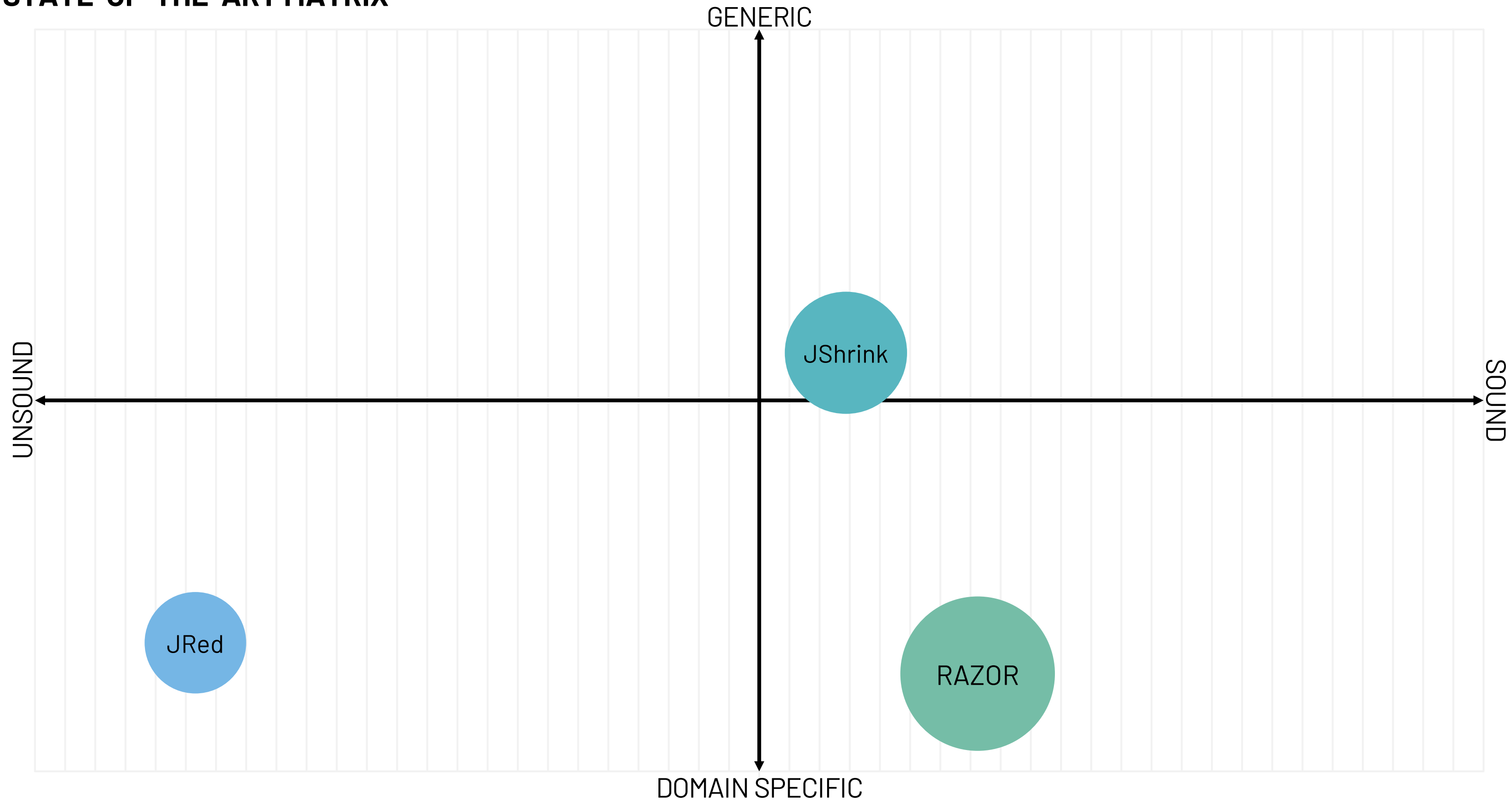
Quian et. al., RAZOR: A Framework for Post-deployment Software Debloating [USENIX'19]

STATE-OF-THE-ART MATRIX



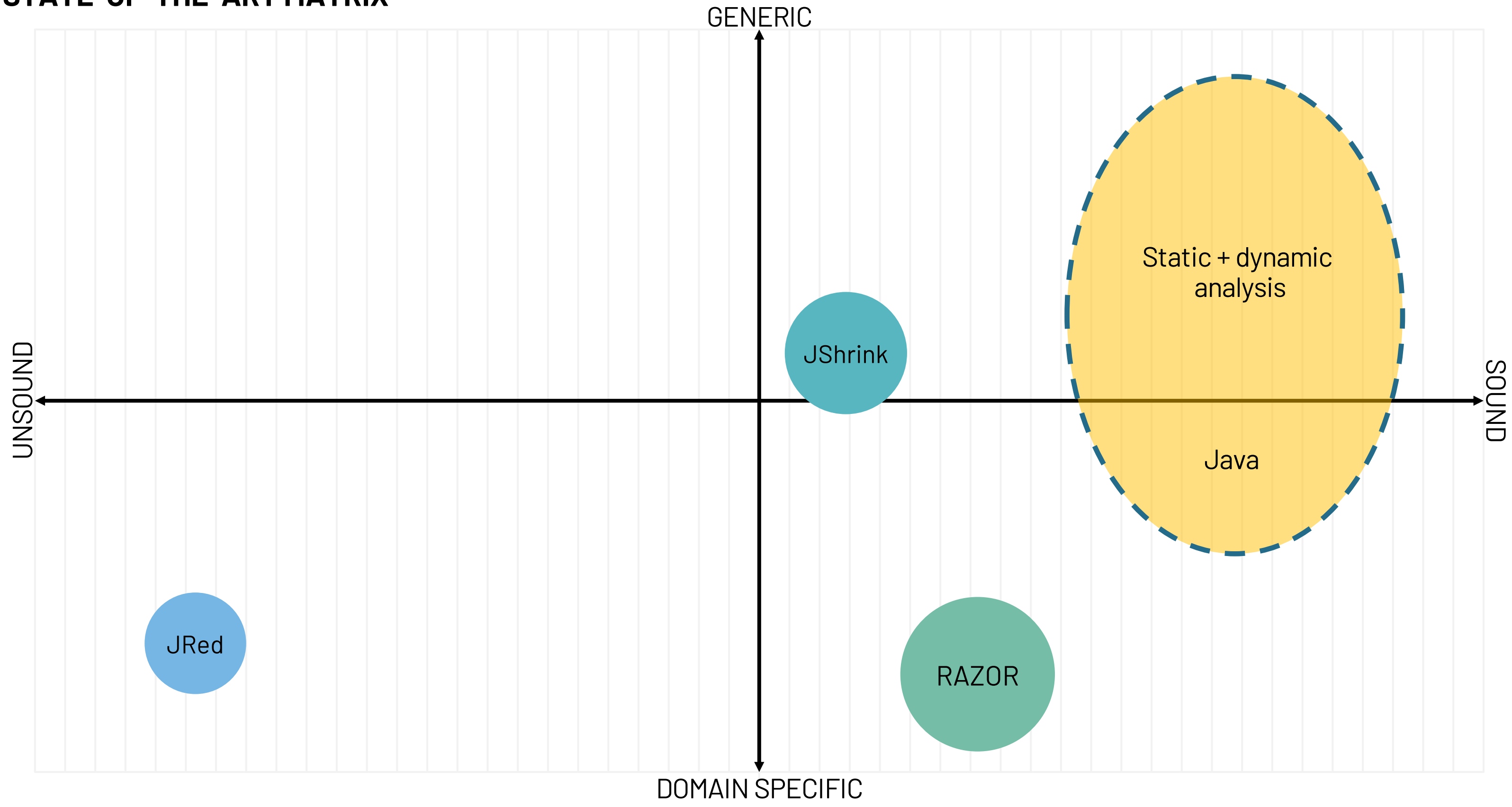
Bruce et. al., JShrink: In-depth Investigation into Debloating Modern Java Applications [FSE'20]

STATE-OF-THE-ART MATRIX



THIS WORK!

STATE-OF-THE-ART MATRIX



THIS WORK!

Empirical Software Engineering (2021) 26:45
https://doi.org/10.1007/s10664-020-09914-8

Check for updates

A comprehensive study of bloated dependencies in the Maven ecosystem

César Soto-Valero[1] · Nicolas Harrand[1] · Martin Monperrus[1] · Benoit Baudry[1]

Accepted: 23 September 2020 Published online: 25 March 2021

Abstract
Build automation tools and package managers have a profound influence on software development. They facilitate the reuse of third-party libraries, support a clear separation between the application's code and its external dependencies, and automate several software development tasks. However, the wide adoption of these tools introduces new challenges related to dependency management. In this paper, we propose an original study of one such challenge: the emergence of bloated dependencies. Bloated dependencies are libraries that are packaged with the application's compiled code but that are actually not necessary to build and run the application. They artificially grow the size of the built binary and increase maintenance effort. We propose DepClean, a tool to determine the presence of bloated dependencies in Maven artifacts. We analyze 9,639 Java artifacts hosted on Maven Central, which include a total of 723,444 dependency relationships. Our key result is as follows: 2.7% of the dependencies directly declared are bloated, 15.4% of the inherited dependencies are bloated, and 57% of the transitive dependencies of the studied artifacts are bloated. In other words, it is feasible to reduce the number of dependencies of Maven artifacts to 1/4 of its current count. Our qualitative assessment with 30 notable open-source projects indicates that developers pay attention to their dependencies when they are notified of the problem. They are willing to remove bloated dependencies: 21/26 answered pull requests were accepted and merged by developers, removing 140 dependencies in total: 75 direct and 65 transitive.

Keywords Dependency management · Software reuse · Debloating · Program analysis

1 Introduction

Software reuse, a long time advocated software engineering practice (Naur and Randell 1969; Krueger 1992), has boomed in the last years thanks to the widespread adoption of build automation and package managers (Cox 2019; Soto-Valero et al. 2019). Package managers provide both a large pool of reusable packages, a.k.a. libraries, and systematic ways to

Communicated by: Gabriele Bavota

César Soto-Valero
cesarsv@kth.se

[1] KTH Royal Institute of Technology, Stockholm, Sweden

1st CONTRIBUTION

DepClean: Automatically detecting and removing bloated dependencies in Maven projects

OVERVIEW

OVERVIEW

P

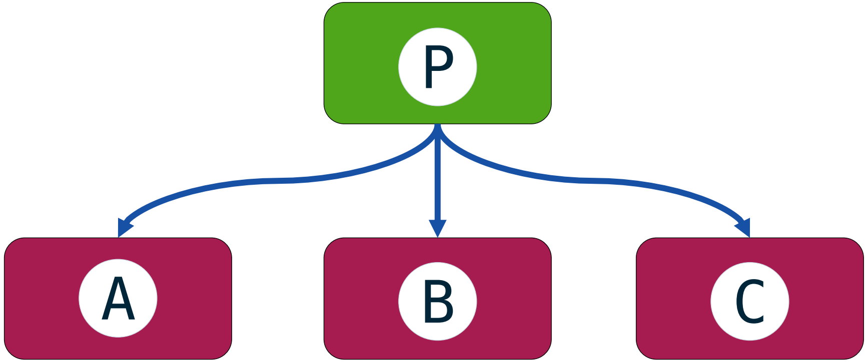
OVERVIEW


```
<dependency>
    <groupId>org.A</groupId>
    <artifactId>A</artifactId>
</dependency>
<dependency>
    <groupId>org.B</groupId>
    <artifactId>B</artifactId>
</dependency>
<dependency>
    <groupId>org.C</groupId>
    <artifactId>C</artifactId>
</dependency>
```

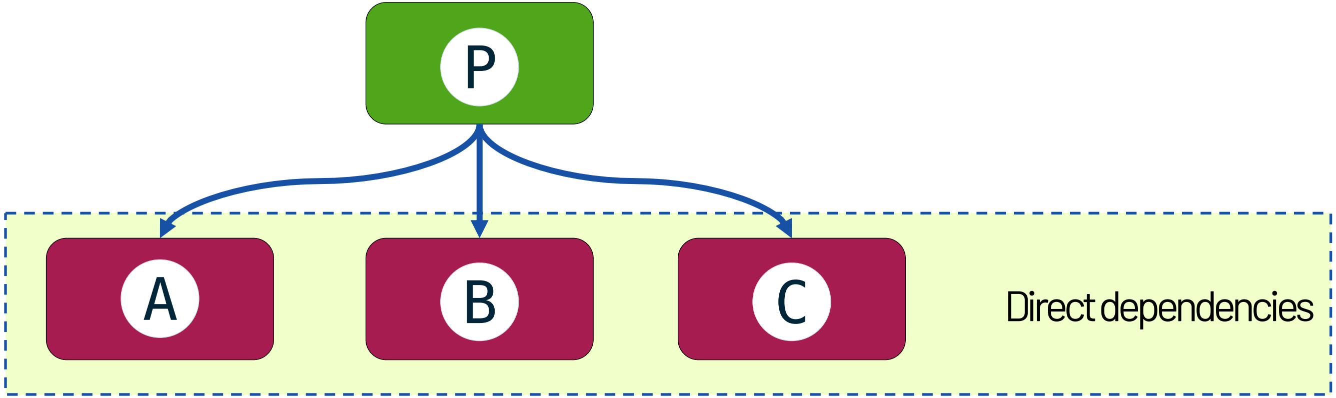
OVERVIEW

P

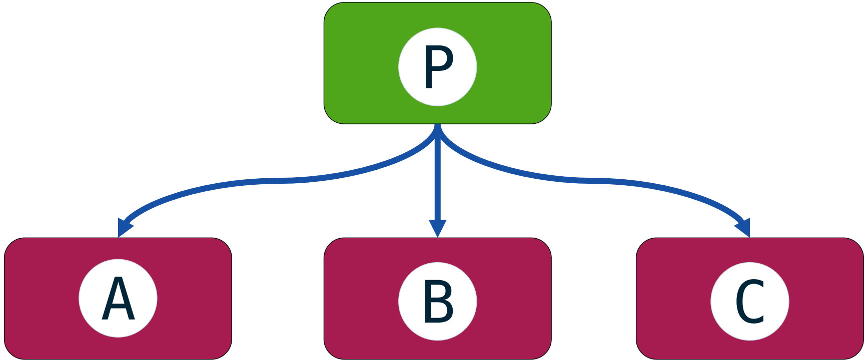
OVERVIEW



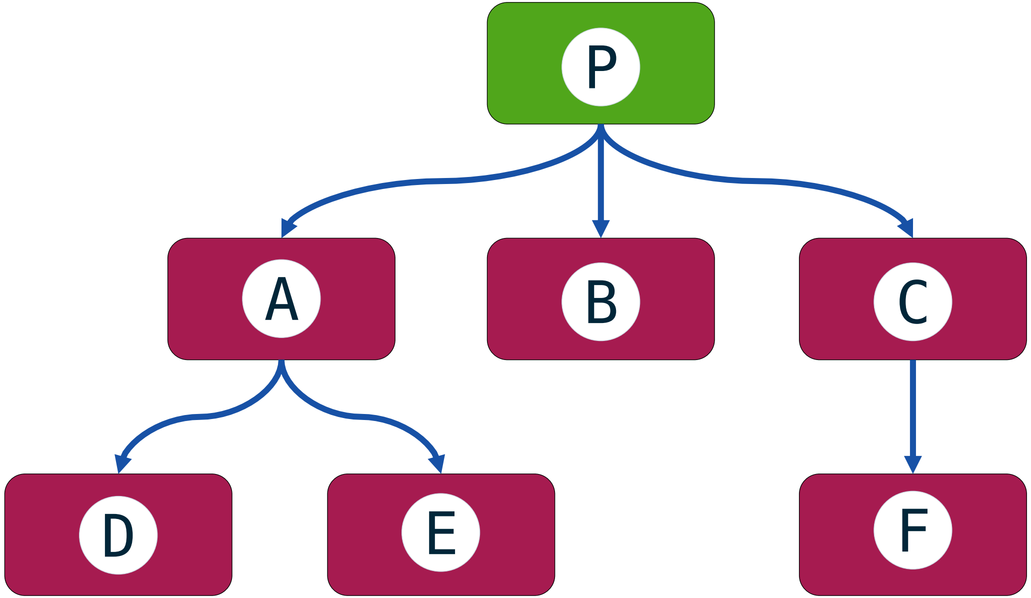
OVERVIEW



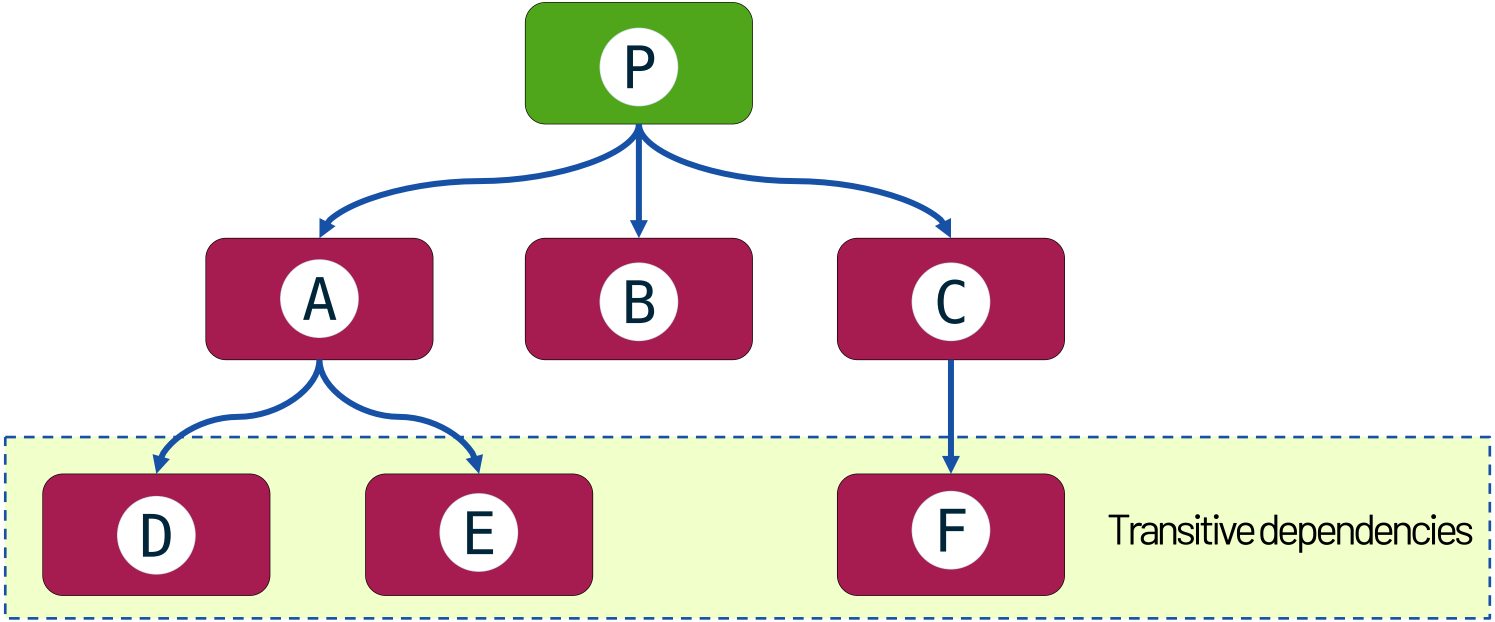
OVERVIEW



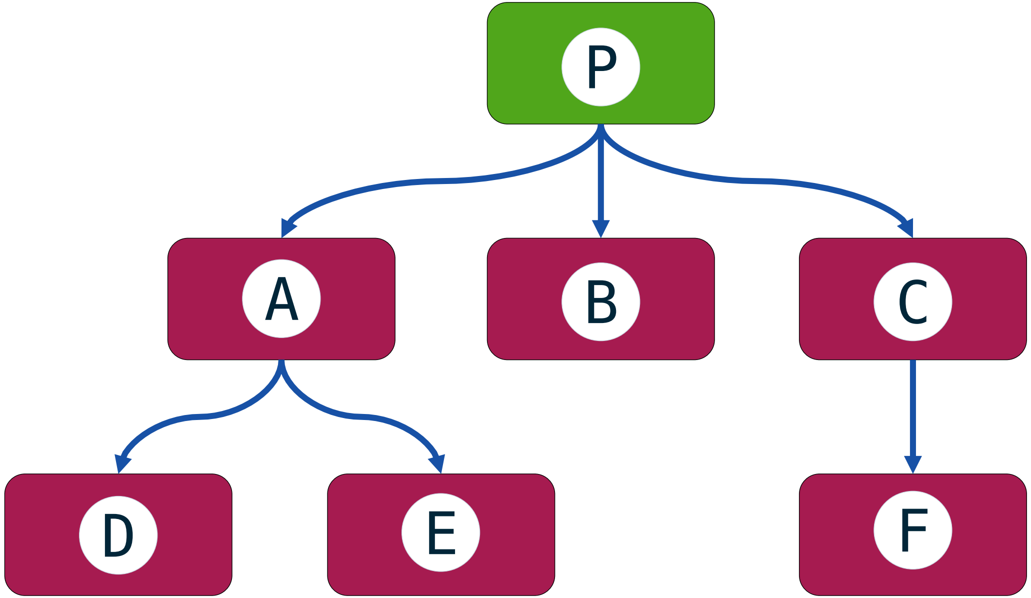
OVERVIEW



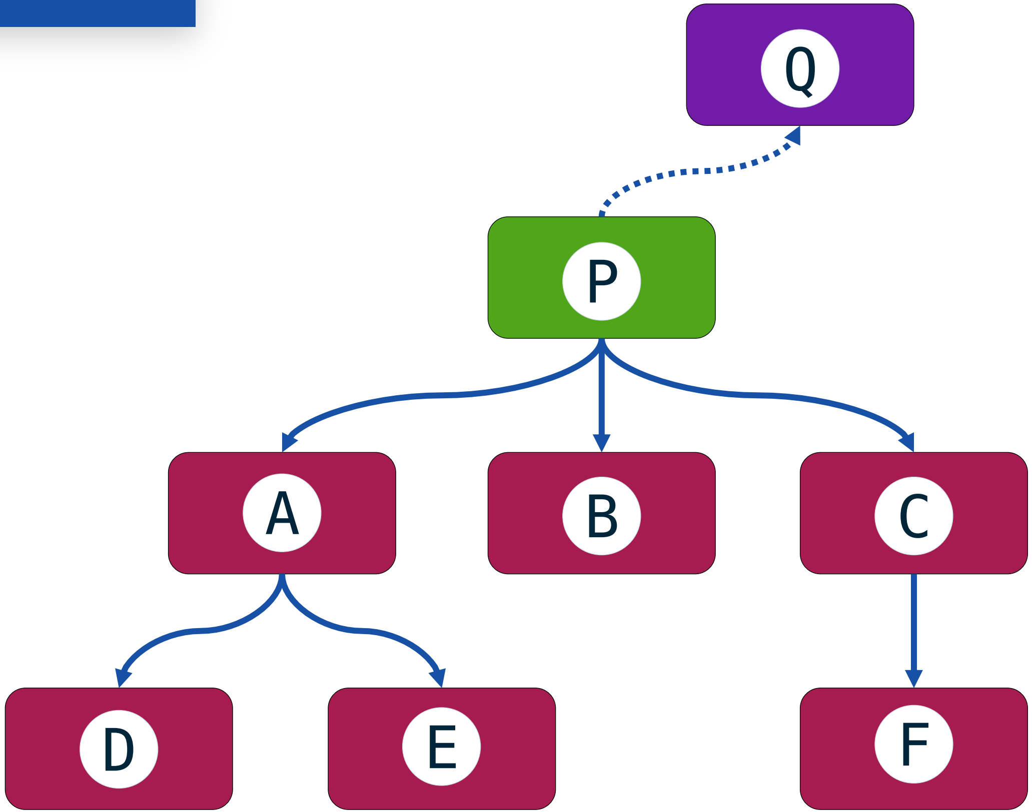
OVERVIEW



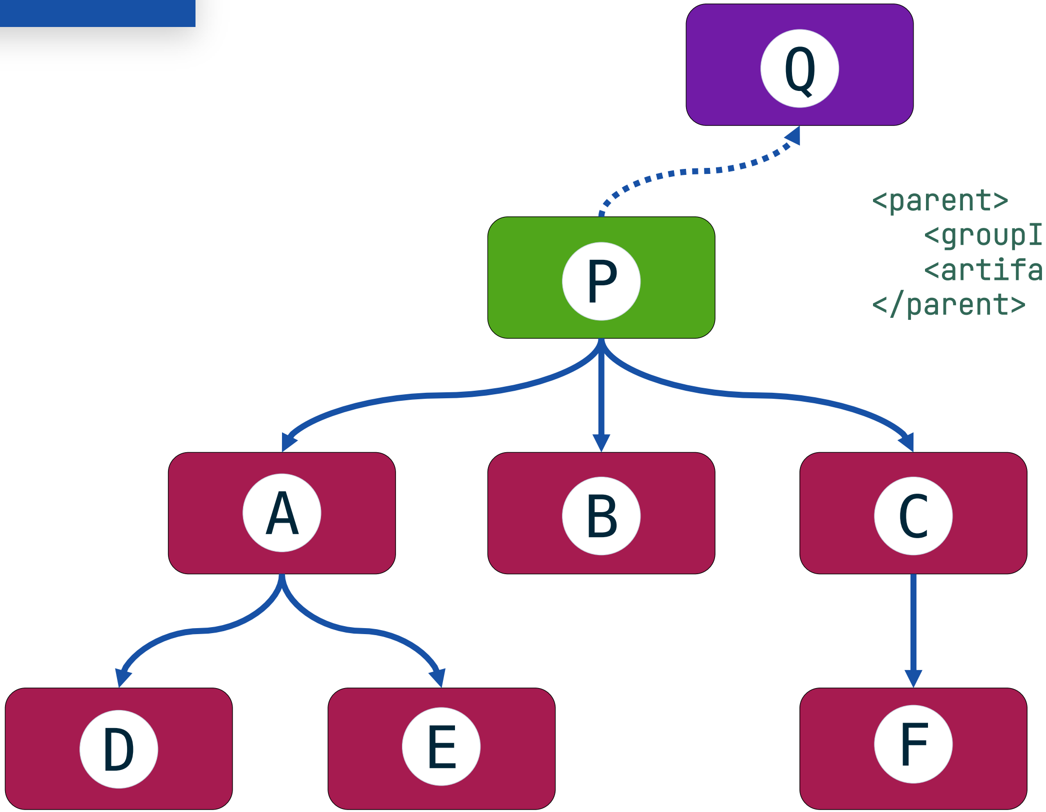
OVERVIEW



OVERVIEW

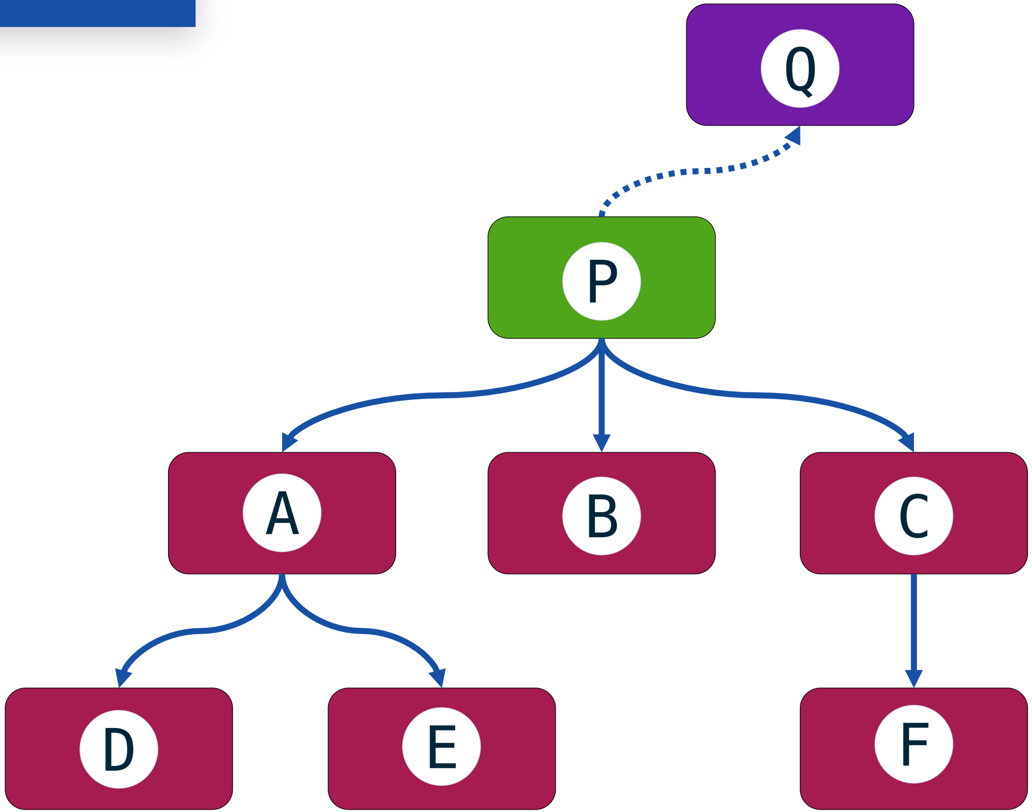


OVERVIEW

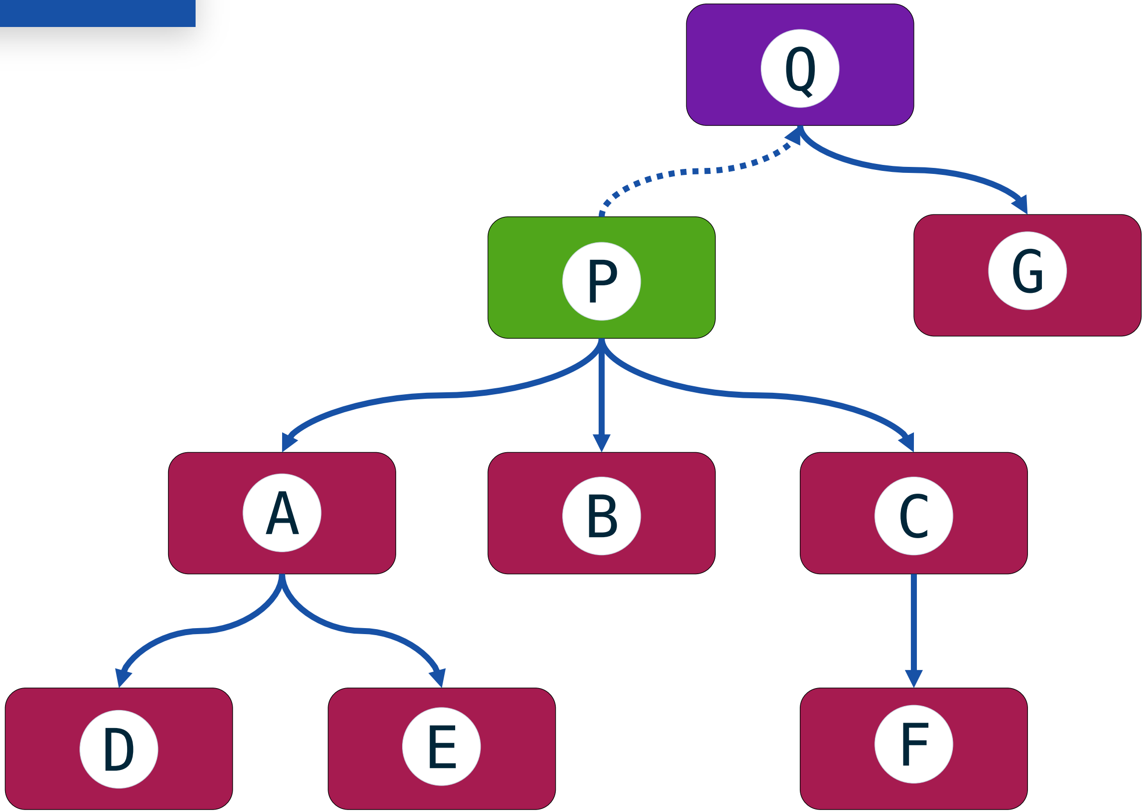



```xml
<parent>
   <groupId>org.Q</groupId>
   <artifactId>Q</artifactId>
</parent>
```

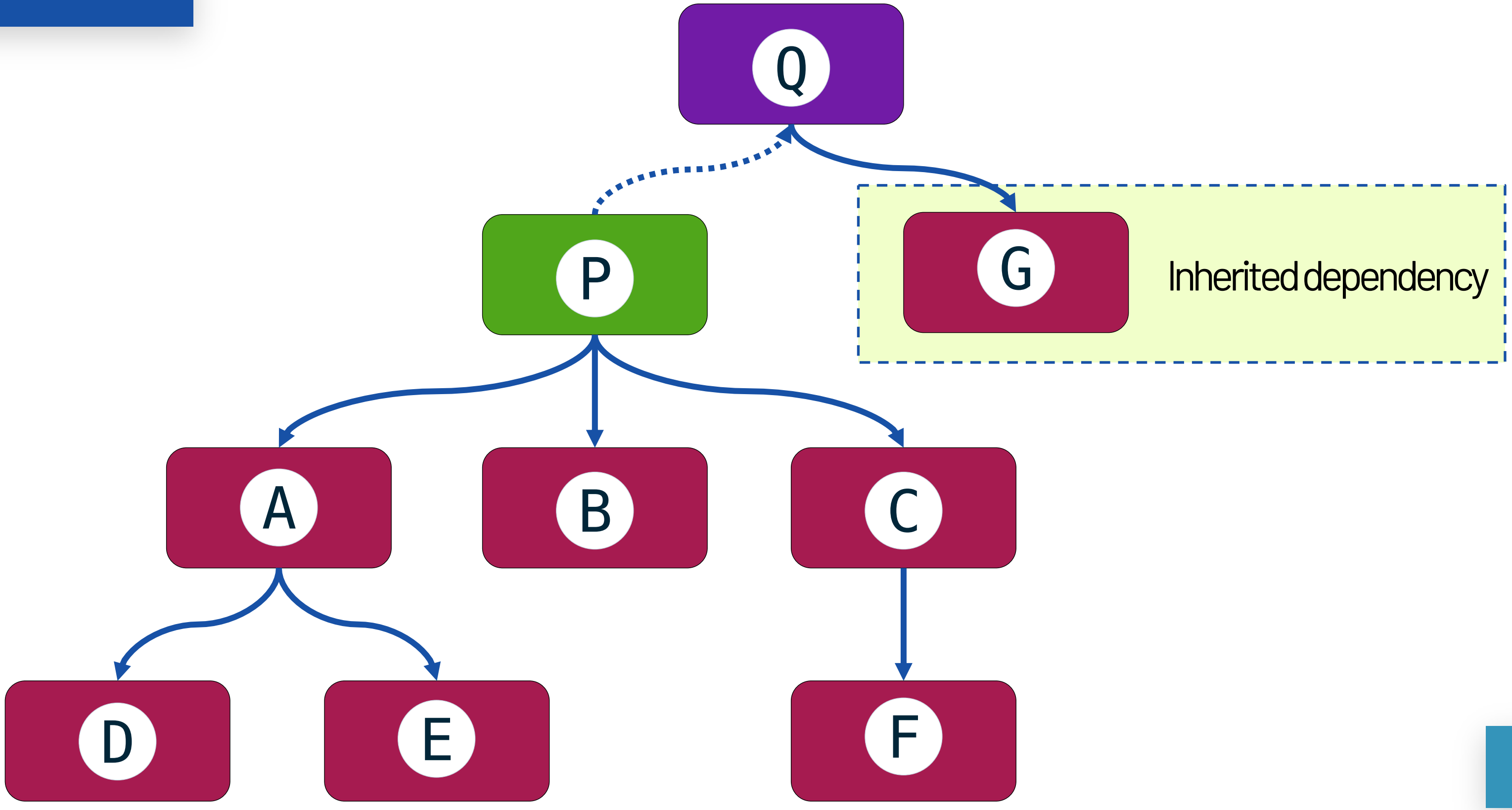
OVERVIEW



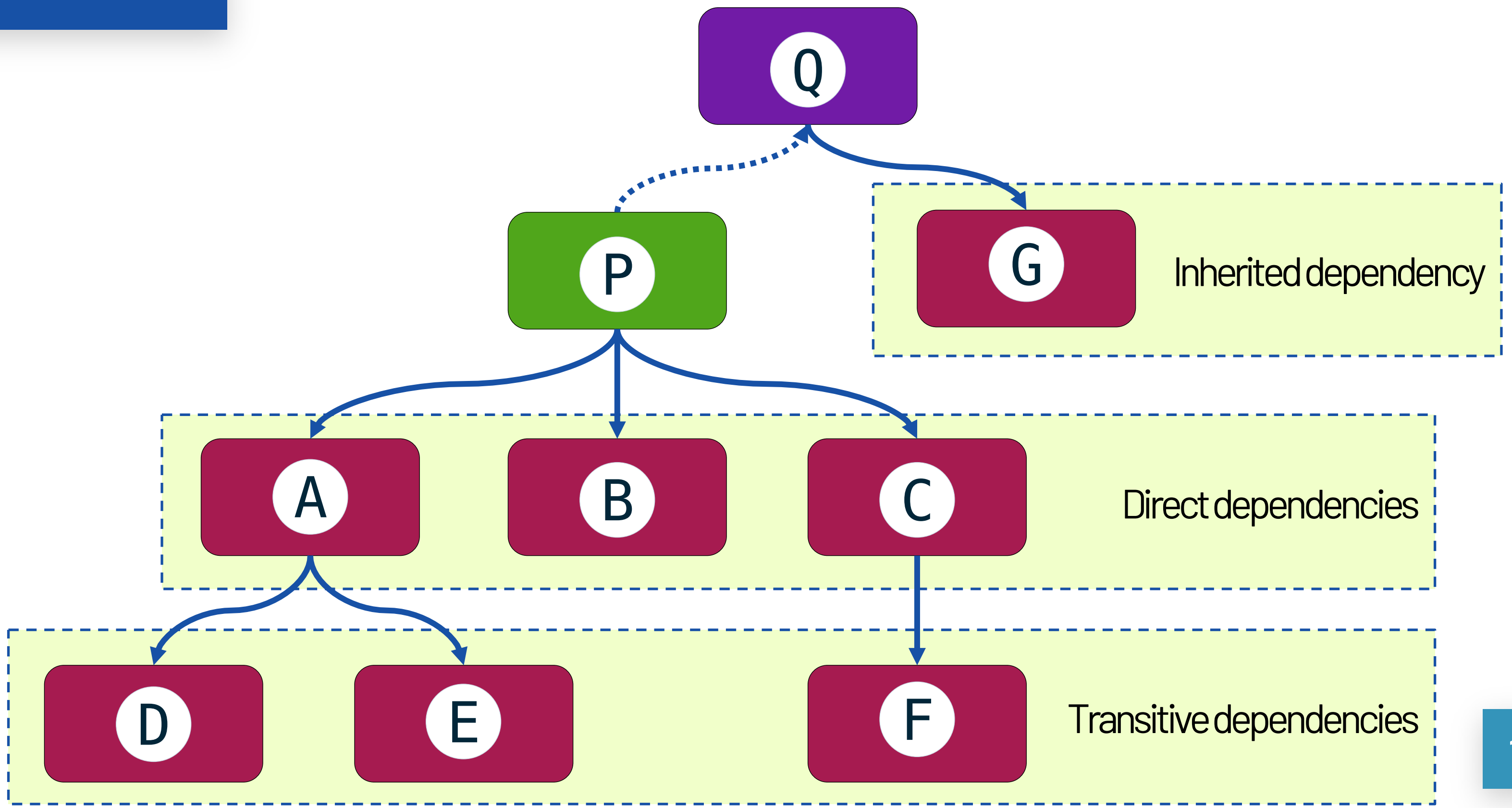
OVERVIEW



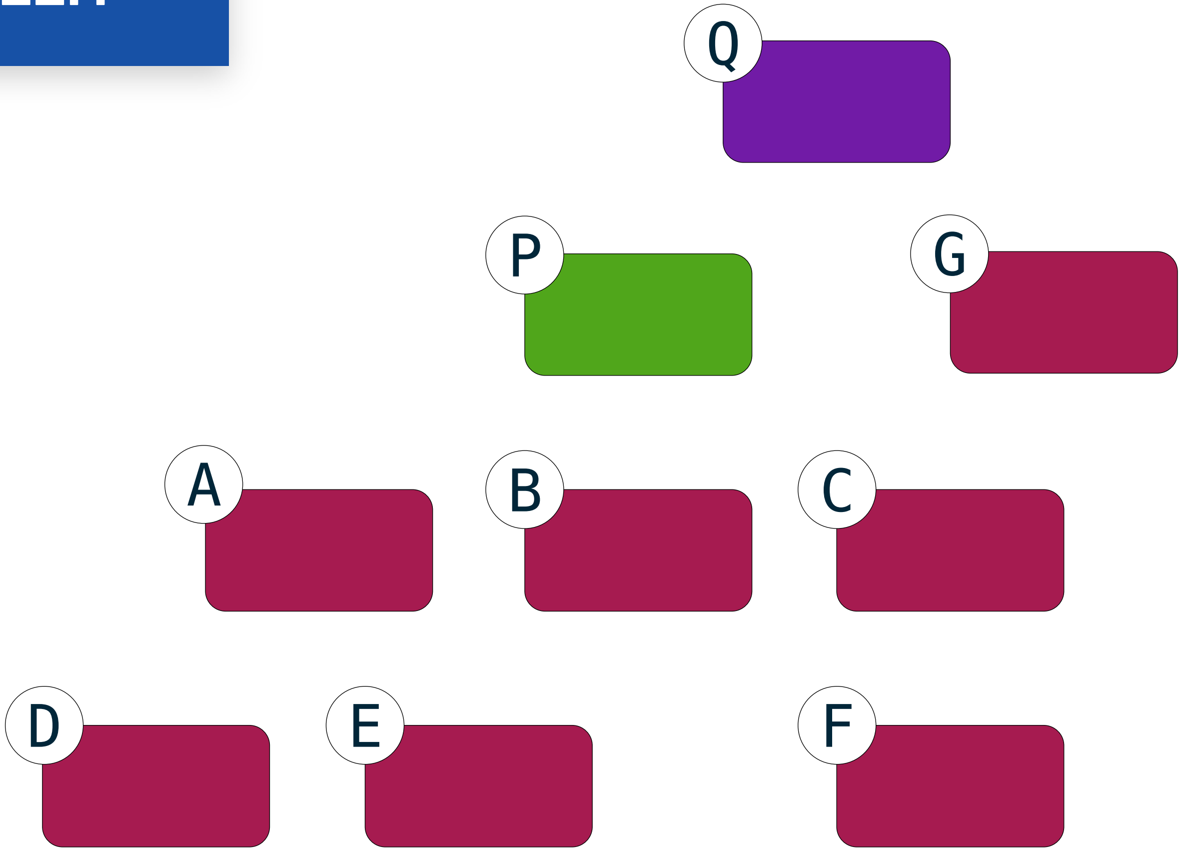
OVERVIEW



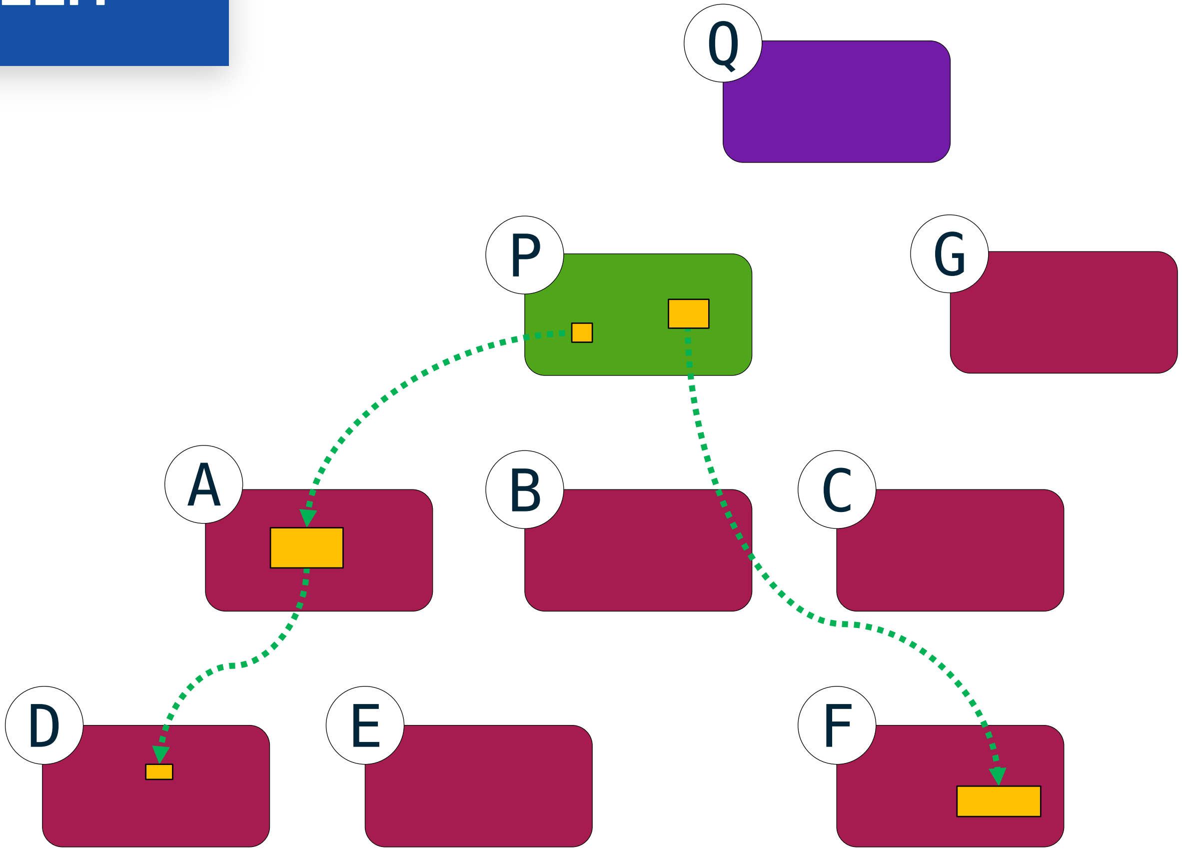
OVERVIEW


Q
G
Inherited dependency
P
A
B
C
Direct dependencies
D
E
F
Transitive dependencies

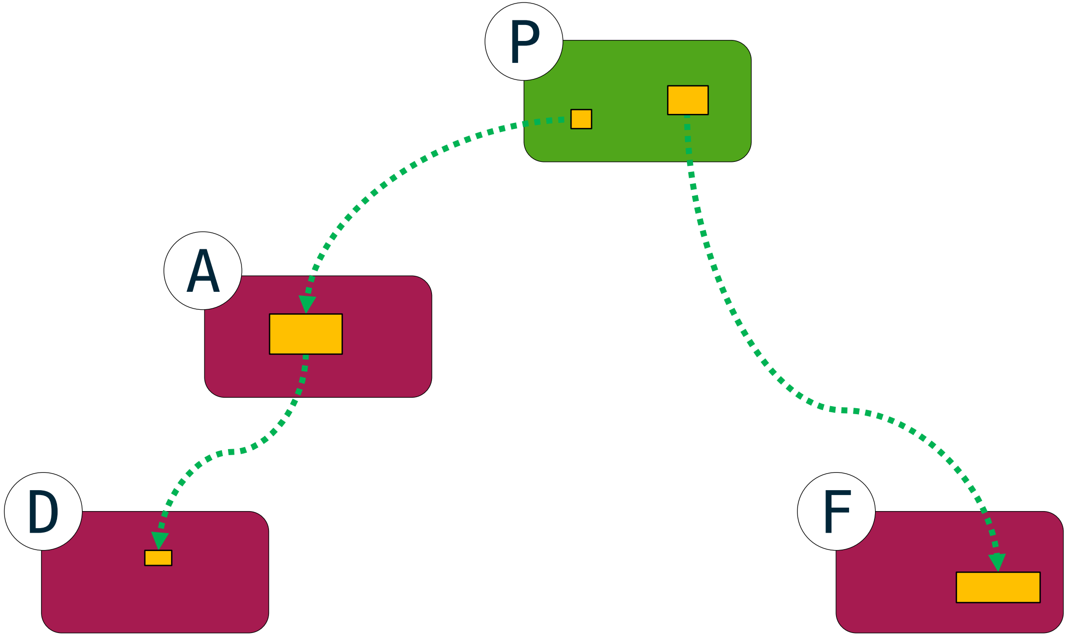
PROBLEM



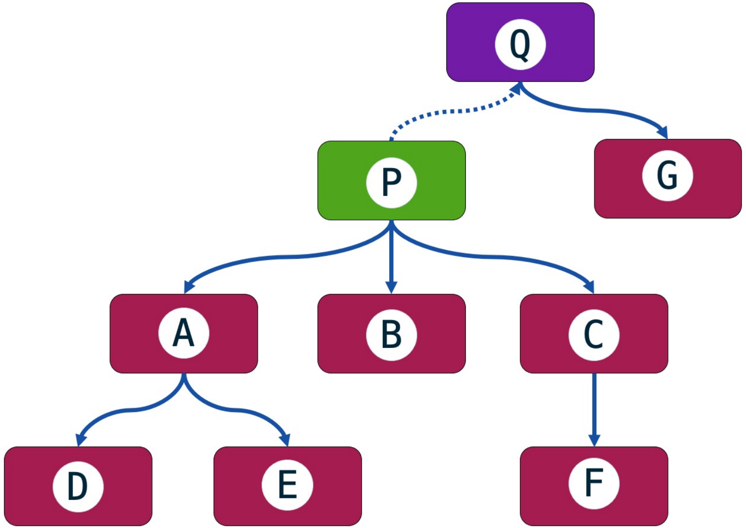
PROBLEM



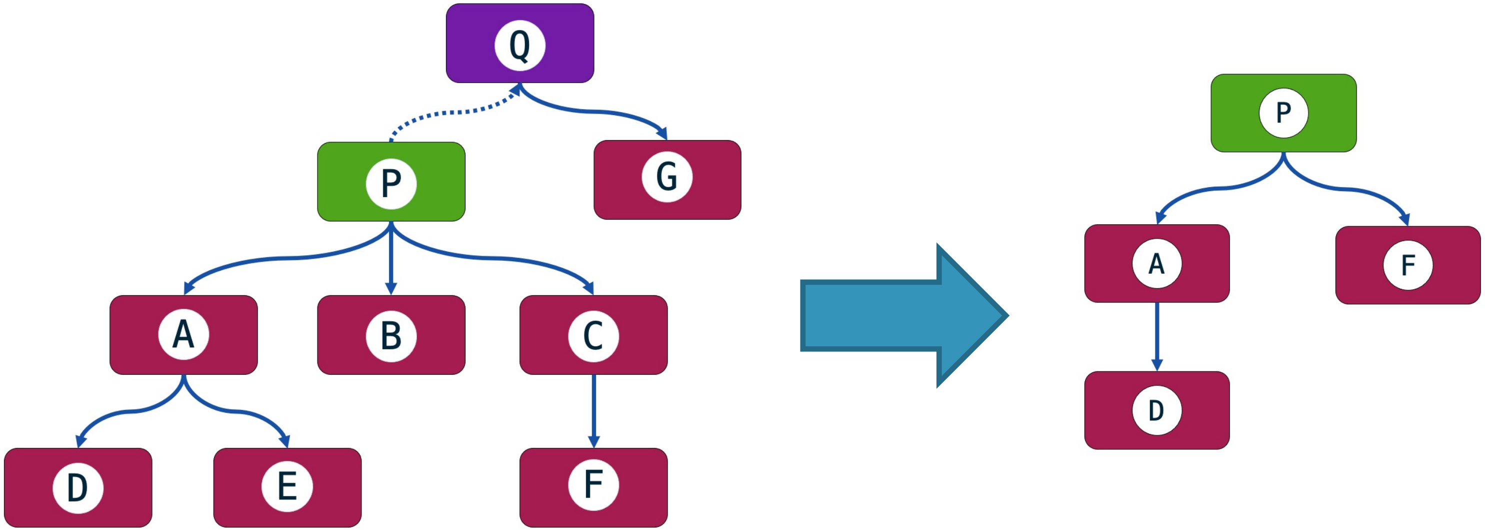
PROBLEM



PROBLEM

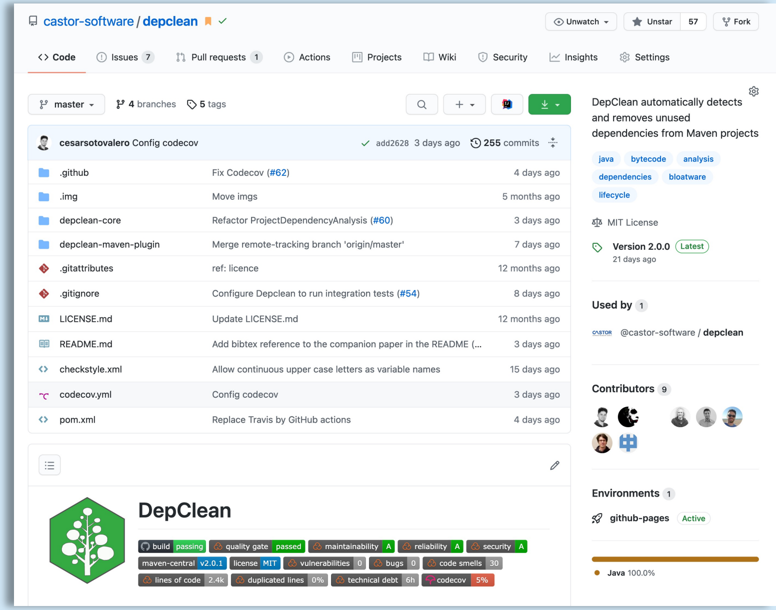


PROBLEM



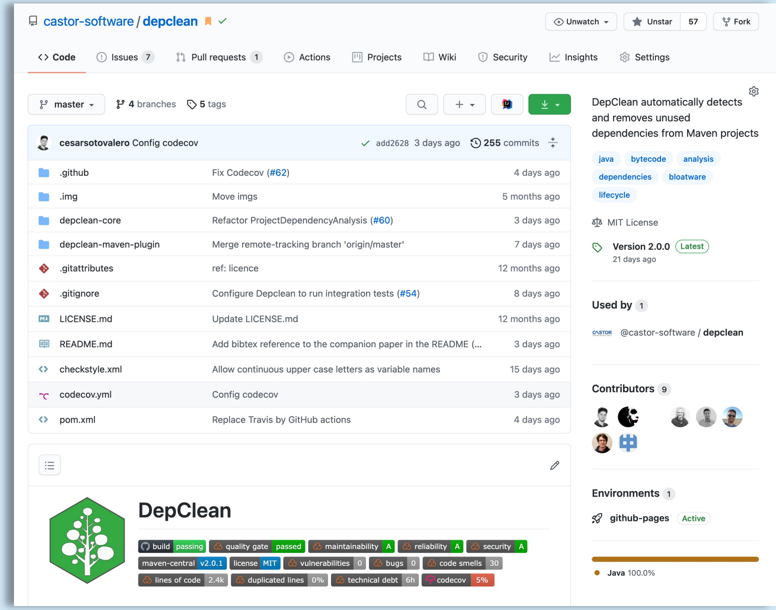
DEPCLEAN TOOL

https://github.com/castor-software/depclean



DEPCLEAN TOOL

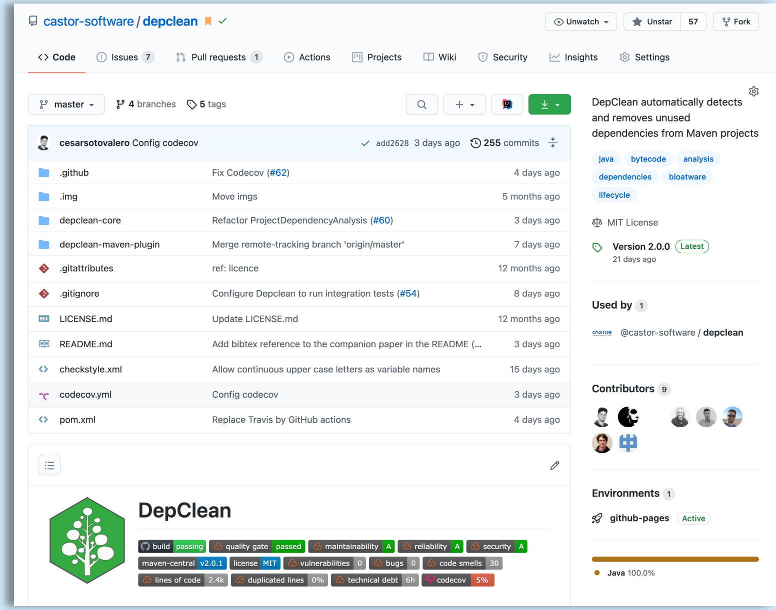
https://github.com/castor-software/depclean



Uses advanced static bytecode analysis to detect and remove bloated dependencies

DEPCLEAN TOOL

https://github.com/castor-software/depclean

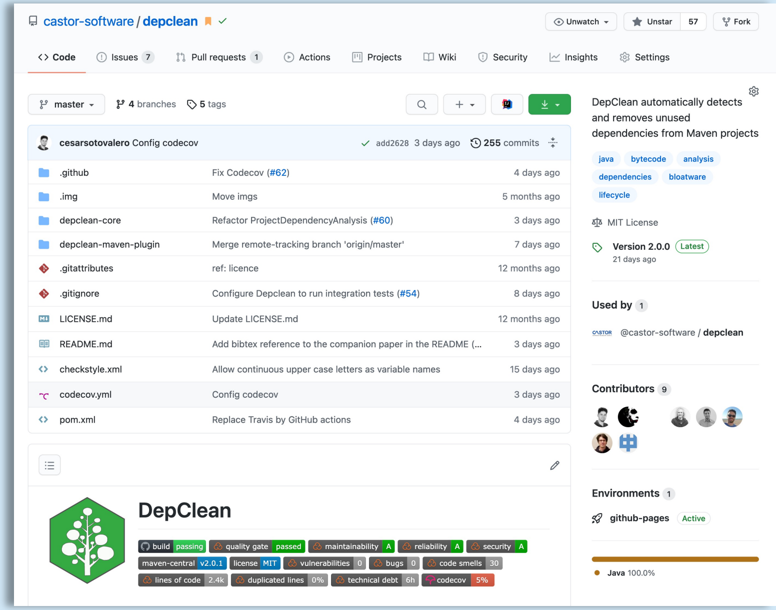


Uses advanced static bytecode analysis to detect and remove bloated dependencies

Automatic generation of a debloated POM file

DEPCLEAN TOOL

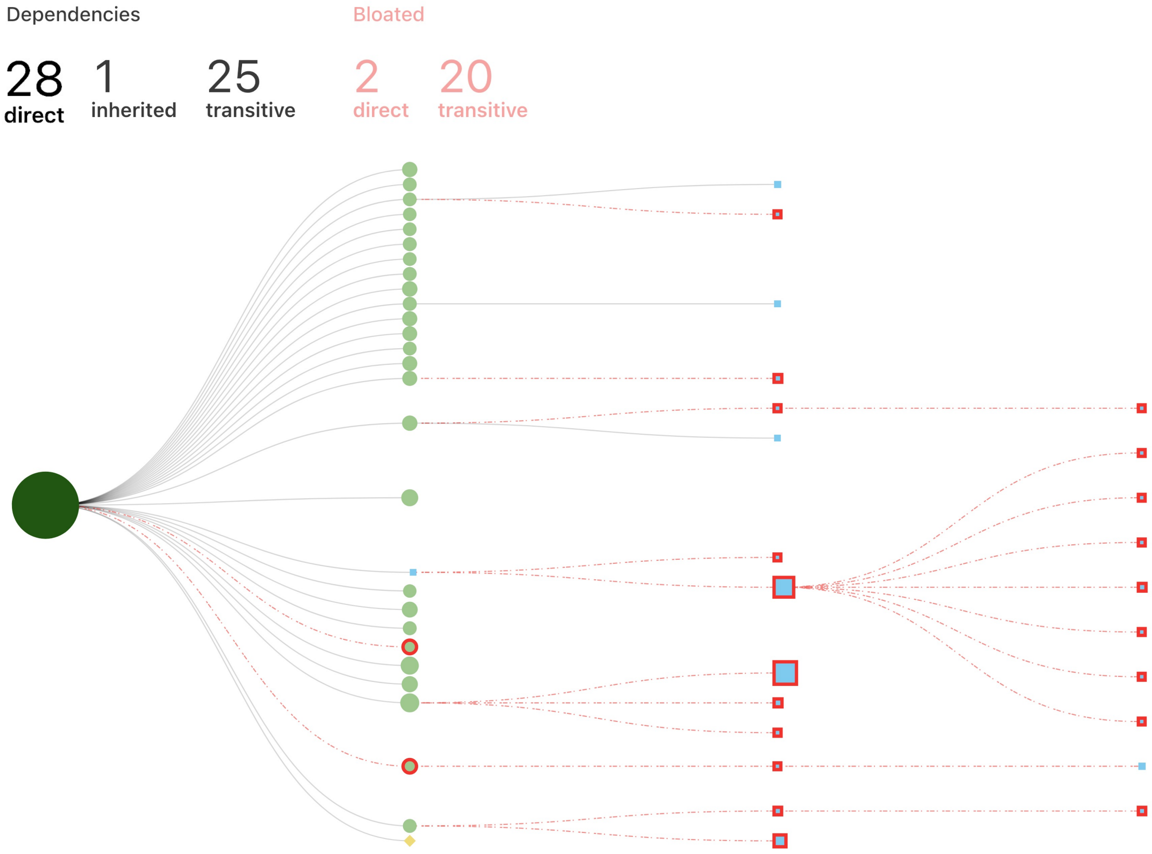
https://github.com/castor-software/depclean



Uses advanced static bytecode analysis to detect and remove bloated dependencies

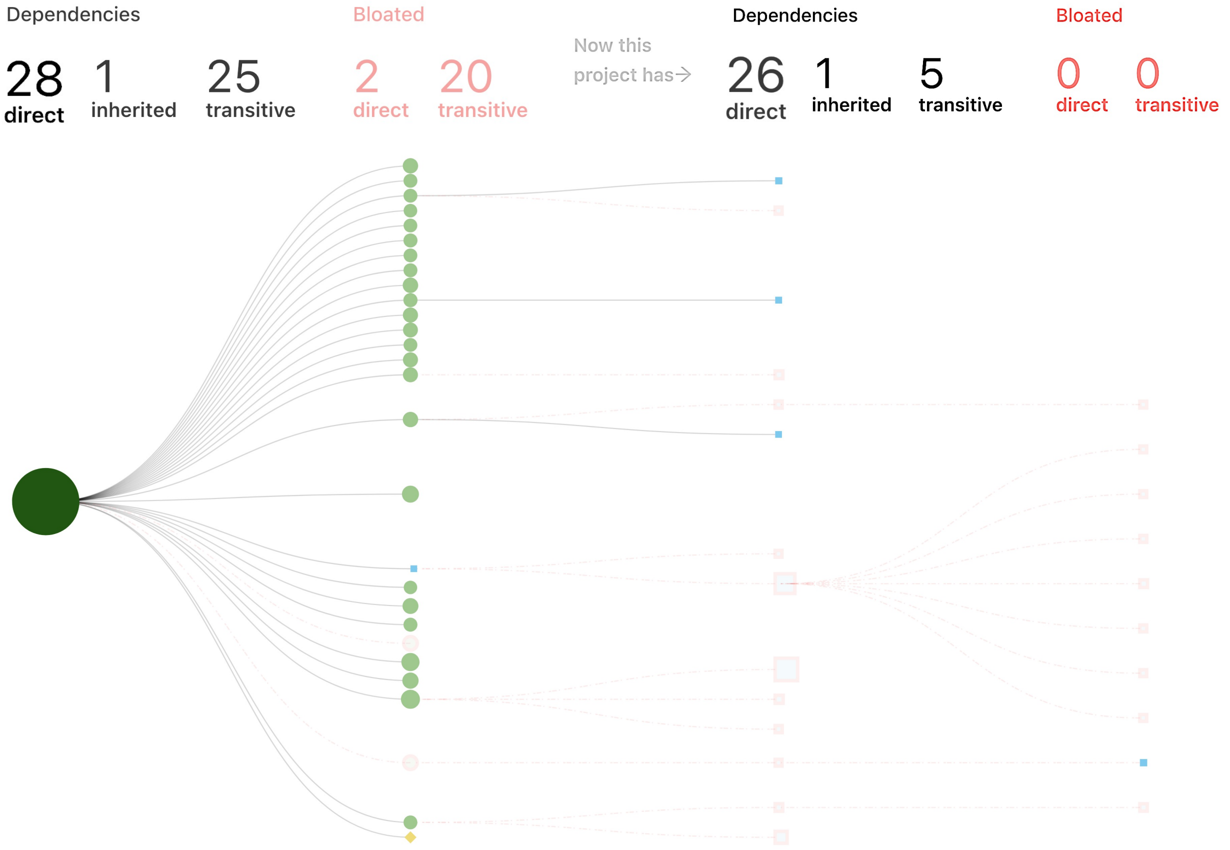
Automatic generation of a debloated POM file

Maven plugin easy to integrate in a CI pipeline



Example: maven-core project (v3.7.0)

https://github.com/castor-software/depclean-web

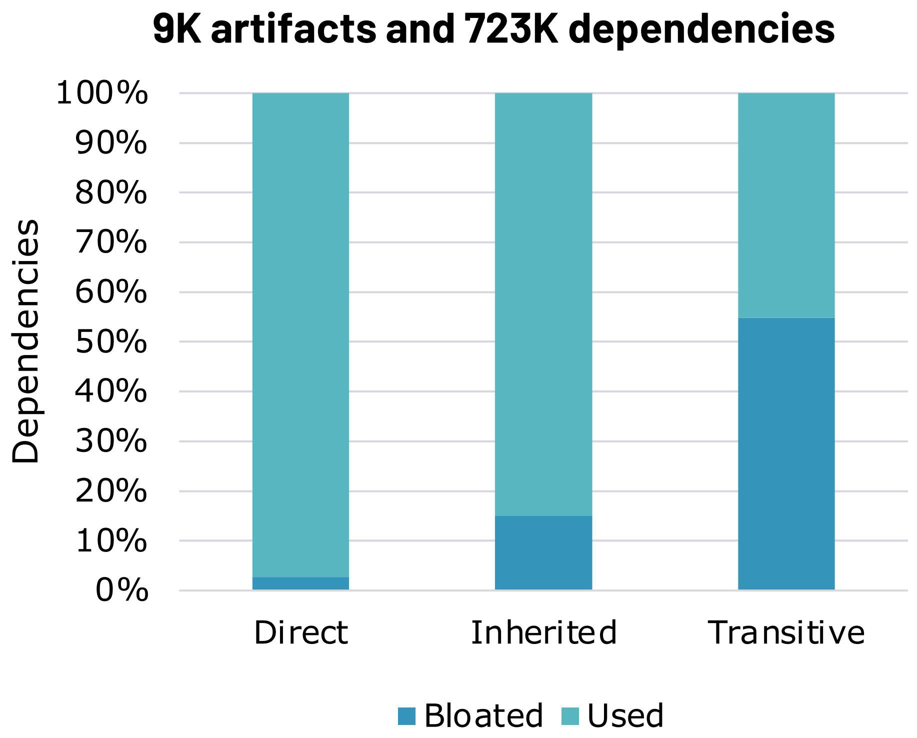


Example: maven-core project (v3.7.0)

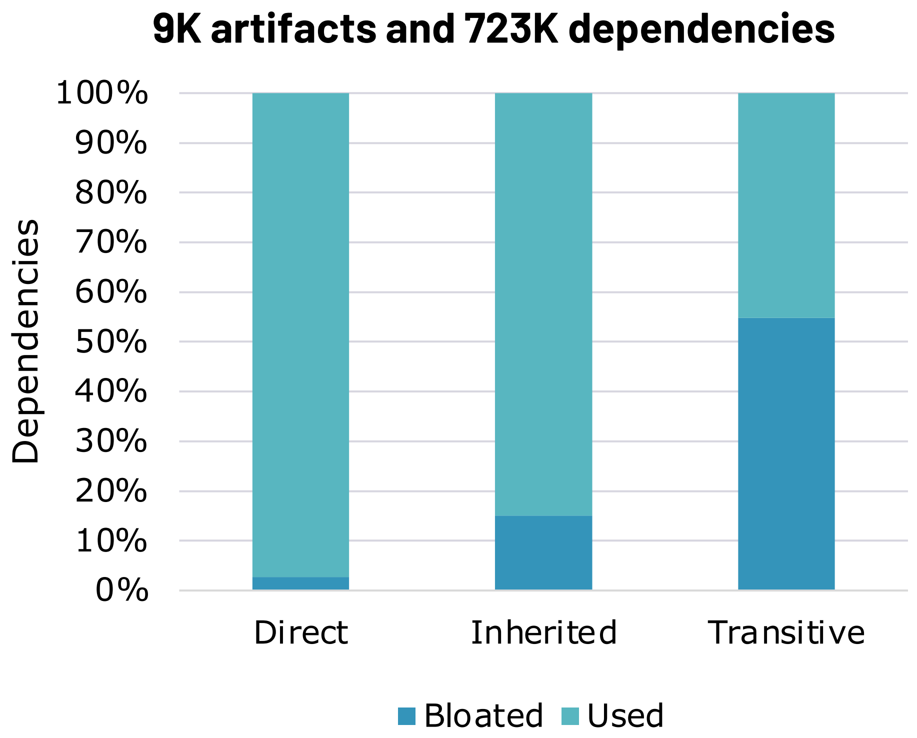
https://github.com/castor-software/depclean-web

HOW MUCH DEPENDENCY BLOAT EXISTS?

HOW MUCH DEPENDENCY BLOAT EXISTS?



HOW MUCH DEPENDENCY BLOAT EXISTS?



- 2.7% of direct dependencies are bloated
- 15.1% of inherited dependencies are bloated
- 57% of transitive dependencies are bloated

ARE DEVOPERS WILLING TO REMOVE BLOAT?

ARE DEVOPERS WILLING TO REMOVE BLOAT?

USER STUDY ON 30 PROJECTS

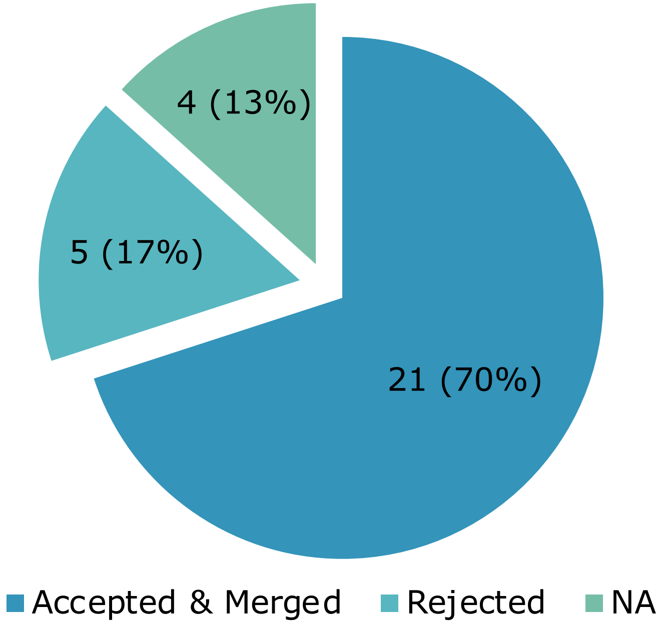
- Jenkins
- Neo4j
- Flink
- Spoon
- Checkstyle
- CoreNLP
- jHiccup
- Alluxio
- TeaVM

…

Full list: https://tinyurl.com/depclean-experiments

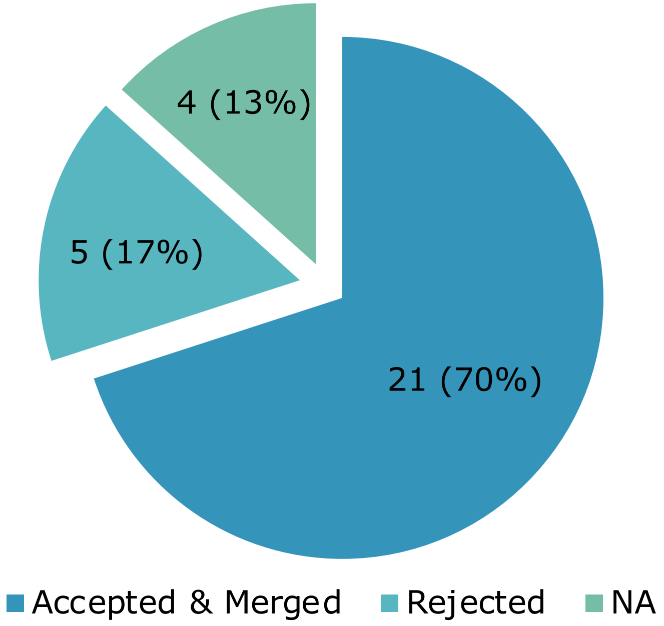
ARE DEVOPERS WILLING TO REMOVE BLOAT?

30 pull requests in 30 notable open source projects



ARE DEVOPERS WILLING TO REMOVE BLOAT?

30 pull requests in 30 notable open source projects



Removed 140 bloated dependencies in 21 projects thanks to DepClean

SUMMARY OF 1st CONTRIBUTION

SUMMARY OF 1st CONTRIBUTION

- **There is a lot of code bloat in Maven Central**
  - Caused by the induced transitive dependencies
  - Caused by the heritage mechanism of multi-module projects
  - Caused by software development practices

SUMMARY OF 1st CONTRIBUTION

- **There is a lot of code bloat in Maven Central**
  - Caused by the induced transitive dependencies
  - Caused by the heritage mechanism of multi-module projects
  - Caused by software development practices
- **Software developers care**
  - They are willing to remove bloated dependencies

SUMMARY OF 1st CONTRIBUTION

- **There is a lot of code bloat in Maven Central**
  - Caused by the induced transitive dependencies
  - Caused by the heritage mechanism of multi-module projects
  - Caused by software development practices
- **Software developers care**
  - They are willing to remove bloated dependencies
- **DepClean**
  - It is useful to automatically detect and remove bloated dependencies

2nd CONTRIBUTION

Longitudinal Analysis of Bloated Java Dependencies

A Longitudinal Analysis of Bloated Java Dependencies

César Soto-Valero
KTH Royal Institute of Technology, Sweden
cesarsv@kth.se

Thomas Durieux
KTH Royal Institute of Technology, Sweden
thomas@durieux.me

Benoit Baudry
KTH Royal Institute of Technology, Sweden
baudry@kth.se

ABSTRACT

Motivated by the negative impact of software bloat on security, performance, and maintenance, several works have proposed techniques to remove bloat. However, no work has analyzed how bloat evolves over time or how it emerges in software projects. In particular, a concern when removing bloated code is to know if it might be useful in subsequent versions of the application. In this work, we study the evolution and emergence of bloated Java dependencies. These are third-party libraries that are packaged in the application binary but are not needed to run the application. We analyze the history of 435 Java projects. This historical data includes 48,469 distinct dependencies, which we study across a total of 31,515 versions of Maven dependency trees. We empirically demonstrate the constant increase of the amount of bloated dependencies over time. A key finding of our analysis is that 89.2 % of the direct dependencies that are bloated remain bloated in all subsequent versions of the studied projects. This empirical evidence suggests that developers can safely remove a bloated dependency. We further report novel insights regarding the unnecessary maintenance efforts induced by bloat, we identify that 22 % of dependency updates are made on bloated dependencies.

ACM Reference Format:
César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A Longitudinal Analysis of Bloated Java Dependencies. In *ESEC/FSE'21*. ACM, New York, NY, USA, 11 pages. https://doi.org/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software is bloated. From single Unix commands [6] to web browsers [15], most applications embed a part of code that is unnecessary to their correct operation. Several debloating tools have emerged in recent years [7, 14, 15, 17, 19, 21] to address the security and maintenance issues posed by excessive code at various granularity levels. However, these works do not analyze the evolution of bloat over time. Understanding software bloat in the perspective of software evolution is crucial to promote debloating tools towards software developers. In particular, developers, when proposed to adapt a debloating tool, wonder if a piece of bloated code might be needed in coming releases, or what is the actual issue with bloat.

This work proposes the first longitudinal analysis of software bloat. We focus on one specific type of bloat: bloated dependencies [21]. These are software libraries that are unnecessarily part of software projects, i.e., when the dependency is removed from the project, it still builds successfully. Soto-Valero et al. [21] show that the Maven ecosystem is permeated with bloated dependencies, and that they are present even in well maintained Java projects. They also demonstrate that software developers are keen on removing bloated dependencies, but that removing code is a complex socio-technical decision, which benefits from solid evidence about the actual benefits of debloating.

Motivated by these observations about bloated dependencies, we conduct a large scale empirical study about the evolution of these dependencies in Java projects. We analyze the emergence of bloat, the evolution of the dependencies statuses, and the impact of bloat on maintenance. We have collected a unique dataset of 31,515 versions of dependency trees from 435 open-source Java projects. Each version of a tree is a snapshot of one project's dependencies, for which we determine a status, i.e. bloated or used. We rely on DepClean,[1] the state-of-the-art tool to detect bloated dependencies in Maven projects. We analyze the evolution of 48,469 distinct dependencies per project and we observe that 40,493/48,469 (83.5 %) of them are bloated at one point in time, in our dataset.

Our longitudinal analysis of bloated Java dependencies investigates both the evolution of bloat, as well as the relation between bloat and regular maintenance activities such as dependency updates. We present original quantitative results regarding the evolution of bloated dependencies. We first show a clear increasing trend in the number of bloated dependencies. Next, we investigate how the usage status of dependencies evolves over time. This analysis is a key contribution of our work where we demonstrate that a dependency that is bloated is very likely to remain bloated over subsequent versions of a project. We present the first observations about the impact of regular maintenance activities on software bloat. Besides, we analyze the impact of Dependabot, a popular dependency management bot, on these activities. We show that developers regularly update bloated dependencies, and that many of these updates are suggested by Dependabot. Furthermore, we systematically investigate the root of the bloat emergence, and find that 84.3 % of the bloated dependencies are bloated as soon as they are added in the dependency tree of a project.

To summarize, the contributions of this paper are:

- A longitudinal analysis of software dependencies' usage in 31,515 versions of Maven dependency trees. Our results confirm the generalized presence of bloated dependencies and show their increase over time.
- A quantitative analysis of the stability of bloated dependencies: 89.2 % of direct dependencies remain bloated. This is a concrete insight that motivates debloating dependencies.

ESEC/FSE'21, August 23–27, 2021 , Athens, Greece, Virtual Event

ACM ISBN nnnnnnn.nnnnnnn.
https://doi.org/nnnnnnn.nnnnnnn

[1] https://github.com/castor-software/depclean

PROBLEM

PROBLEM

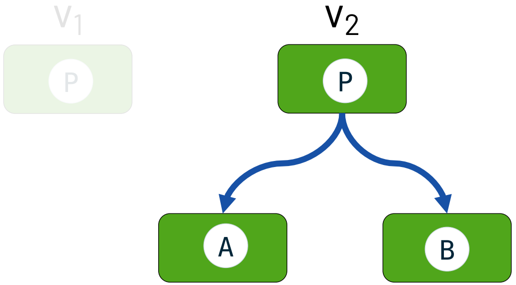
SOFTWARE EVOLVES OVER TIME

Time

PROBLEM

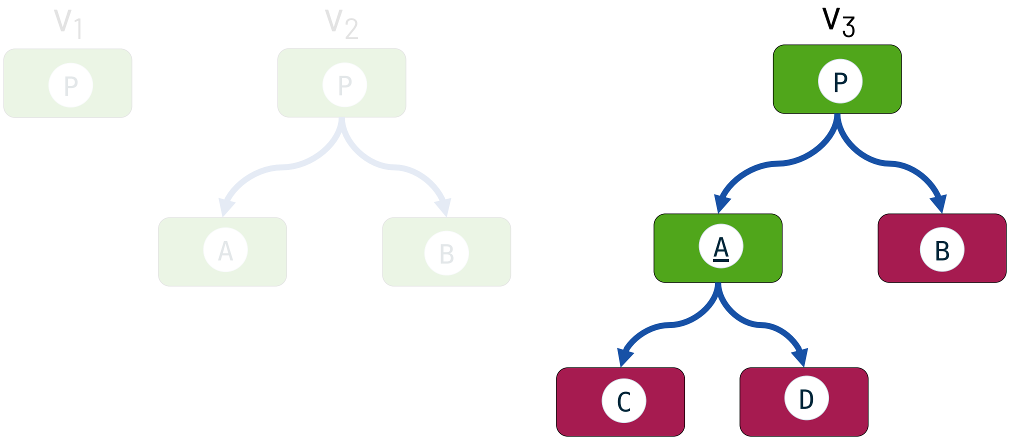
SOFTWARE EVOLVES OVER TIME

PROBLEM



SOFTWARE EVOLVES OVER TIME

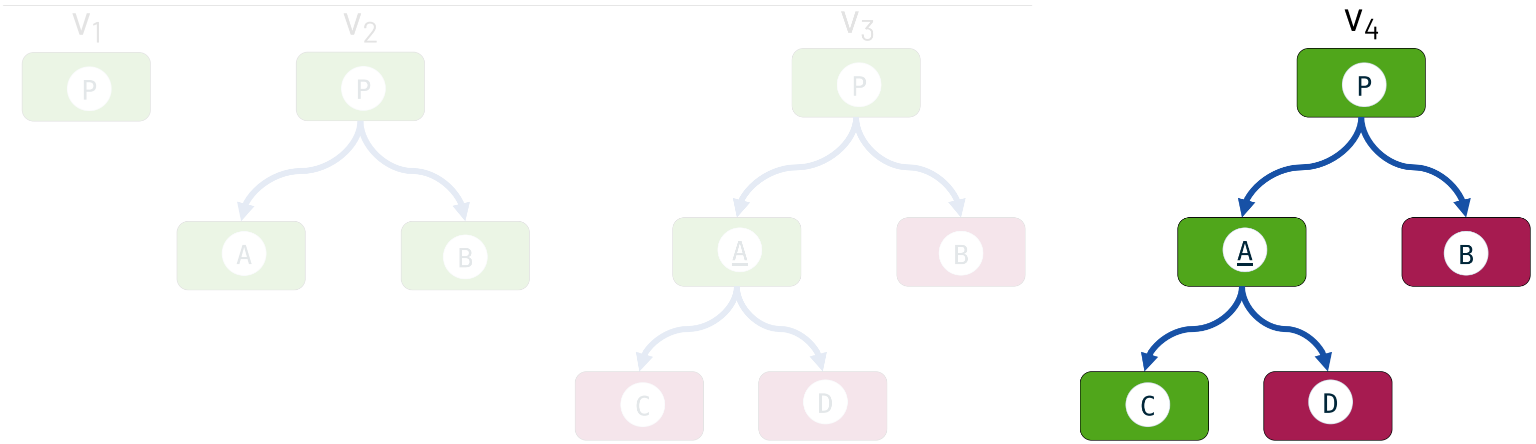
PROBLEM



SOFTWARE EVOLVES OVER TIME

Time

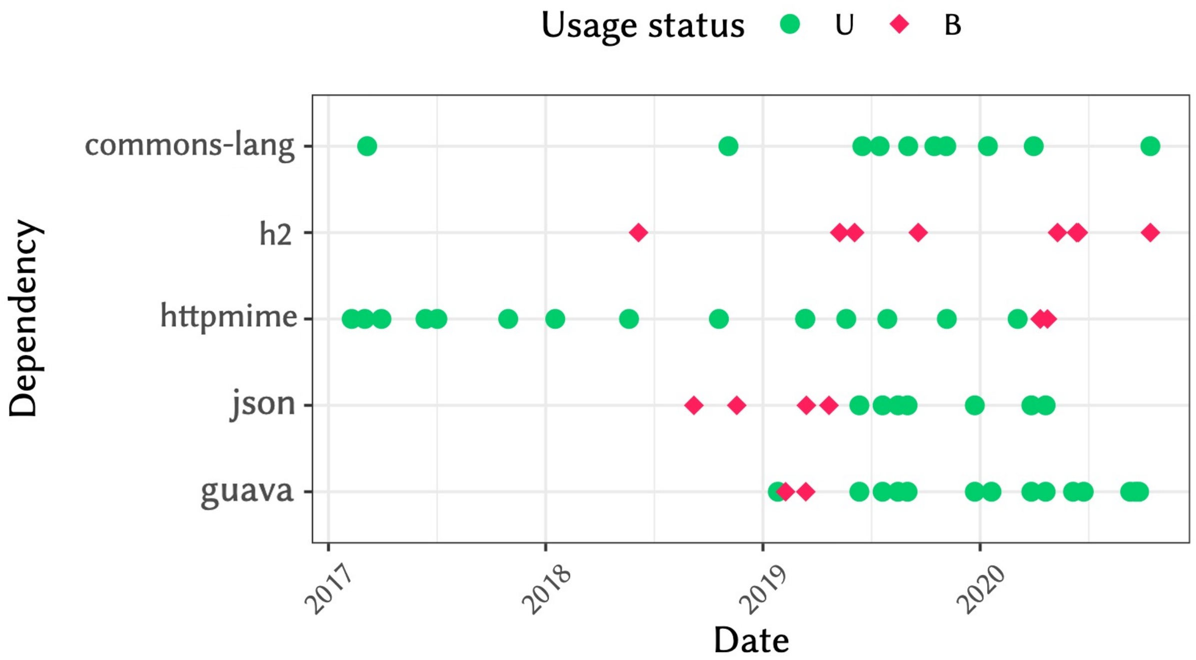
PROBLEM



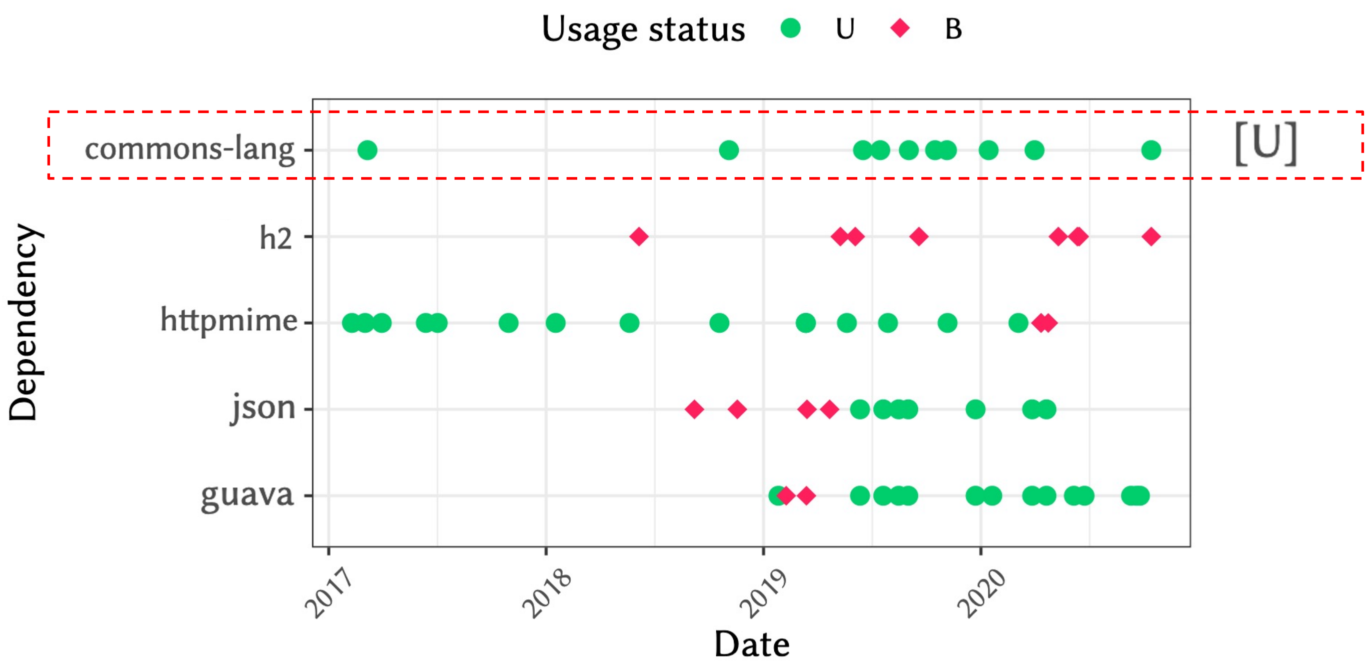
SOFTWARE EVOLVES OVER TIME

HOW DOES THE USAGE STATUS EVOLVES?

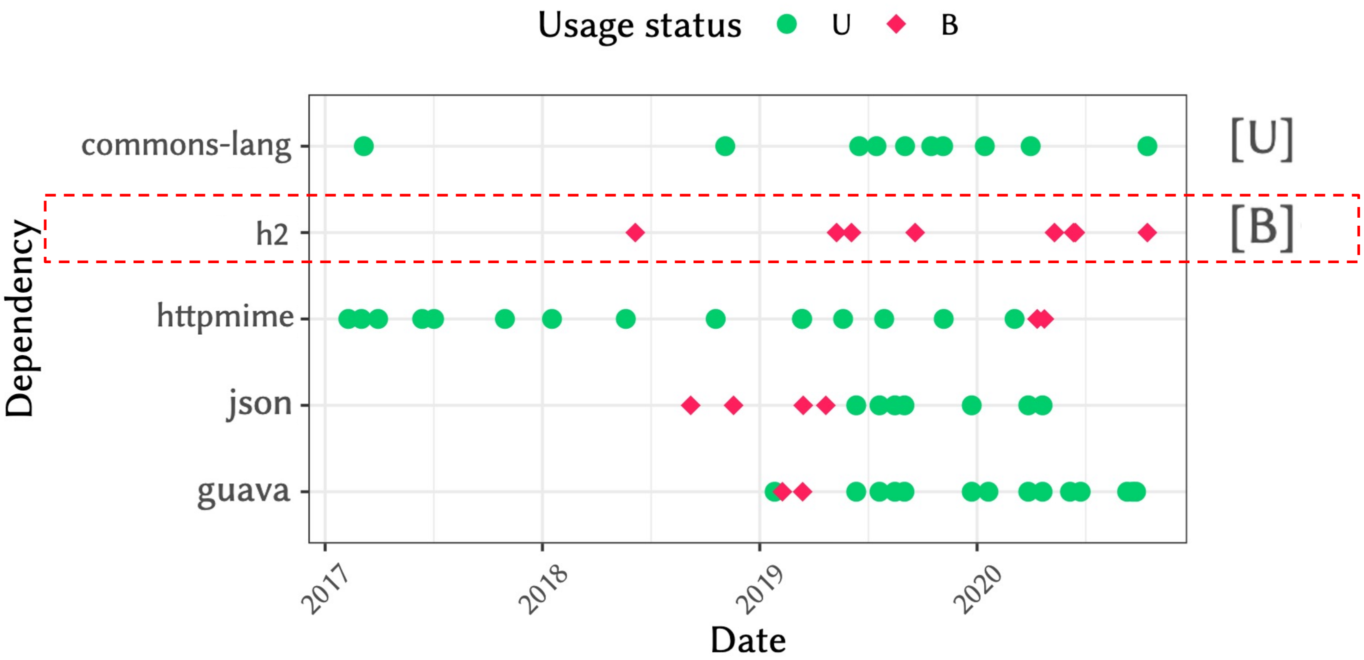
HOW DOES THE USAGE STATUS EVOLVES?



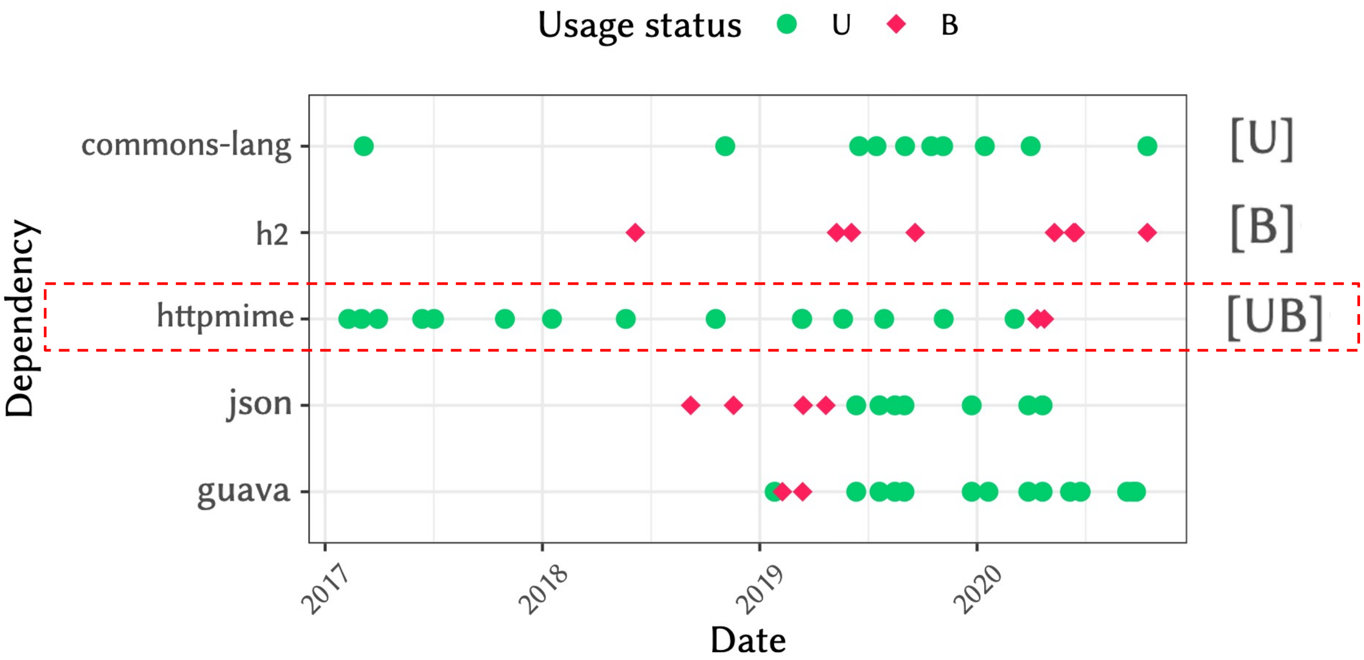
HOW DOES THE USAGE STATUS EVOLVES?



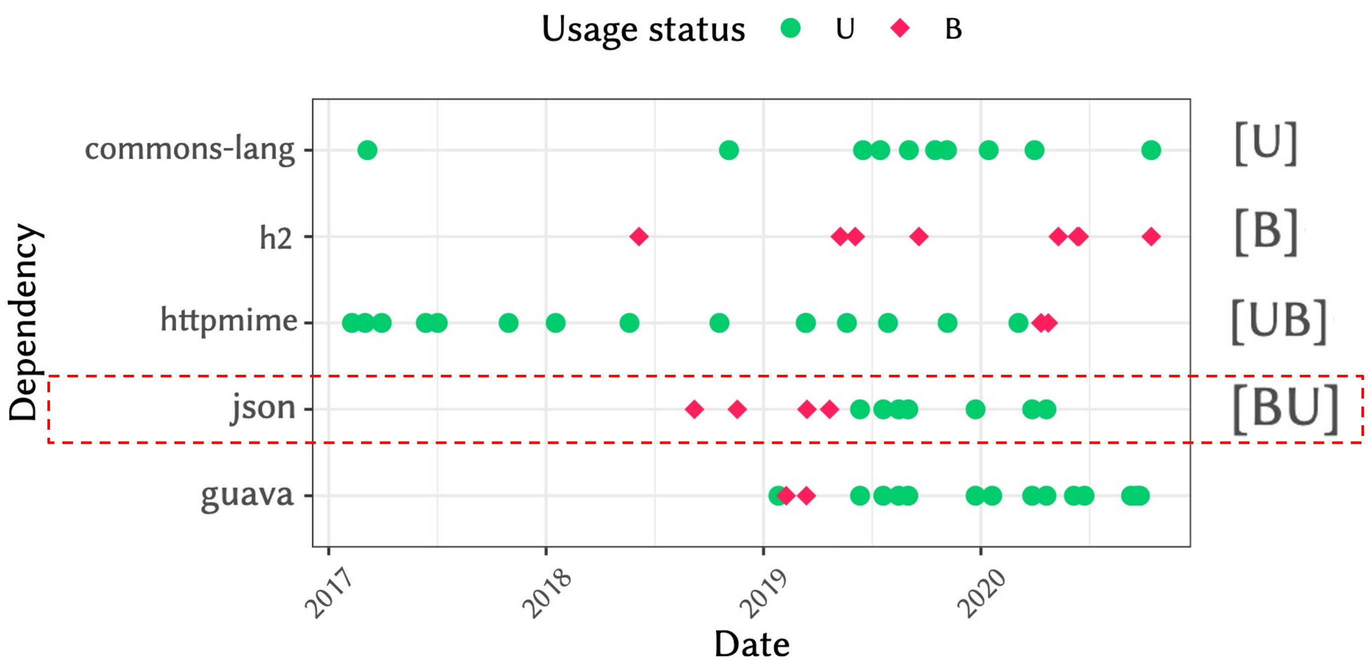
HOW DOES THE USAGE STATUS EVOLVES?



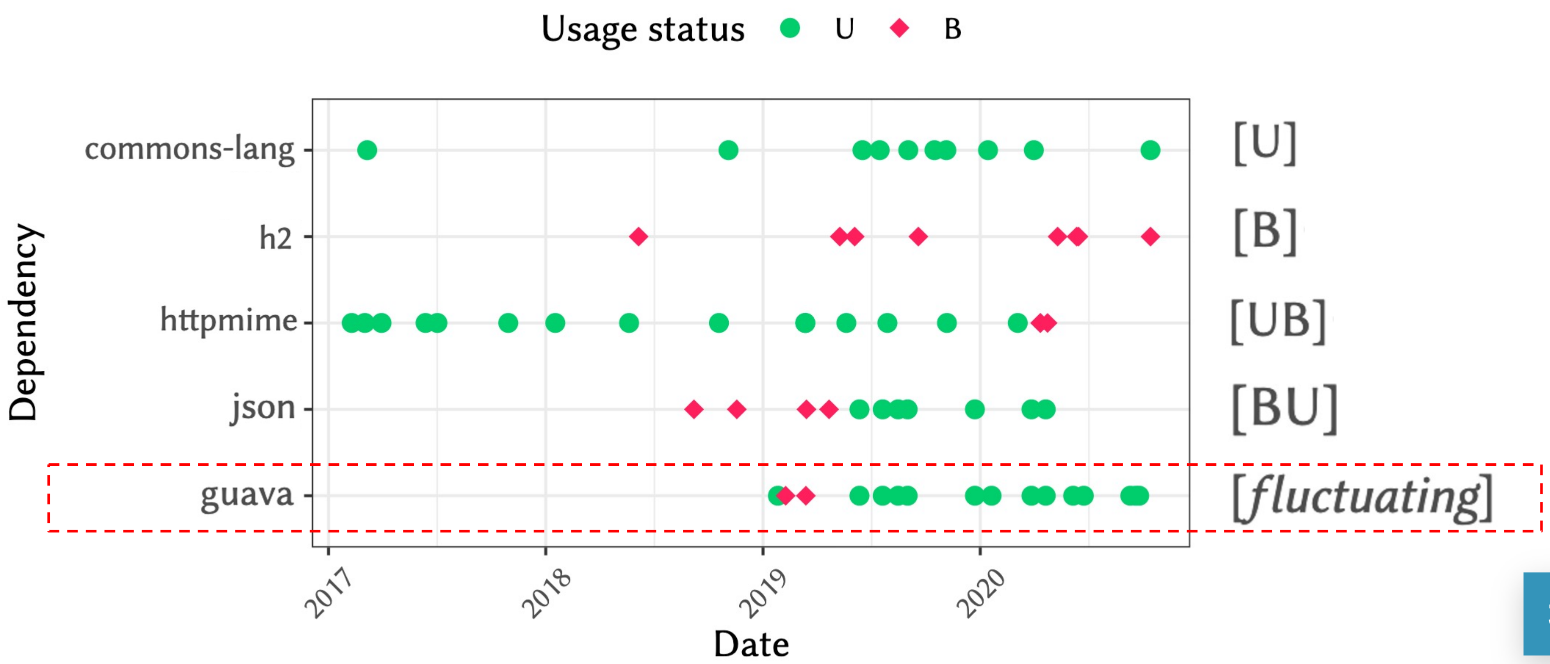
HOW DOES THE USAGE STATUS EVOLVES?



HOW DOES THE USAGE STATUS EVOLVES?

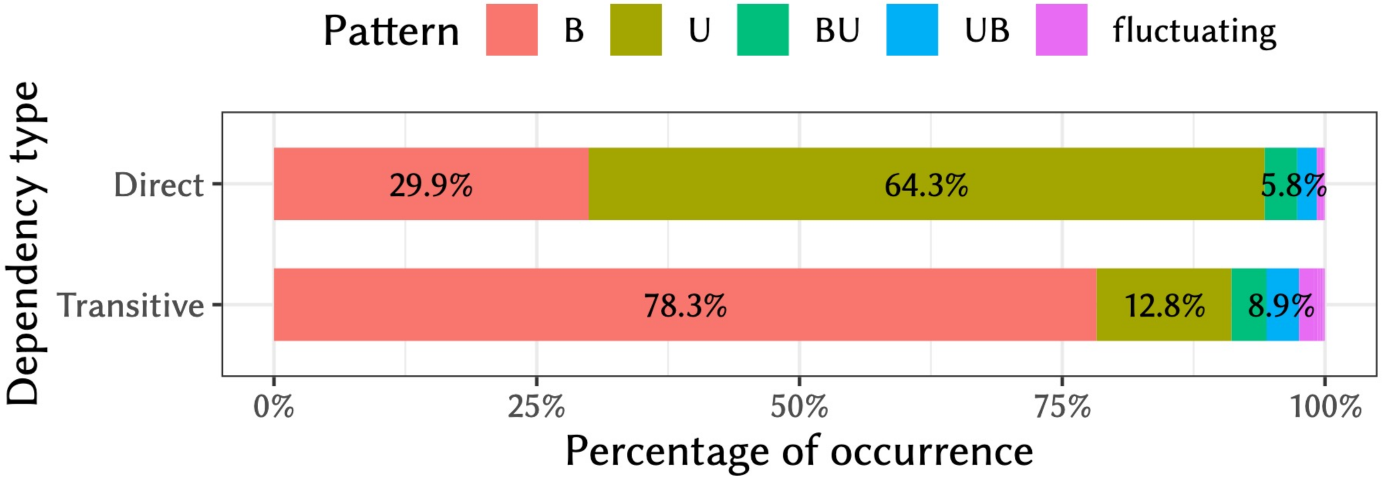


HOW DOES THE USAGE STATUS EVOLVES?

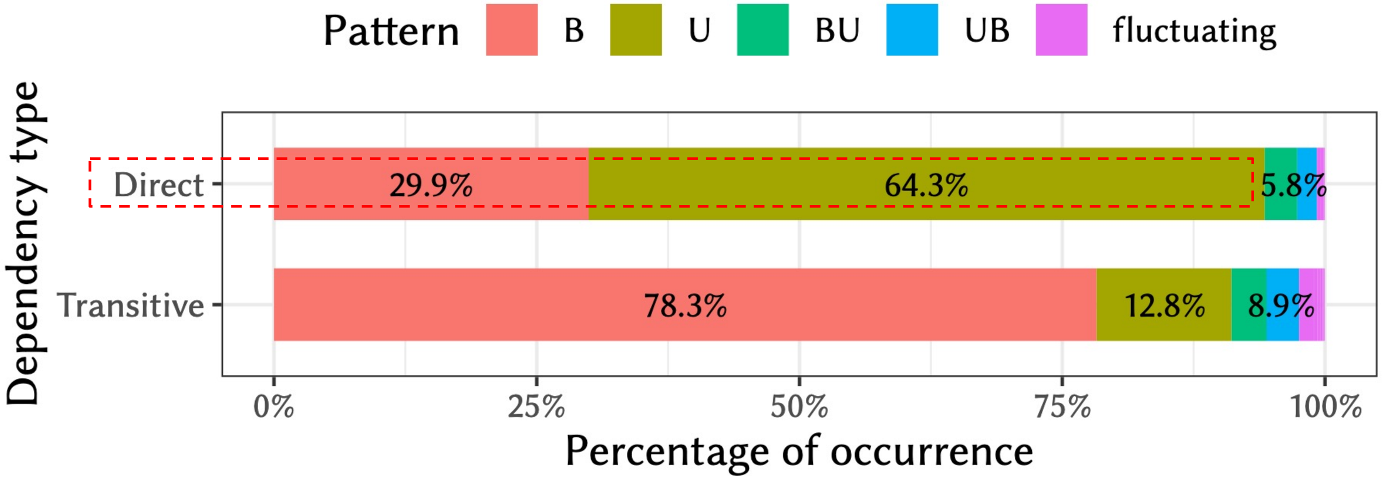


HOW DOES THE USAGE STATUS EVOLVES?

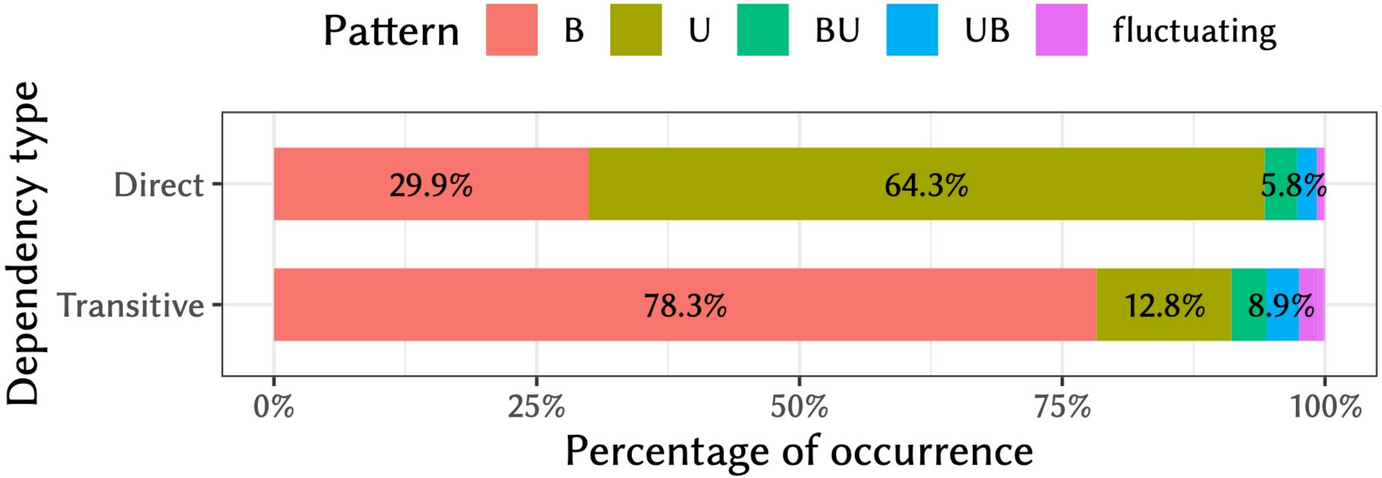
HOW DOES THE USAGE STATUS EVOLVES?



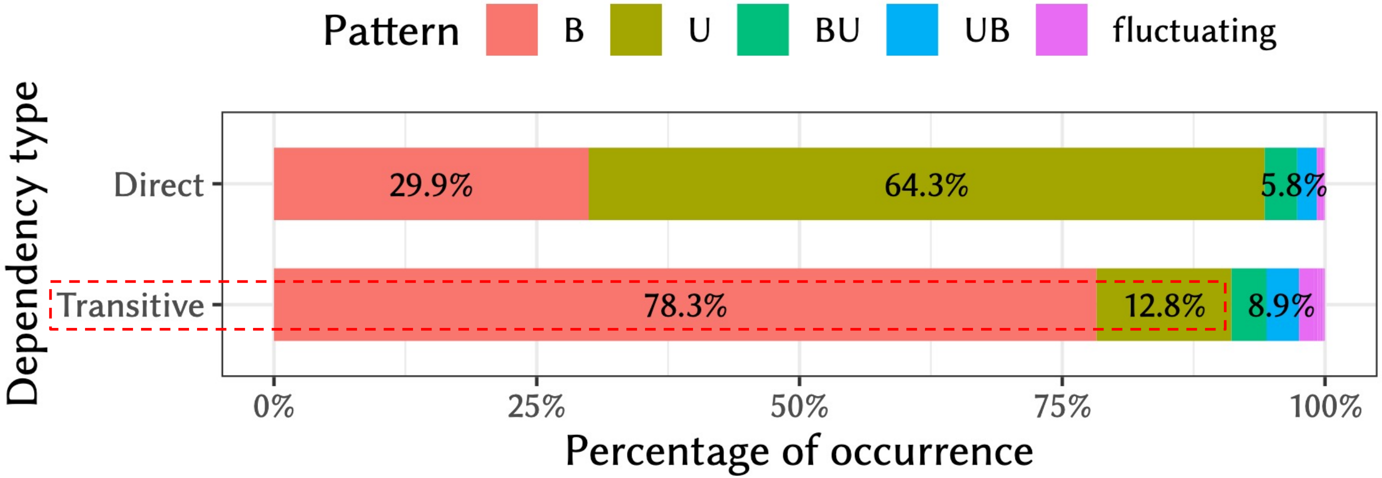
HOW DOES THE USAGE STATUS EVOLVES?



HOW DOES THE USAGE STATUS EVOLVES?

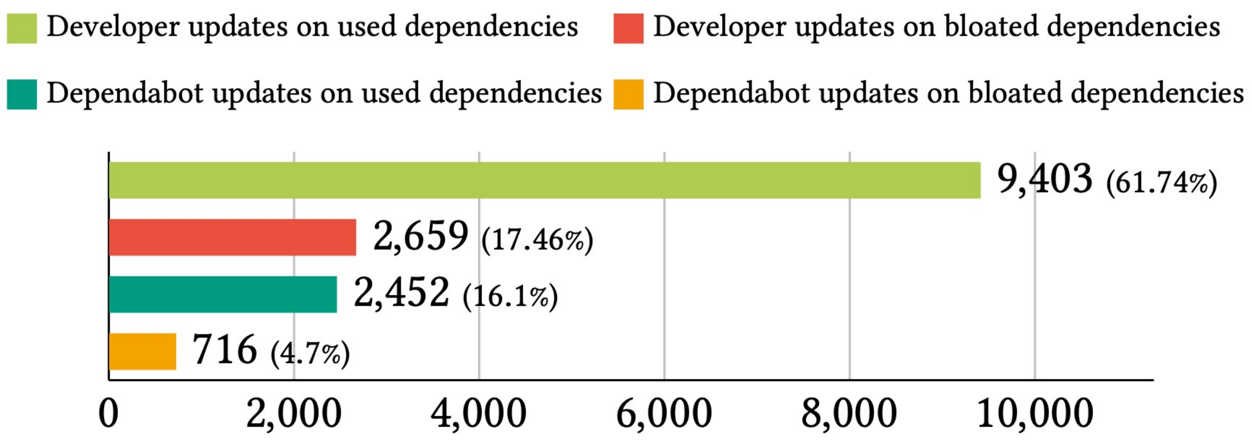


HOW DOES THE USAGE STATUS EVOLVES?

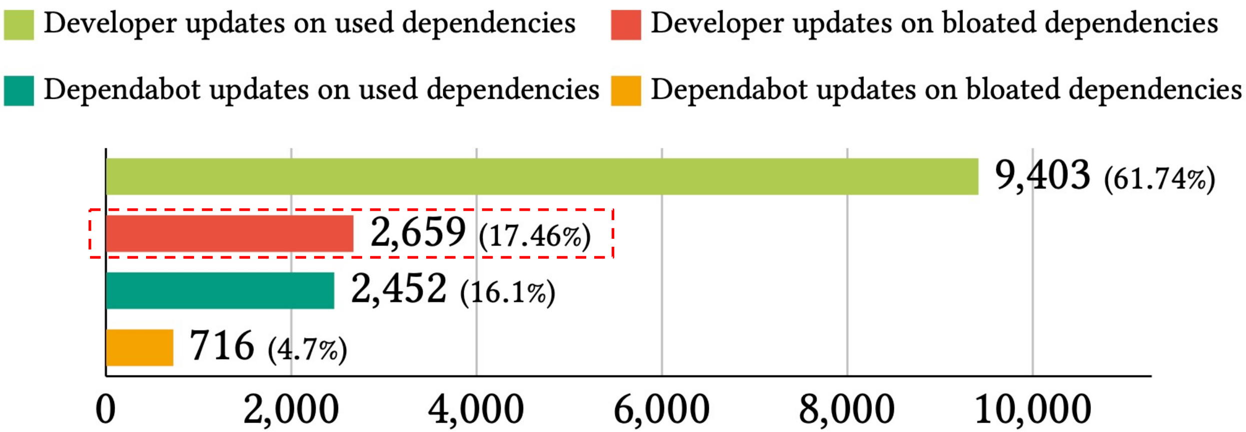


DO DEVELOPERS UPDATE BLOATED DEPENDENCIES?

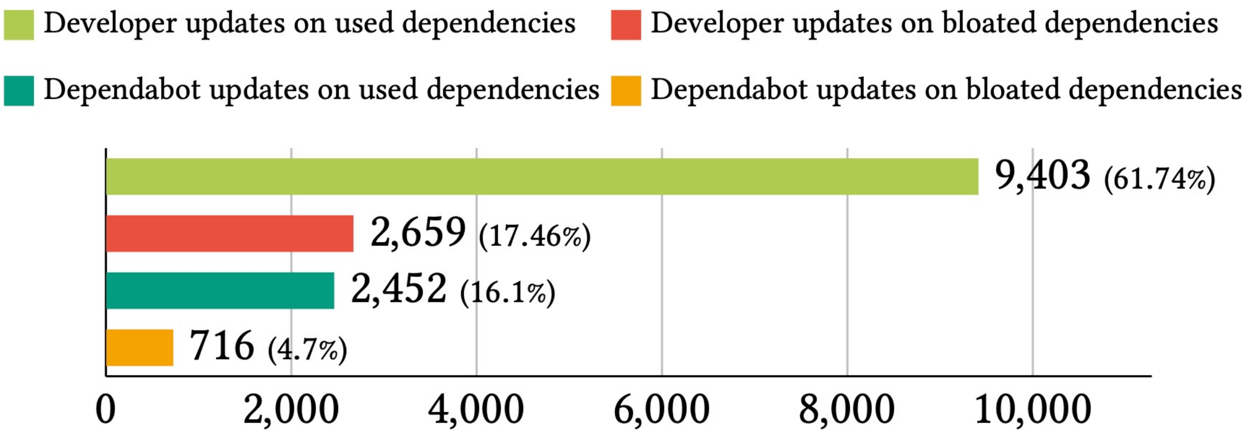
DO DEVELOPERS UPDATE BLOATED DEPENDENCIES?



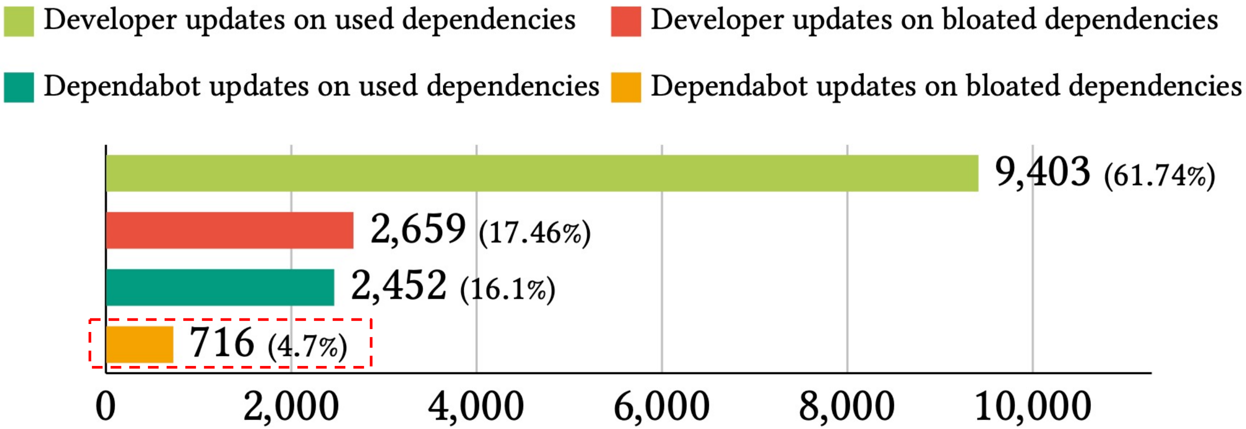
DO DEVELOPERS UPDATE BLOATED DEPENDENCIES?



DO DEVELOPERS UPDATE BLOATED DEPENDENCIES?



DO DEVELOPERS UPDATE BLOATED DEPENDENCIES?



SUMMARY OF 2nd CONTRIBUTION

SUMMARY OF 2nd CONTRIBUTION

- **The usage status of dependencies is mostly constant over time**
  - It is safe to debloat dependencies (> 90% of dependencies do not change)

SUMMARY OF 2nd CONTRIBUTION

- **The usage status of dependencies is mostly constant over time**
  - It is safe to debloat dependencies (> 90% of dependencies do not change)
- **Developers often update bloated dependencies**
  - An unnecessary maintenance effort due to the lack of tools

SUMMARY OF 2nd CONTRIBUTION

- **The usage status of dependencies is mostly constant over time**
  - It is safe to debloat dependencies (> 90% of dependencies do not change)
- **Developers often update bloated dependencies**
  - An unnecessary maintenance effort due to the lack of tools
- **Some dependency updates are suggested by Dependabot**
  - First empirical evidence of false alarms related to dependency management caused by bots

3rd CONTRIBUTION

JDBL: Trace-based Debloat for Java Bytecode

Trace-based Debloat for Java Bytecode

César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry
KTH Royal Institute of Technology, Stockholm, Sweden
Email: {cesarsv, tdurieux, harrand, baudry}@kth.se

Abstract—Software bloat is code that is packaged in an application but is actually not used and not necessary to run the application. The presence of bloat is an issue for software security, for performance, and for maintenance. In recent years, several works have proposed techniques to detect and remove software bloat. In this paper, we introduce a novel technique to debloat Java bytecode through dynamic analysis, which we call trace-based debloat. We have developed JDBL, a tool that automates the collection of accurate execution traces and the debloating process. Given a Java project and a workload, JDBL generates a debloated version of the project that is syntactically correct and preserves the original behavior, modulo the workload. We evaluate the feasibility and the effectiveness of trace-based debloat with 395 open-source Java libraries for a total 10M+ lines of code. We demonstrate that our approach significantly reduces the size of these libraries while preserving the functionalities needed by their clients.

Index Terms—Software Bloat, Dynamic Analysis, Program Specialization, Build Automation, Software Maintenance

1 Introduction

Software systems have a natural tendency to grow over time, both in terms of size and complexity [1], [2], [3]. A part of this growth comes with the addition of new features or different types of patches. Another part is due to the potentially useless code that accumulates over time. This phenomenon, known as software bloat, is becoming more prevalent with the emergence of large software frameworks [4], [5], [6], and the widespread practice of code reuse [7], [8]. Software debloat consists of automatically removing unnecessary code [9]. This poses several challenges for code analysis and transformation: determine the bloated parts of the software system [10], [11], [12], remove this part while keeping the integrity of the system, produce a debloated version of the system that can still run and provide useful features. In this context, the problem of effectively and safely debloating real-world applications remains a long-standing software engineering endeavor.

Previous works on software debloat have proposed different techniques, each of them tailored to a specific language. Significant efforts have been devoted to software debloat for C/C++ executable binaries. In this context, debloat starts from a program in which dependencies are compiled and linked statically [13], [14], [12]. Debloat approaches for Java are scarce in the literature, and rely on static analysis to detect unreachable code [15], [16]. These techniques are challenged by the dynamic features of the language, such as type-induced dependencies [17], dynamic class loading [18], and reflection [19]. In addition, static analysis techniques are conservative and do not remove *unused code* [20], [16], i.e., the parts of an application that can be reached statically but are not executed at run-time, within a specific period, in a production environment.

In this paper, we propose a novel software debloat technique to remove unused Java bytecode: *trace-based debloat*. The core novelty consists of steering the debloat process with information obtained from the collection of execution traces. This technique aims to capture bytecode usage information by monitoring the dynamic behavior of the system. Its automatable nature allows us to scale the debloat technique to large and diverse software projects, without any additional configuration. We implement this approach in a tool called JDBL, the Java DeBLoat tool designed to debloat Java projects configured to build with Maven.

JDBL is the first software tool to debloat Java bytecode that combines trace collection, bytecode removal, and build validation. JDBL is designed around three debloat phases. First, JDBL addresses the challenge of spotting unnecessary code while keeping the program cohesive. It leverages diverse code coverage tools to collect a set of accurate execution traces for debloating. This process involves executing the Maven project with an existing workload and monitoring its behavior at run-time through dynamic analysis. Second, JDBL modifies the bytecode to remove unnecessary code, i.e. code that has not been traced in the first phase. Bytecode removal is performed on the project as well as on the whole tree of third-party dependencies. Third, JDBL validates the debloat by executing the workload and verifying that the debloated project preserves the behavior of the original project.

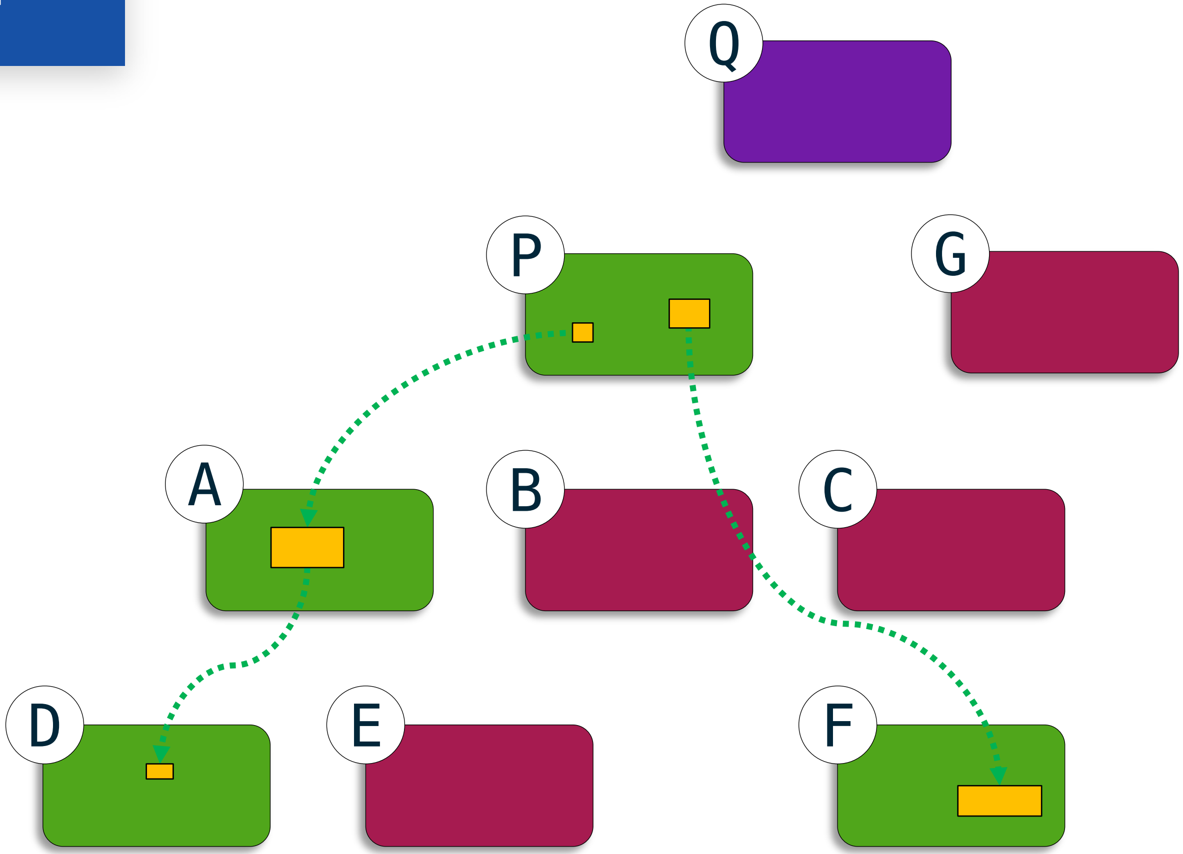
We run JDBL on a curated benchmark of 395 open-source Java libraries, to evaluate its correctness and effectiveness. Our results show that JDBL is capable of automatically debloating 311 (78.7 %) real-world Java libraries, and preserving the correctness of 220 (70.7 %) of these libraries. We provide quantitative evidence of the massive presence of unnecessary code in software artifacts: 62.2 % of classes in the libraries are bloated. The removal of this bloated code significantly reduces bytecode size: JDBL saves 68.3 % of the libraries' disk space, which represents a mean reduction of 25.8 % per library.

In order to further validate the relevance of trace-based debloat, we perform a second set of experiments with projects that use the libraries that we debloated with JDBL. This is the first time that the debloat results are not only evaluated with respect to the debloated subjects, but also on their clients. The goal is to demonstrate that JDBL produces debloated artifacts that still provide relevant functionalities. Our experiment shows firstly that the compilation of 957/1,001 (95.6 %) clients are not affected by the debloat of the library. Secondly, that the behavior of the test suite of 229/283 (80.9 %) clients is preserved.

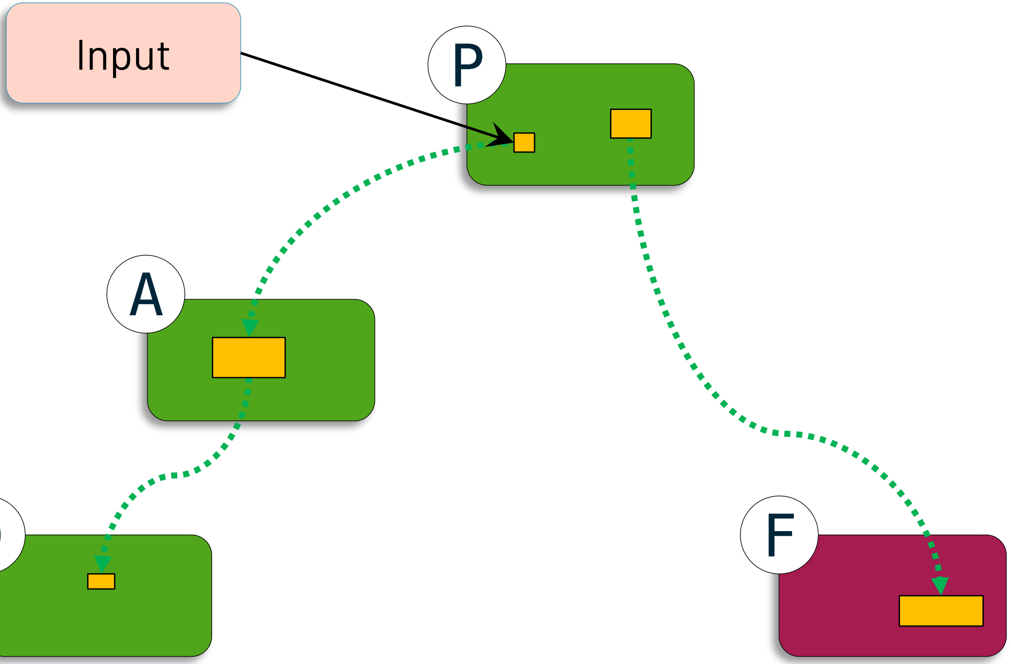
In summary, this paper makes the following contributions:

- The conceptual foundation of trace-based debloat for Java: a practical approach to debloat software through the collection of execution traces.

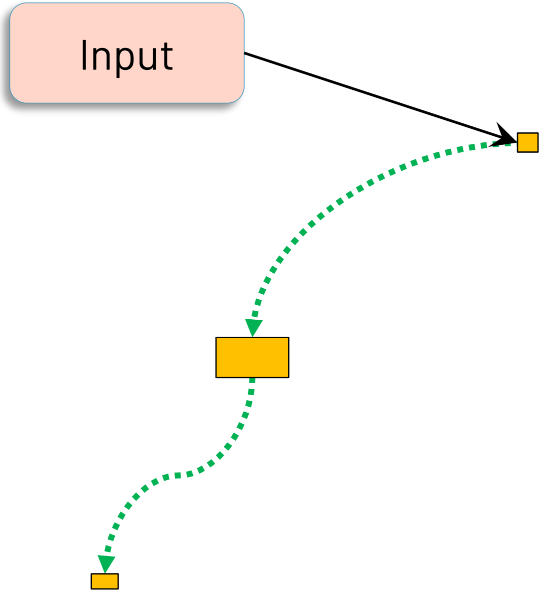
PROBLEM



PROBLEM

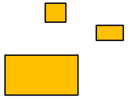


PROBLEM

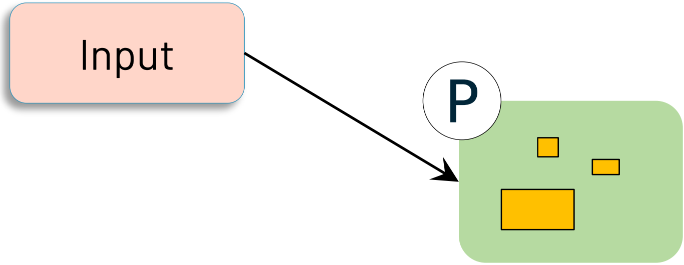


PROBLEM

PROBLEM

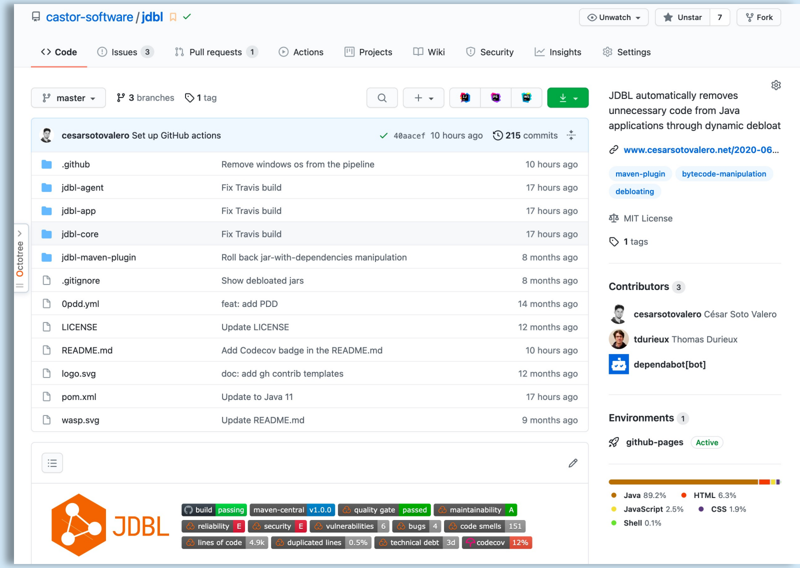


PROBLEM



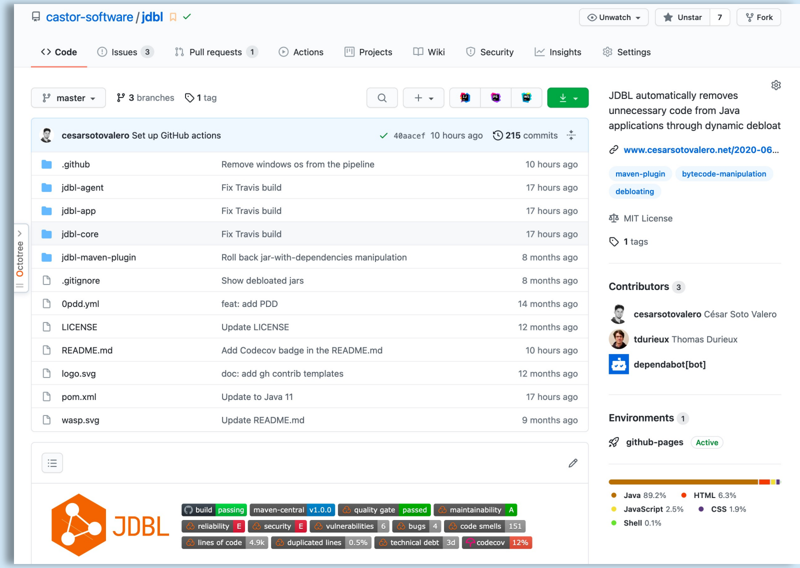
JDBL TOOL

https://github.com/castor-software/jdbl



JDBL TOOL

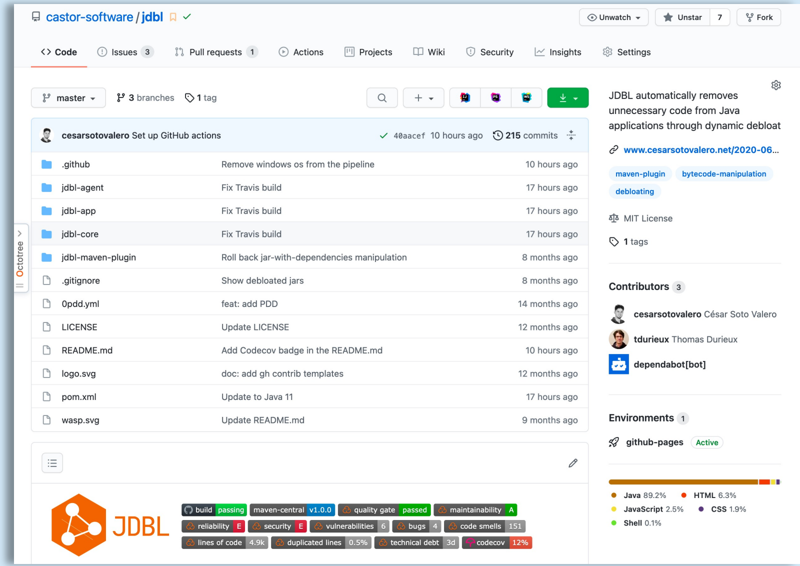
https://github.com/castor-software/jdbl



Relies on dynamic analysis to collect execution traces at runtime

JDBL TOOL

https://github.com/castor-software/jdbl

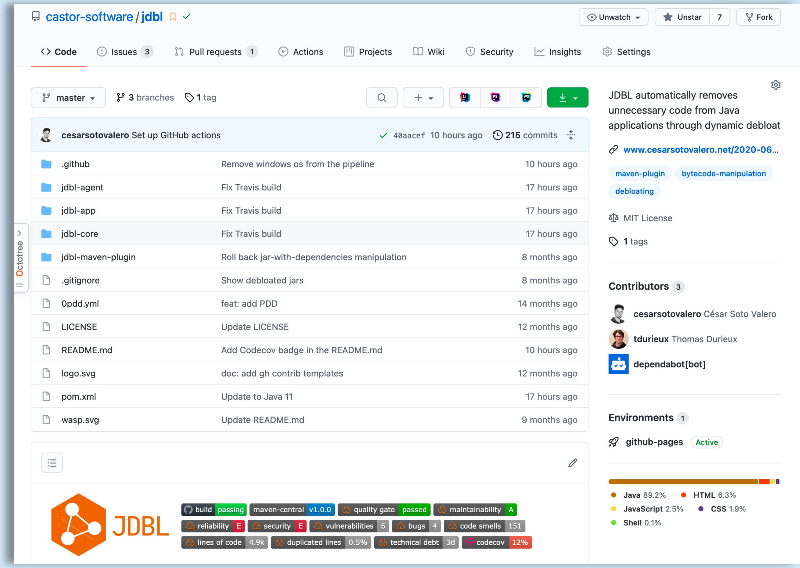


Relies on dynamic analysis to collect execution traces at runtime

Automatically remove unused classes and methods

JDBL TOOL

https://github.com/castor-software/jdbl



- Relies on dynamic analysis to collect execution traces at runtime
- Automatically remove unused classes and methods
- Package the debloated application

APPROACH

JDBL

APPROACH

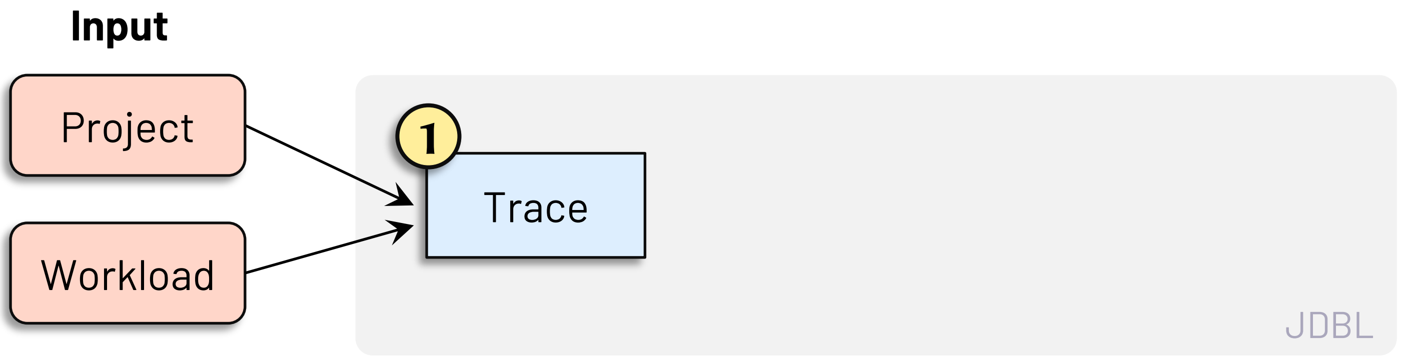
Input

Project

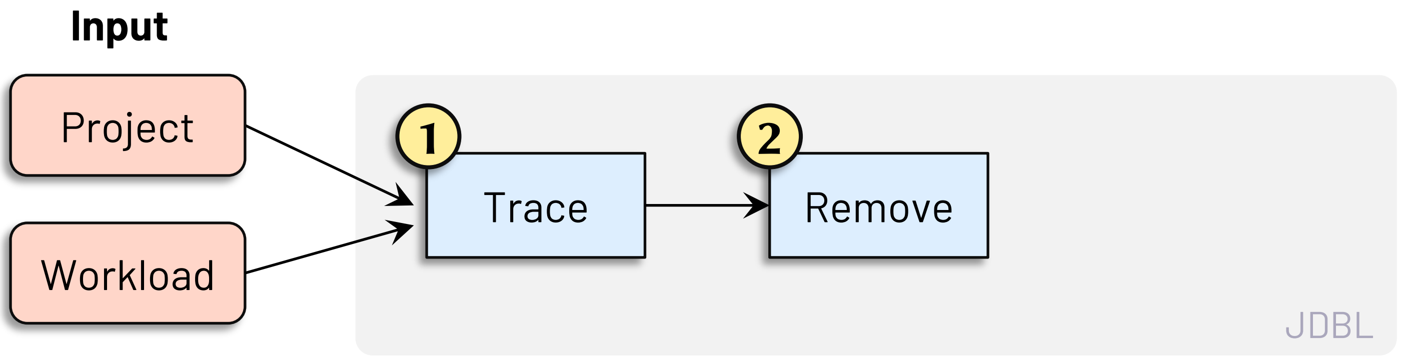
Workload

JDBL

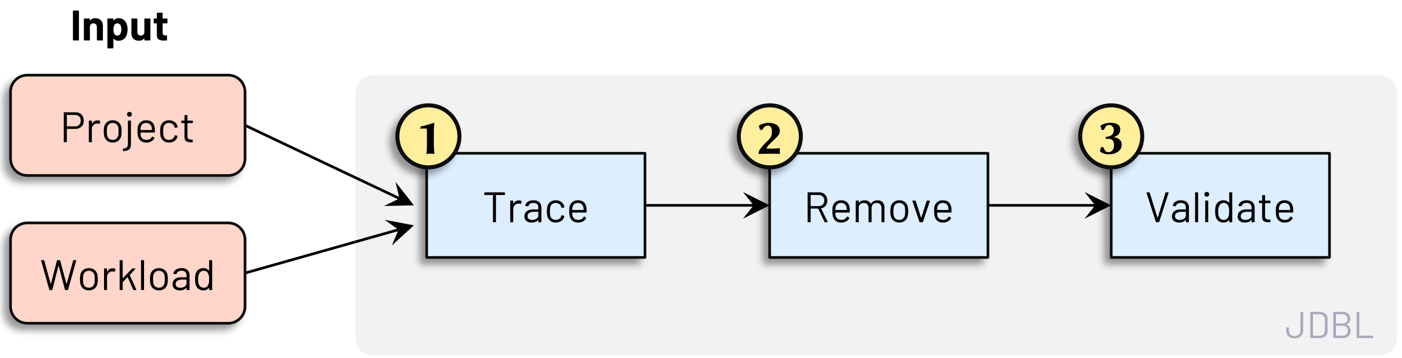
APPROACH



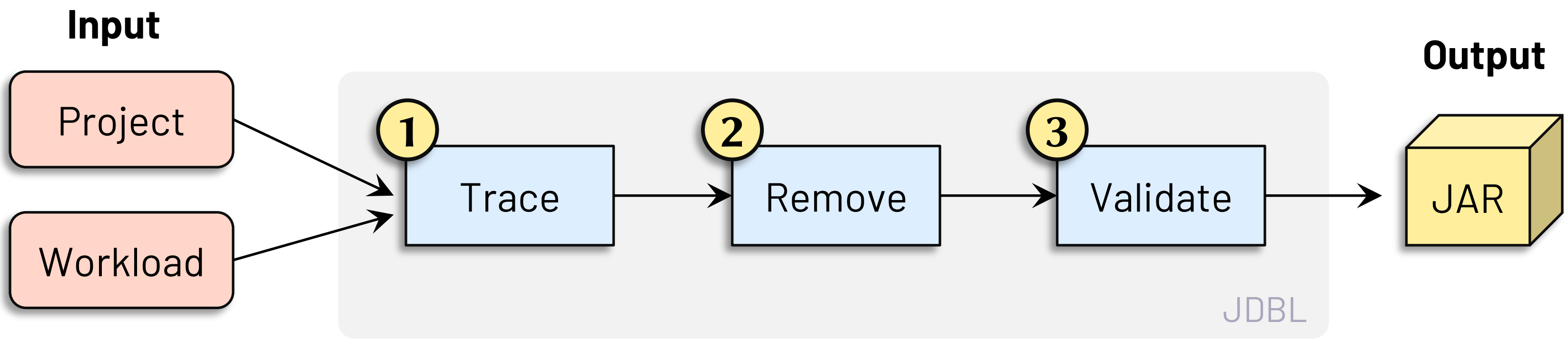
APPROACH



APPROACH

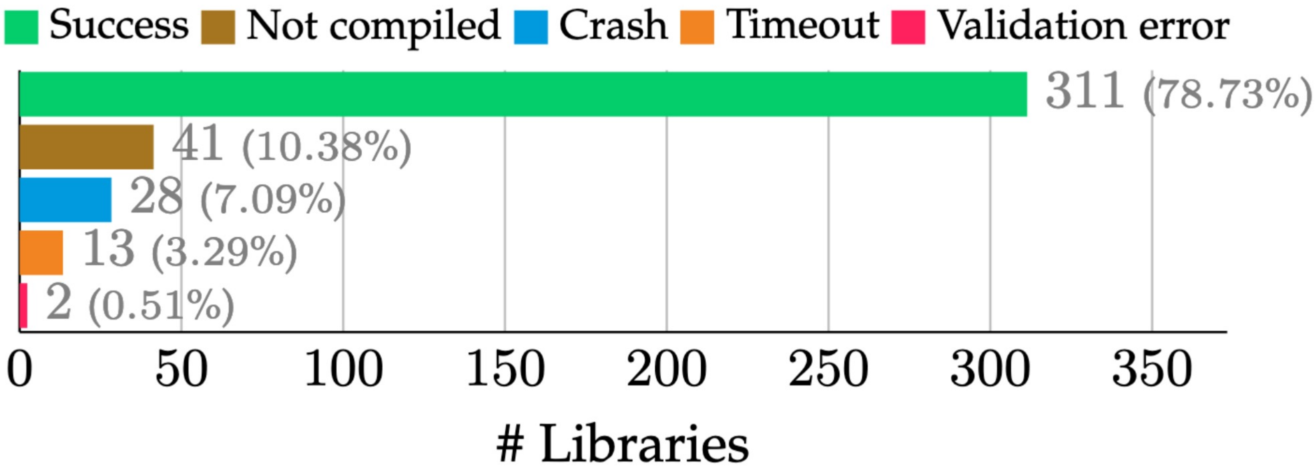


APPROACH


Input
Project
Workload
1
Trace
2
Remove
3
Validate
JDBL
Output
JAR

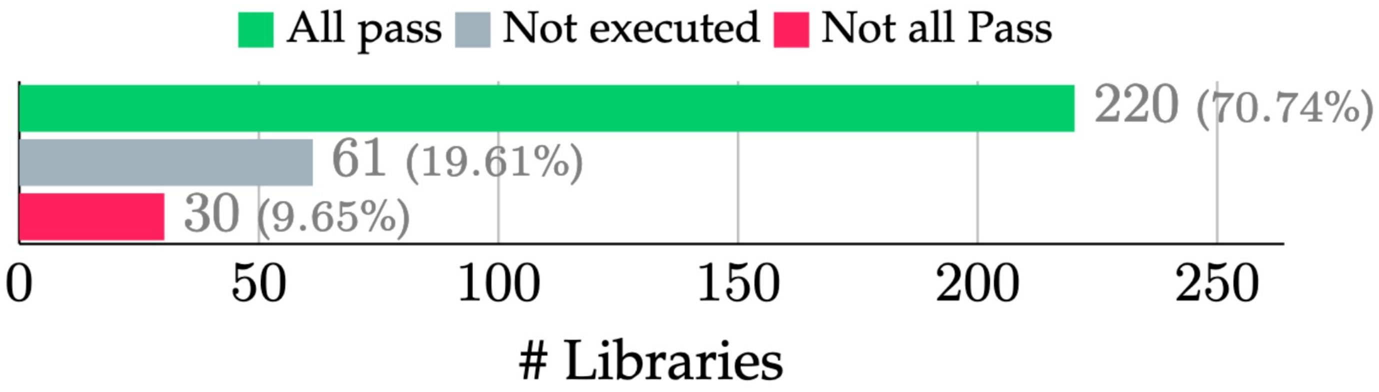
CAN JDBL DEBLOAT AUTOMATICALLY?

CAN JDBL DEBLOAT AUTOMATICALLY?



IS THE BEHAVIOUR PRESERVED?

IS THE BEHAVIOUR PRESERVED?



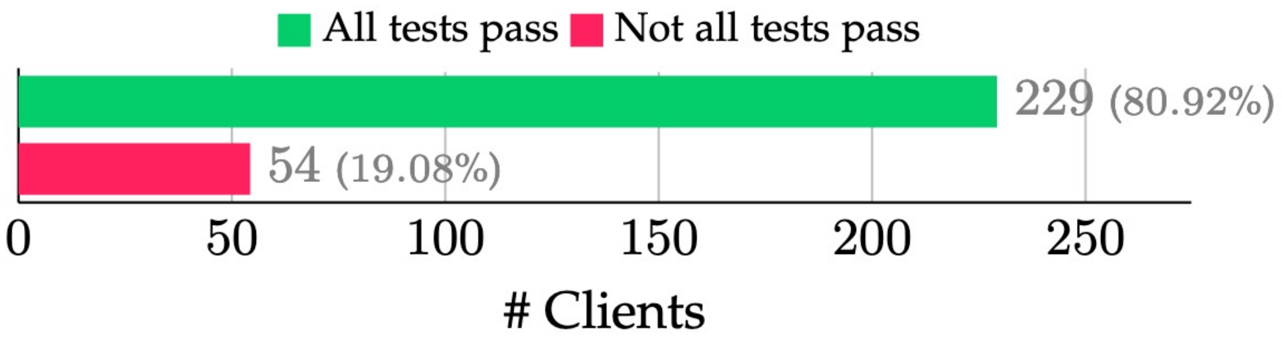
WHAT IS THE BENEFIT?

WHAT IS THE BENEFIT?

	Bloated (%)
Dependencies	52/254 (20.5 %)
Classes	75,273/121,055 (62.2 %)
Methods	505,268/829,015 (60.9 %)

ARE THE CLIENTS AFFECTED?

ARE THE CLIENTS AFFECTED?



SUMMARY OF 3rd CONTRIBUTION

SUMMARY OF 3rd CONTRIBUTION

- **Trace-based debloat is doable**
  - > 78% successfully debloated libraries

SUMMARY OF 3rd CONTRIBUTION

- **Trace-based debloat is doable**
  - > 78% successfully debloated libraries
- **Debloated libraries preserve the original behaviour**
  - > 70% libraries are not affected

SUMMARY OF 3rd CONTRIBUTION

- **Trace-based debloat is doable**
  - > 78% successfully debloated libraries
- **Debloated libraries preserve the original behaviour**
  - > 70% libraries are not affected
- **Debloated libraries are**
  - > 50% smaller than the original

SUMMARY OF 3rd CONTRIBUTION

- **Trace-based debloat is doable**
  - > 78% successfully debloated libraries
- **Debloated libraries preserve the original behaviour**
  - > 70% libraries are not affected
- **Debloated libraries are**
  - > 50% smaller than the original
- **Library clients preserve the original behaviour**
  - > 80% clients are not affected

LESSONS LEARNED

LESSONS LEARNED

LESSONS LEARNED

- **Debloat is hard in practice**
  - Determining what is actually used is not trivial
  - Static + Dynamic analysis may help

LESSONS LEARNED

- **Debloat is hard in practice**
  - Determining what is actually used is not trivial
  - Static + Dynamic analysis may help
- **Debloat is relevant**
  - Package ecosystems are bloated
  - Developers are willing to debloat software
  - More tools are needed for this purpose

FUTURE WORK

FUTURE WORK

- End-to-end software debloat

FUTURE WORK

- End-to-end software debloat
- Debloat containers

FUTURE WORK

- End-to-end software debloat
- Debloat containers
- Debloat specific features

FUTURE WORK

- End-to-end software debloat
- Debloat containers
- Debloat specific features
- Debloat test suites

PHD PROGRESS

PAPERS DIRECTLY RELATED

1. César Soto-Valero, Thomas Durieux, Nicolas Harrand, Benoit Baudry. **Trace-based Debloat for Java Bytecode** [Submitted to TSE]
2. César Soto-Valero, Thomas Durieux, Benoit Baudry. **A Longitudinal Analysis of Bloated Java Dependencies** [Submitted to FSE]
3. Thomas Durieux, César Soto-Valero, Benoit Baudry. **DUETS: A Dataset of Reproducible Pairs of Java Library-Clients** [MSR'21]
4. César Soto-Valero, Nicolas Harrand, Martin Monperrus, Benoit Baudry. **A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem** [EMSE'20]
5. César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, Benoit Baudry. **The Emergence of Software Diversity in Maven Central** [MSR'19]

Full list: https://www.cesarsotovalero.net/publications

OTHER PAPERS

1. Nicolas Harrand, Amine Benelallam, César Soto-Valero, Olivier Barais, Benoit Baudry. **Analyzing 2.3 Million Maven Dependencies to Reveal an Essential Core in APIs** [Submitted to JSS]
2. Gustaf Halvardsson, Johanna Peterson, César Soto-Valero, Benoit **Baudry. Interpretation of Swedish Sign Language using Convolutional Neural Networks and Transfer Learning** [SNCS'21]
3. Nicolas Harrand, César Soto-Valero, Martin Monperrus, Benoit Baudry. **Java Decompiler Diversity and its Application to Meta-decompilation** [JSS'20]
4. Raúl Reina, David Barbado, César Soto-Valero, José M. Sarabia and Alba Roldán. **Evaluation of the Bilateral Function in Para-athletes with Spastic Hemiplegia: a Model-based Clustering Approach** [JSAMS'20]
5. Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, Olivier Barais. **The Maven Dependency Graph: a Temporal Graph-based Representation of Maven Central** [MSR'19]
6. Nicolas Harrand, César Soto-Valero, Martin Monperrus, Benoit Baudry. **The Strengths and Behavioral Quirks of Java Bytecode Decompilers** [SCAM'19]
7. César Soto-Valero, Miguel Pic. **Assessing the Causal Impact of the 3-point Per Victory Scoring System in the Competitive Balance of LaLiga** [IJCSS'19]
8. César Soto-Valero, Yohan Bourcier, Benoit Baudry. **Detection and Analysis of Behavioral T-patterns in Debugging Activities** [MSR'18]

Full list: https://www.cesarsotovalero.net/publications

TEACHER ASSISTANT

1. **DD2482 Automated Software Testing and DevOps**, worked with Martin Monperrus & Benoit Baudry at KTH, Spring 2021
2. **WASP Software Engineering and Cloud Computing**, worked with Martin Monperrus & Benoit Baudry at KTH, Spring 2021
3. **DD2480 Software Engineering Fundamentals**, worked with Cyrille Artho at KTH, Spring 2021
4. **DD1369 Software Engineering in Project Form**, worked with Dena Hussain at KTH, Fall 2020
5. **DD2460 Software Safety and Security**, worked with Cyrille Artho at KTH, Spring 2020
6. **DD2482 Automated Software Testing and DevOps**, worked with Martin Monperrus & Benoit Baudry at KTH, Spring 2020
7. **DM1590 Machine Learning for Media Technology**, worked with Bob Sturm at KTH, Spring 2020
8. **DA2210 Introduction to the Philosophy of Science and Research Methodology for Computer Scientists**, worked with Linda Kann at KTH, Fall 2019
9. **WASP Software Engineering and Cloud Computing**, worked with Martin Monperrus & Benoit Baudry at KTH, Spring 2019
10. **ID2211 Data Mining, Basic Course**, worked with Sarunas Girdzijauskas at KTH, Spring 2019

Full list: https://www.cesarsotovalero.net/service

57 CREDITS

10 courses completed

57 CREDITS

10 courses completed

3 SUPERVISIONS

2 BSc + 1 MSc

57 CREDITS

10 courses completed

3 SUPERVISIONS

2 BSc + 1 MSc

8 PAPERS REVIEWED

3 as primary reviewer + 5 as sub-reviewer

57 CREDITS

10 courses completed

3 SUPERVISIONS

2 BSc + 1 MSc

8 PAPERS REVIEWED

3 as primary reviewer + 5 as sub-reviewer

99 CITATIONS

Slow and steady wins the race

3 PROJECTS

DepAnalyzer + DepClean + JDBL

3 PROJECTS

DepAnalyzer + DepClean + JDBL

10+ TRIPS

4 Countries

3 PROJECTS

DepAnalyzer + DepClean + JDBL

10+ TRIPS

4 Countries

3 PROJECTS

DepAnalyzer + DepClean + JDBL

10+ TRIPS

4 Countries

50+ MERGED PRs

Still low, more to come!

3 PROJECTS

DepAnalyzer + DepClean + JDBL

10+ TRIPS

4 Countries

50+ MERGED PRs

Still low, more to come!

12 PRESENTATIONS

e.g., SL, FOSDEM'21

1 BABY

The greatest challenge!

THANKS!

Any questions?

You can find me at:

- cesarsv@kth.se
- https://www.cesarsotovalero.net