ASE 2020
ONLINE CONFERENCE
21-25 SEPTEMBER 2020

CASTLE Research Group
Code AnalysiS, Testing, and LEarning

MockSniffer

Characterizing & Recommending Mocking Decisions for Unit Tests

Hengcheng Zhu, Lili Wei, Ming Wen, Yepang Liu, Shing-Chi Cheung,
Qin Sheng, and Cui Zhou

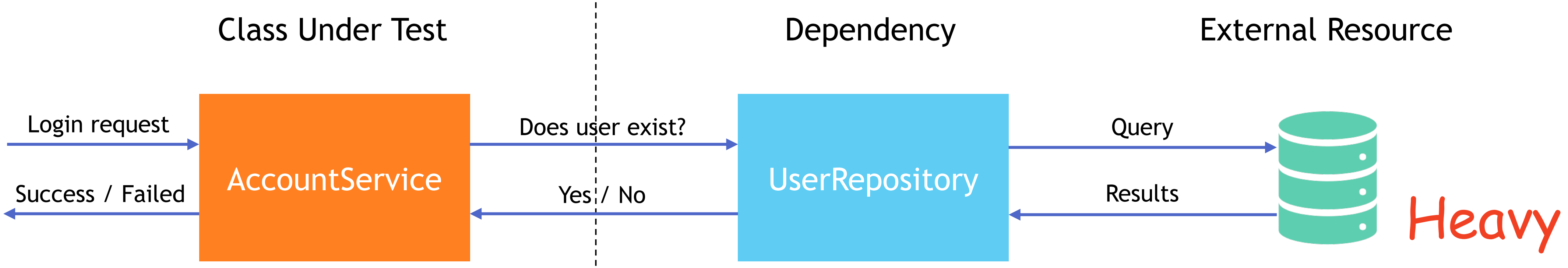
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

HUAZHONG UNIVERSITY OF SCIENCE & TECHNOLOGY

SUSTech Southern University of Science and Technology

WeBank

Testing a Class with Dependencies



3 Start testing

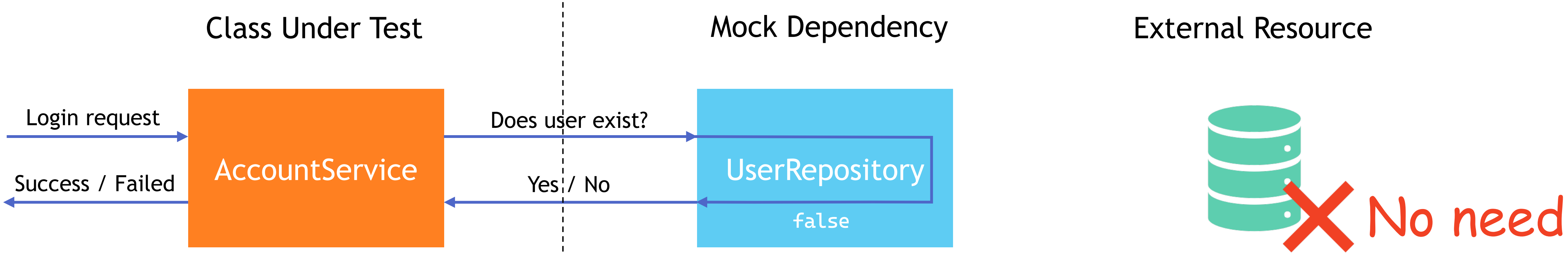
2 Initiate the dependency

1 Prepare a database

Developers' focus

Can this be simplified?

Mocking in Unit Tests



② Test the AccountService

① A mock UserRepository which
- Return false directly
- Without connecting to the database

Faster

Improper Mocking Decisions

Under-mocking

Did not mock a dependency that should be mocked

Inefficient test execution

Flaky tests [1]

Side effect to the environment

> "*The unit tests for the `camel-hazelcast` component use real `HazelcastInstance` objects, which is **very slow**. We should use mock objects instead to **speed up testing**.*"
>
> -- Issue 6826, Camel

[1] Luo et al. An empirical analysis of flaky tests. [FSE 2014]

Improper Mocking Decisions (cont.)

Over-mocking

Mock the dependencies that should not be mocked

"*Mockito is used in `SchedulerTestUtils` to mock `ExecutionVertex` and Execution for testing. It fails to mock every getter so that other tests use it may encounter NPE issues.*"

-- Issue 16300, Flink

Flink

Increase development cost

False alarms

Mocking Decisions Are Not Easy to Make

“13% of the mocks are introduced later in the lifetime of the test class.
17% of these mocks are removed afterwards.

Spadini et al. **Mock Objects for Testing Java Systems: Why and How Developers Use Them, and How They Evolve** [EMSE 2019]”

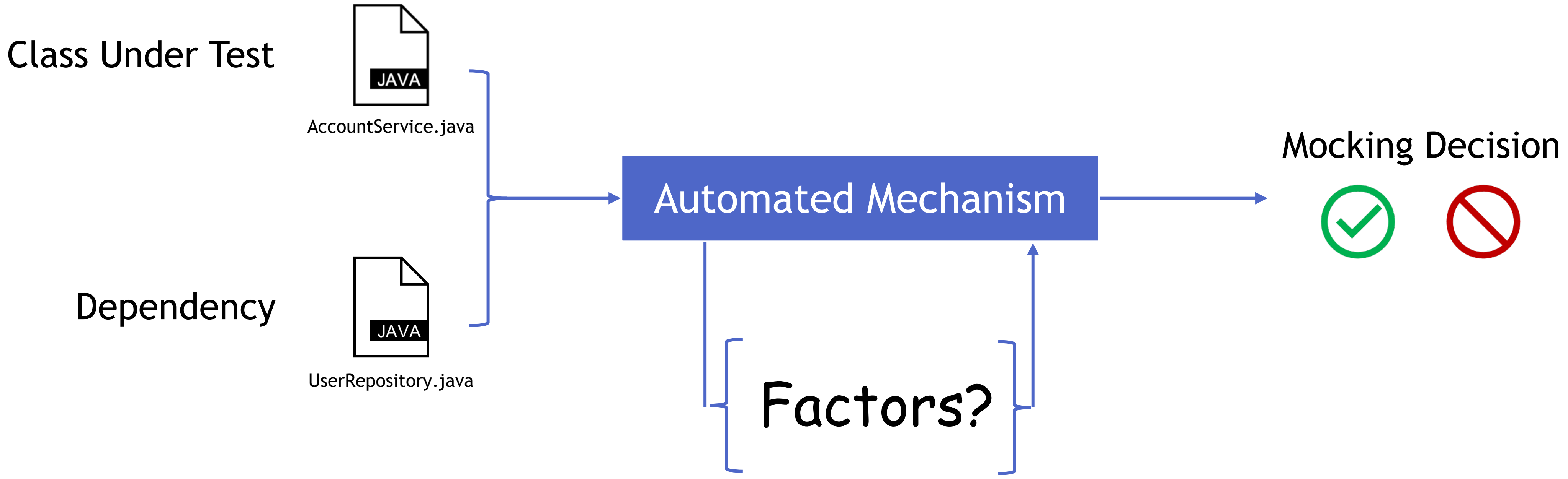
“We highlighted the need to automate the process of identifying APIs that need to be mocked and dependencies between the identified APIs to ease the process of testing.

Marri et al. **An Empirical Study of Testing File-System-Dependent Software with Mock Objects** [AST 2009]”

We Aim to Answer…

Which dependencies
should be mocked?

Research Goal



Existing Findings

“Software testers usually mock only a small number and portion of software dependencies. Software testers tend to mock source code classes than libraries.

Mostafa et al. **An Empirical Study on the Usage of Mocking Frameworks in Software Testing** [QSIC 2014]”

“Classes that deal with external resources are often mocked.
Classes that are slow and complex to setup are good candidates to be mocked.

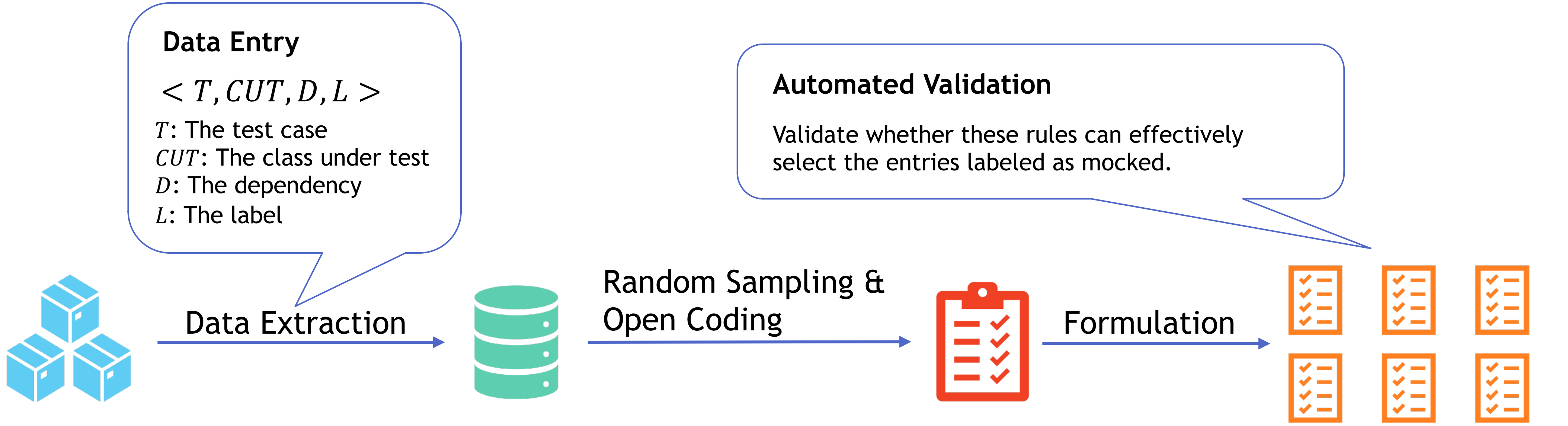
Spadini et al. **Mock Objects for Testing Java Systems: Why and How Developers Use Them, and How They Evolve** [EMSE 2019]”

High-level, qualitative

Cannot automate

Empirical Study

Goal: Find code-level characteristics (rules) of the mocked dependencies



Projects

4 large-scale, open-source projects

Dataset

50k test cases
354k data entries

Observations

5 observations

Code-level Rules

10 rules

Empirical Findings – API Usage

Classes related to environment or concurrency are often mocked.

Rule 1.1: Referencing environment-dependent or concurrent classes.

Networking, disk I/O, database, threading, access control,
e.g., `File`, `InetAddress`, `ExecutorService`

Rule 1.2: Encapsulating external resources.

e.g., implements `Closable`, `AutoClosable`

Rule 1.3: Calling `synchronized` methods.

Empirical Findings - Interactions

Dependencies affecting the runtime control flows of methods in CUTs are often mocked.

Class Under Test (in project Camel)

```
if(endpointConfig.isOverWrite()){
  oStream.info.getFileSystemn().delete(...);
} else {
  throw new RuntimeCamelException(...);
}
```

true to cover the if branch

false to cover the else branch

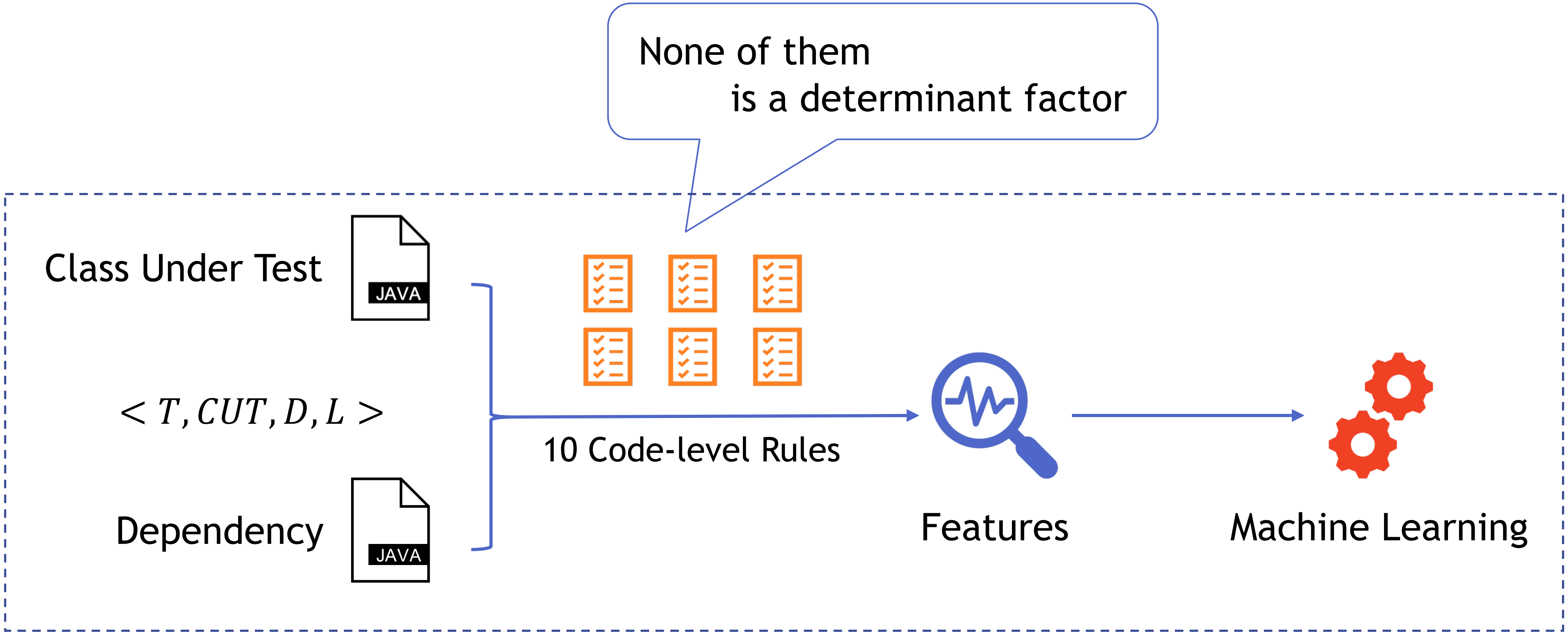
Test Script

```
when(endpointConfig.isOverWrite())
    .thenReturn(false);
```

Rule 4.1: Affecting CUT's runtime control flows via return values.

Rule 4.2: Affecting CUT's runtime control flows via exceptions.

Combining the Observations



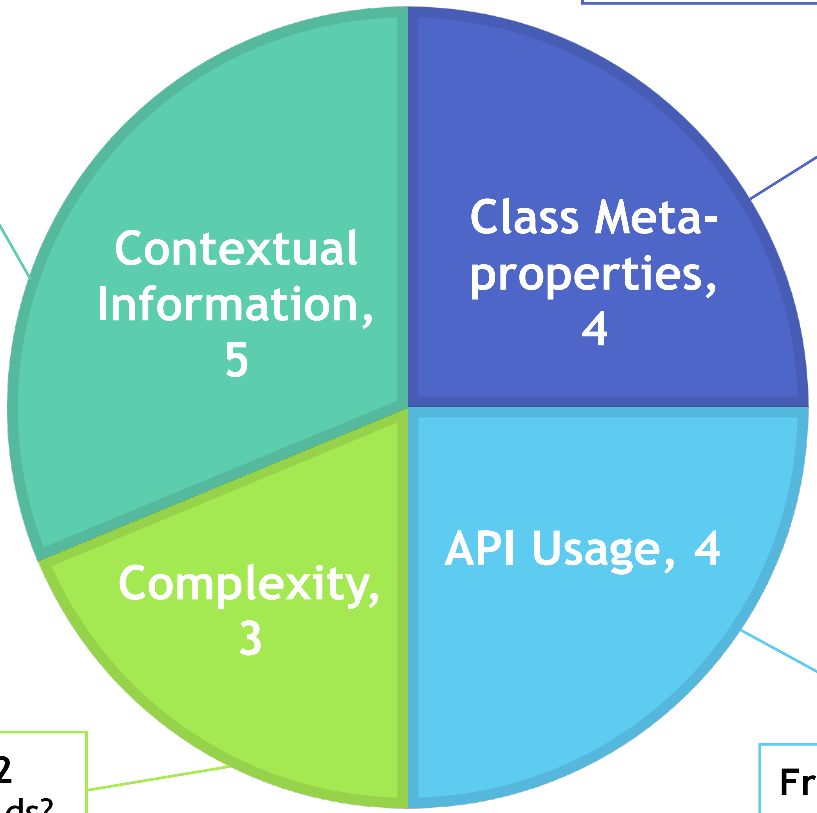
MockSniffer

Features & Algorithms

Features

From observation 4 & 5
e.g., Does it throw an exception caught by the CUT?

From observation 3 and existing work
e.g., Is it a JDK class?



From observation 2
e.g., How many fields?

From observation 1
e.g., How many call sites to a database API?

Algorithms

- **Gradient Boosting (Default)**
- Random Forest
- Ada Boosting
- Decision Tree
- Support Vector Machine
- Naïve Bayes

Evaluation Subjects

hadoop

OOZIE

Apache CXF

druid

APACHE STORM™

APACHE Camel

HIVE

DUBBO

APACHE HBASE

Flink

Research Questions

Effectiveness

1. Is MockSniffer more effective than existing strategies?
2. Does machine learning help?

Application

3. Potential application scenarios?
4. Performance in these scenarios?

Baselines

Baseline #1: Existing Heuristics

- Mock all the classes in the code base [2]
- Mock all the interfaces [3]
- Do not mock JDK classes [3]

Baseline #2: EvoSuite Mock List

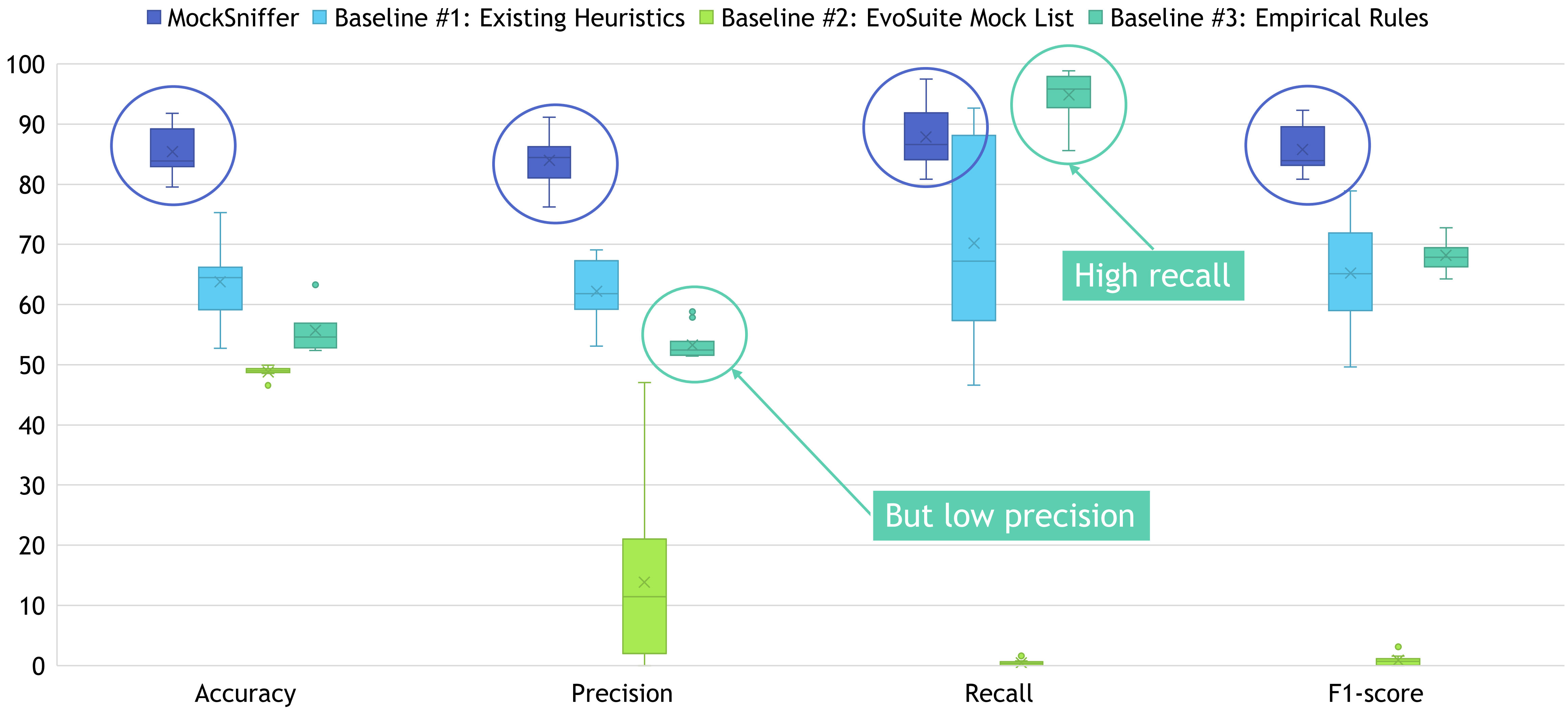
- Mock all the classes in the EvoSuite [4] mock list

Baseline #3: Empirical Rules

- Mock if any of the rules in our empirical study matches

[2] Mostafa et al. An Empirical Study on the Usage of Mocking Frameworks in Software Testing. [QSIC 2014]
[3] Spadini et al. Mock objects for testing java systems: Why and how developers use them, and how they evolve. [EMSE 2019]
[4] Fraser et al. EvoSuite: automatic test suite generation for object-oriented software. [FSE 2011]

MockSniffer vs. Baselines

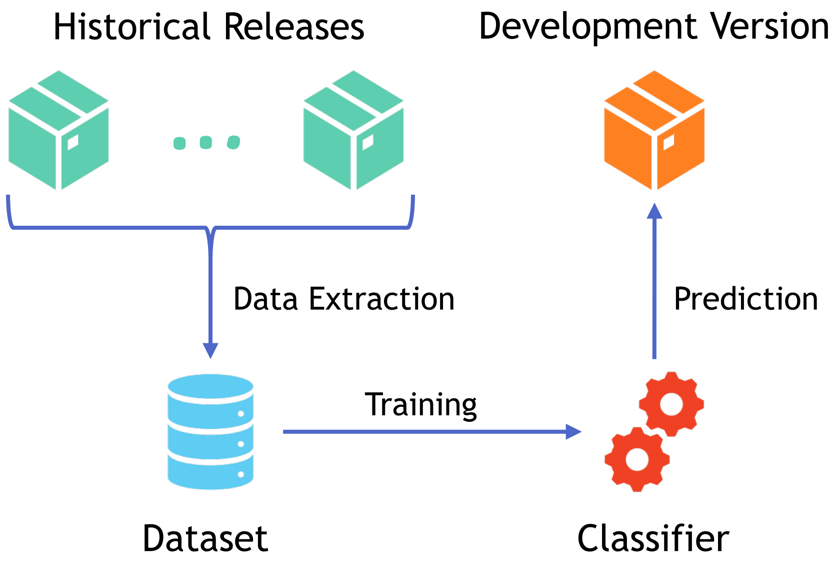


Application Scenarios

Cross-version Prediction

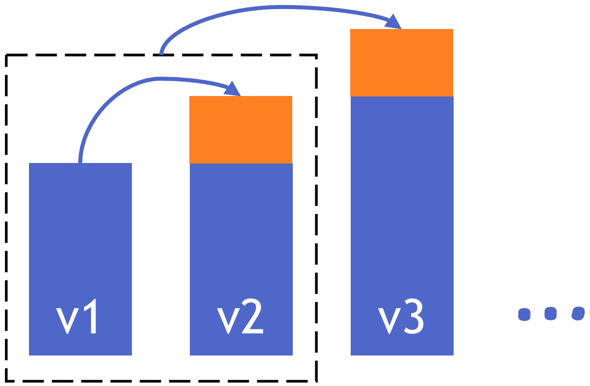
Real-world Scenario

Target: Mature Projects



Experiment Setup

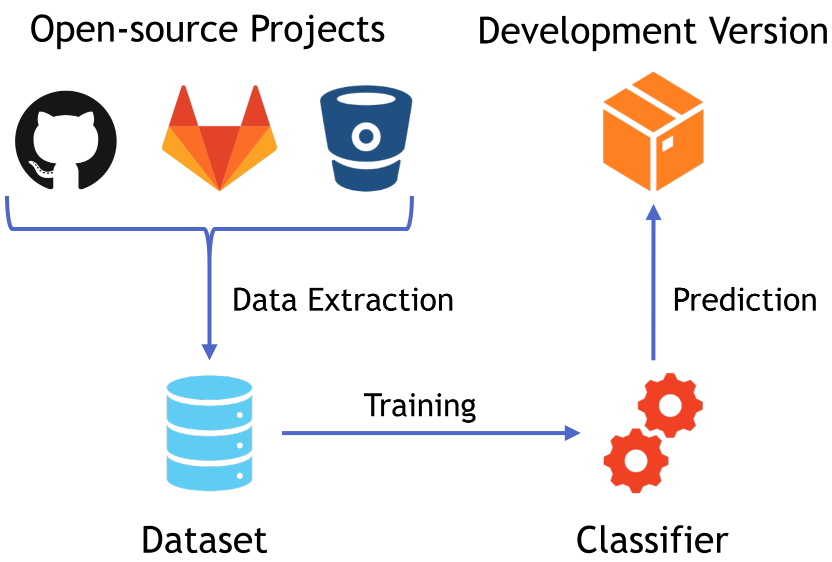
- Train with historical releases
- Predict on new data entries



Cross-project Prediction

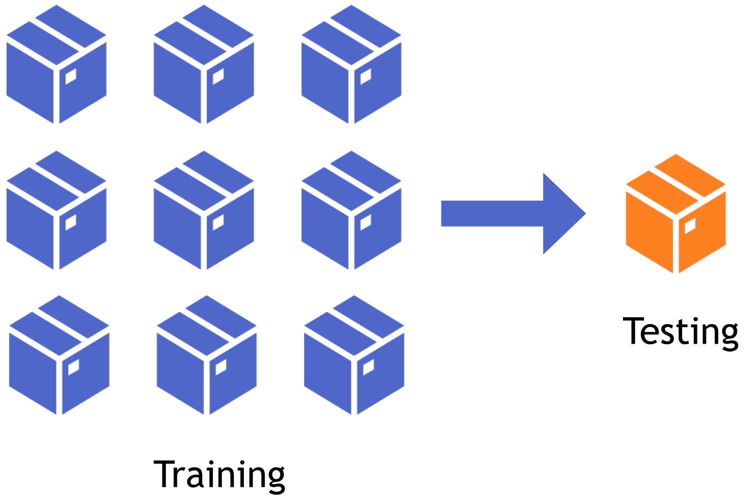
Real-world Scenario

Target: New Projects



Experiment Setup

- Train with 9 projects
- Predict on 1 project



Comparison of Two Scenarios

Cross-version Prediction

- Prec. **78.82%**, Recall 60.63% (avg.)
- More focus, higher precision
- Catch the similarities within the project

Cross-project Prediction

- Prec. 72.65%, Recall **67.92%** (avg.)
- More diverse, higher recall
- Can cover different mocking strategies

Recall in Oozie

Large changes took place from 4.x to 5.x

30.67% → 2.1x → 62.93%

Cross-version → Cross-project

Contribution

Mock Recommendation Technique

1st

Potential Application Scenarios

2

10

Code-level Characteristics

546k

Large-scale Dataset

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

SUSTech Southern University of Science and Technology

CASTLE Research Group

Code AnalysiS, Testing, and LEarning

HUAZHONG UNIVERSITY OF SCIENCE & TECHNOLOGY

WeBank

THANKS

Presented by Hengcheng Zhu
hzhuaq@connect.ust.hk

Artifacts available at
https://aka.henryhc.net/mocksniffer

MockSniffer: Characterizing and Recommending Mocking Decisions for Unit Tests

Hengcheng Zhu*
Southern University of Science and Technology
Shenzhen, China
zhuhc2016@mail.sustech.edu.cn

Lili Wei*
The Hong Kong University of Science and Technology
Hong Kong, China
lweiae@cse.ust.hk

Ming Wen
Huazhong University of Science and Technology
Wuhan, China
mwenaa@hust.edu.cn

Yepang Liu†
Southern University of Science and Technology
Shenzhen, China
liuyp1@sustech.edu.cn

Shing-Chi Cheung†
The Hong Kong University of Science and Technology
Hong Kong, China
scc@cse.ust.hk

Qin Sheng
WeBank Co Ltd
Shenzhen, China
entersheng@webank.com

Cui Zhou
WeBank Co Ltd
Shenzhen, China
cherryzzhou@webank.com

ABSTRACT

In unit testing, mocking is popularly used to ease test effort, reduce test flakiness, and increase test coverage by replacing the actual dependencies with simple implementations. However, there are no clear criteria to determine which dependencies in a unit test should be mocked. Inappropriate mocking can have undesirable consequences: under-mocking could result in the inability to isolate the class under test (CUT) from its dependencies while over-mocking increases the developers' burden on maintaining the mocked objects and may lead to spurious test failures. According to existing work, various factors can determine whether a dependency should be mocked. As a result, mocking decisions are often difficult to make in practice. Studies on the evolution of mocked objects also showed that developers tend to change their mocking decisions: 17% of the studied mocked objects were introduced sometime after the test scripts were created and another 13% of the originally mocked objects eventually became unmocked. In this work, we are motivated to develop an automated technique to make mocking recommendations to facilitate unit testing. We studied 10,846 test scripts in four actively maintained open-source projects that use mocked objects, aiming to characterize the dependencies that are mocked in unit testing. Based on our observations on mocking practices, we designed and implemented a tool, *MockSniffer*, to identify and recommend mocks for unit tests. The tool is fully automated and requires only the CUT and its dependencies as input. It leverages machine learning techniques to make mocking recommendations by holistically considering multiple factors that can affect developers' mocking decisions. Our evaluation of *MockSniffer* on ten open-source projects showed that it outperformed three baseline approaches, and achieved good performance in two potential application scenarios.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Software and its engineering** → **Software maintenance tools**; *Software testing and debugging*.

KEYWORDS

Mocking, unit testing, recommendation system, dependencies

ACM Reference Format:
Hengcheng Zhu, Lili Wei, Ming Wen, Yepang Liu, Shing-Chi Cheung, Qin Sheng, and Cui Zhou. 2020. MockSniffer: Characterizing and Recommending Mocking Decisions for Unit Tests. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416539

1 INTRODUCTION

Unit testing has been widely adopted to assure the quality of program units, namely classes, by testing them in isolation. In practice, a class under test (CUT) is commonly coupled with other classes in

*This work was conducted when Hengcheng Zhu was a visiting student at HKUST (The Hong Kong University of Science and Technology). The first two authors contributed equally to this work.
†Yepang Liu and Shing-Chi Cheung are corresponding authors.