When testing meets Code Review: Why and How Developers Review Tests

Davide Spadini, Mauricio Aniche,
Margaret-Anne Storey, Magiel Bruntink, Alberto Bacchelli

University of Victoria

University of Zurich[UZH]

When testing meets Code Review: Why and How Developers Review Tests

Davide Spadini, Mauricio Aniche,
Margaret-Anne Storey, Magiel Bruntink, Alberto Bacchelli

 **@DavideSpadini**

 **ishepard**


University of Victoria

TU Delft


University of Zurich[UZH]

Automated Detection of Test Fixture Strategies and Smells

Michaela Greiler, Arie van Deursen
Delft University of Technology
{m.s.greiler|arie.vanDeursen}@tudelft.nl

Margaret-Anne Storey
University of Victoria, BC, Canada
mstorey@uvic.ca

***Abstract*—Designing automated tests is a challenging task. One important concern is how to design test fixtures, i.e. code that initializes and configures the system under test so that it is in an appropriate state for running particular automated tests. Test designers may have to choose between writing in-line fixture code for each test or refactor fixture code so that it can be reused for other tests. Deciding on which approach to use is a balancing act, often trading off maintenance overhead with slow test execution. Additionally, over time, test code quality can erode and test smells can develop, such as the occurrence of overly general fixtures, obscure in-line code and dead fields. In this paper, we show that test smells related to fixture set-up occur in industrial projects. We present a static analysis technique to identify fixture related test smells. We implemented this test analysis technique in a tool, called TestHound, which provides reports on test smells and recommendations for refactoring the smelly test code. We evaluate the tool through three industrial case studies and show that developers find that the tool helps them to understand, reflect on and adjust test code.**

***Keywords*-test code comprehension; maintainability; test fixture; test smells; software testing; test code refactoring;**

I. Introduction

Modern software development practice dictates early and frequent (automated) testing. While automated test suites written by developers are helpful from a (continuous) integration and regression testing perspective, they lead to a substantial amount of test code. Like production code, test code needs to be maintained, understood, and adjusted, which can become very costly. The long term success of automated testing is highly influenced by the maintainability of the test code [14]. To support easier maintainability of a system, test methods should be clearly structured, well named and small in size [7]. The duplication of code across test methods should be avoided.

One important part of a test is the code that initializes the system under test (SUT), sets up all dependencies and puts the SUT in the right state to fulfill all preconditions needed to exercise the test. In line with Meszaros, we refer to this part of a test as the *test fixture* [14]. Developers can adopt several strategies for structuring their fixture code. The most straightforward option is to place the setup code directly in the test method, which we refer to as an *in-line setup*. A positive aspect of an *in-line* setup is the proximity of the setup code to the test itself. However, when several test methods require the same fixture, an *in-line* setup can lead to code duplication and high maintenance costs [5]. Also, configuring the SUT within the test method might hide the main purpose of the test and result in an *obscure test* [14].

An alternative approach is to place the setup code in helper methods that can be called by several test methods, which we refer to as a *delegate setup* [14]. With a delegate setup, the developer has to make sure the right methods are invoked at the right time (e.g. as a first statement in a test method).

In today's testing frameworks, such as the widely used xUnit family, there is a dedicated mechanism to manage setup code invocations [1], [8]. Therefore, helper-methods containing the setup code can be marked (e.g. using annotations or naming conventions) as specific setup methods, which we refer to as an *implicit setup*.[1] The advantage of an *implicit setup* is that the framework takes care of invoking the setup code at a certain point in time and for a specific group of tests, but also that the methods are explicitly marked as setup which helps with code comprehension. Often, *implicit* setups are invoked either before each test within a class, or once before all the tests within a class. One main drawback of this approach is that the tests grouped together (i.e. within one class) should have similar needs in the test fixture. Otherwise, tests might only access (small) portions of a broader fixture, which can lead to slow tests and maintenance overhead.

During the evolution of test code, developers have to make conscious decisions about how to set up the test fixture and adjust their fixture strategies, otherwise they end up with poor solutions to recurring implementation and design problems in their test code, so-called *test smells* [5]. Unfortunately, until now, no support has been made available to developers during the analysis and adjustment of test fixtures.

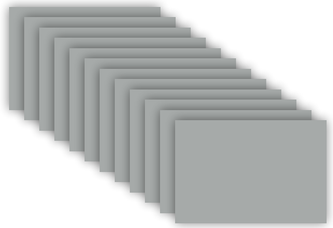
To address this shortcoming, we developed a technique that automatically analyzes test fixtures to detect fixture-related smells and guides improvement activities. We implemented this technique in *TestHound*, a tool for static fixture analysis. We evaluate our technique in a mixed methods research approach. First, we analyzed the test fixtures of three industry-strength software systems. Second, we eval-

[1] For example, in the JUnit frameworks methods can be either named *setUp()* or marked with annotations such as *@Before* or *@BeforeClass*.


Code review

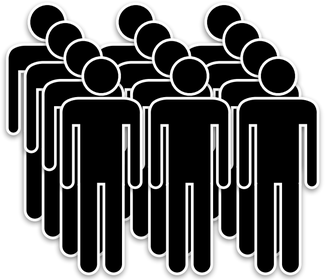
Research Questions

We collected Code Reviews from Gerrit



3 OSS: Eclipse, Qt, OpenStack

12 interviews



RQ1: **How** rigorously is test code reviewed?

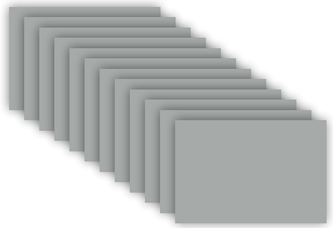
RQ2: **What** do reviewers discuss in test code reviews?

RQ3: Which **practices** do reviewers follow for test files?

RQ4: What **problems** and **challenges** do developers face when reviewing tests?

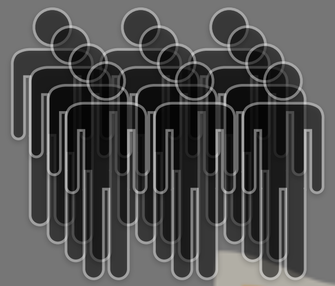
Research Questions

We collected Code Reviews from Gerrit



3 OSS: Eclipse, Qt, OpenStack

12 interviews



RQ1: **How** rigorously is test code reviewed?

RQ2: **What** do reviewers discuss in test code reviews?

RQ3: Which **practices** do reviewers follow for test files?

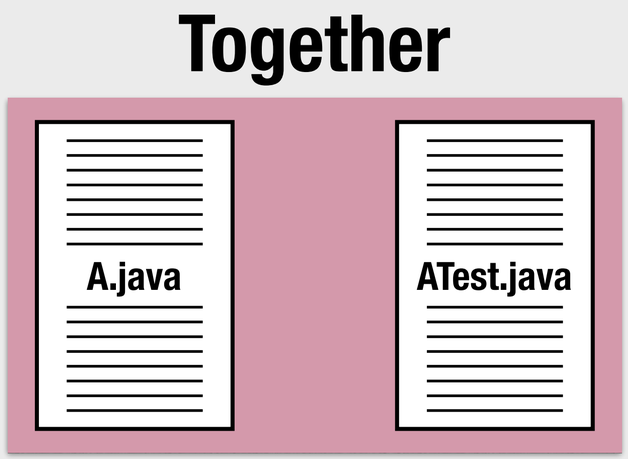
RQ4: What **problems** and **challenges** do developers face when reviewing tests?

RQ1: How rigorously is test code reviewed? — Method

	# of prod. files	# of test files	# of code reviews	# of reviewers	# of comments
Eclipse	215k	19k	60k	1k	95k
Openstack	75k	48k	199k	9k	894k
Qt	158k	8k	114k	1k	19k
Total	**450k**	**77k**	**374k**	**12k**	**1,010k**

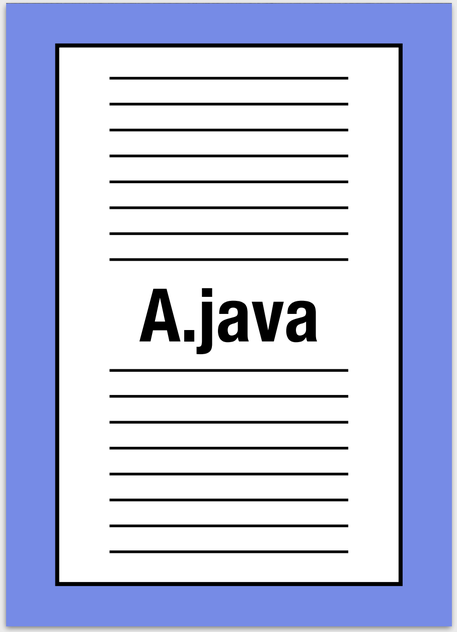
RQ1: How rigorously is test code reviewed? — Results

RQ1: How rigorously is test code reviewed? — Results



RQ1: How rigorously is test code reviewed? — Results

Production alone

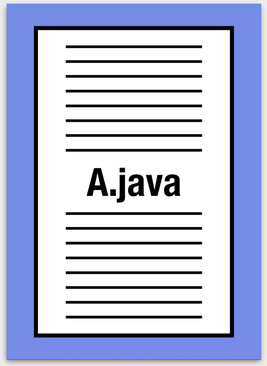


Together

RQ1: How rigorously is test code reviewed? — Results

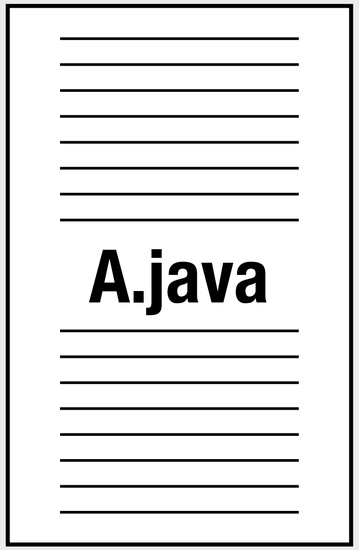
Together

Production alone



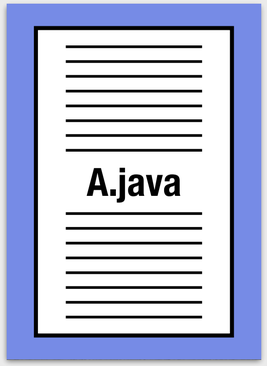
RQ1: How rigorously is test code reviewed? — Results

Test alone



Together

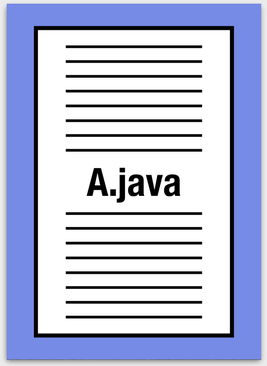
Production alone



RQ1: How rigorously is test code reviewed? — Results

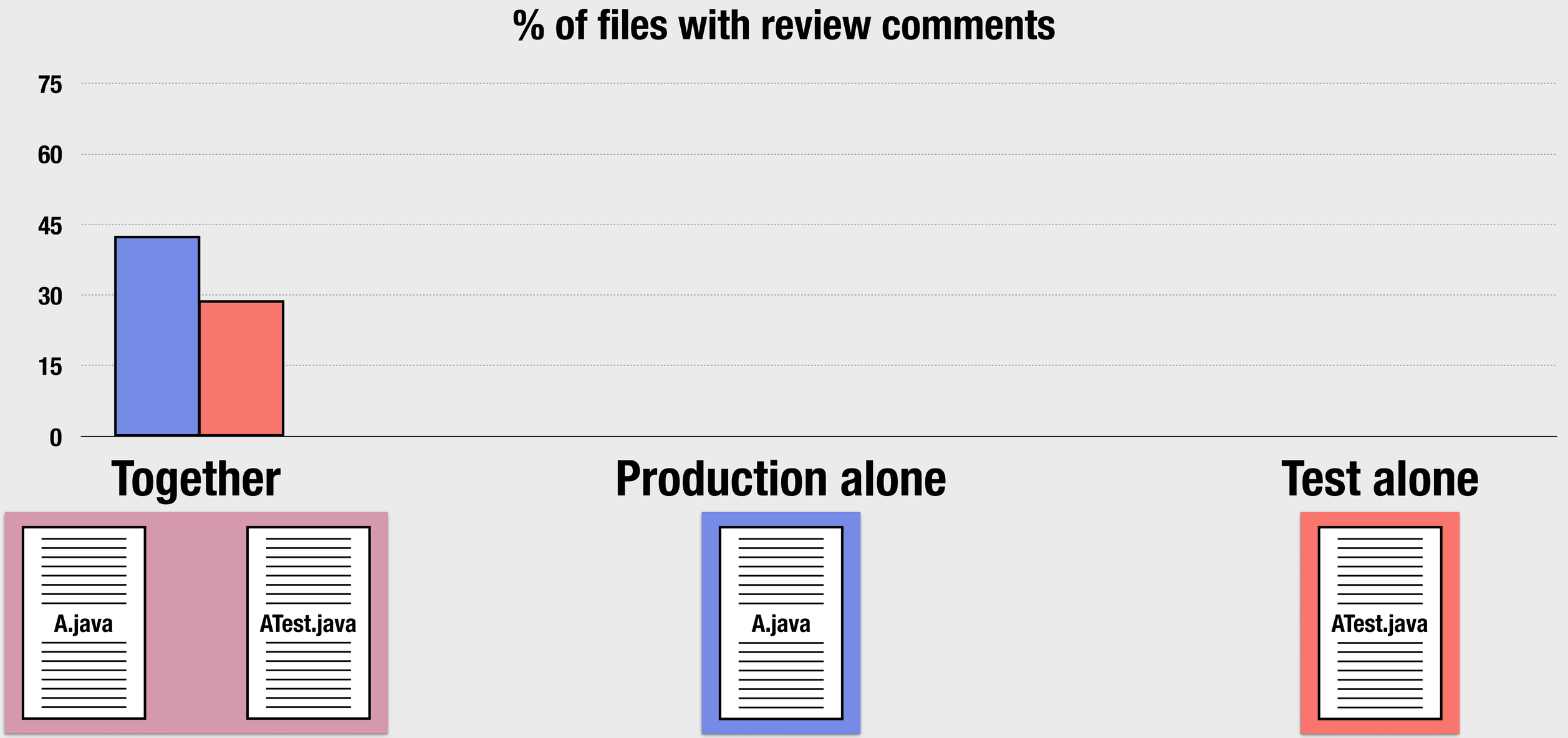
Together

Production alone

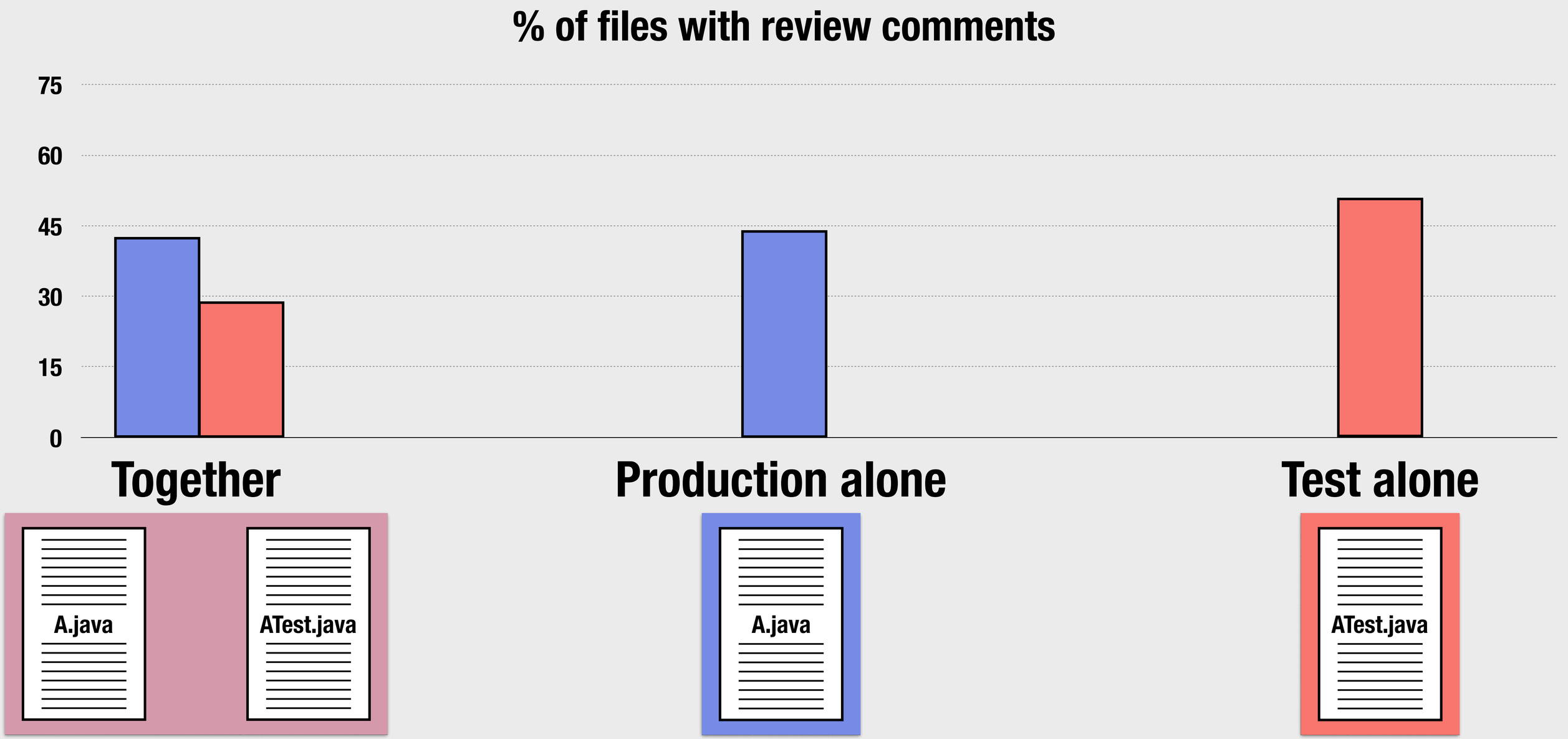


Test alone

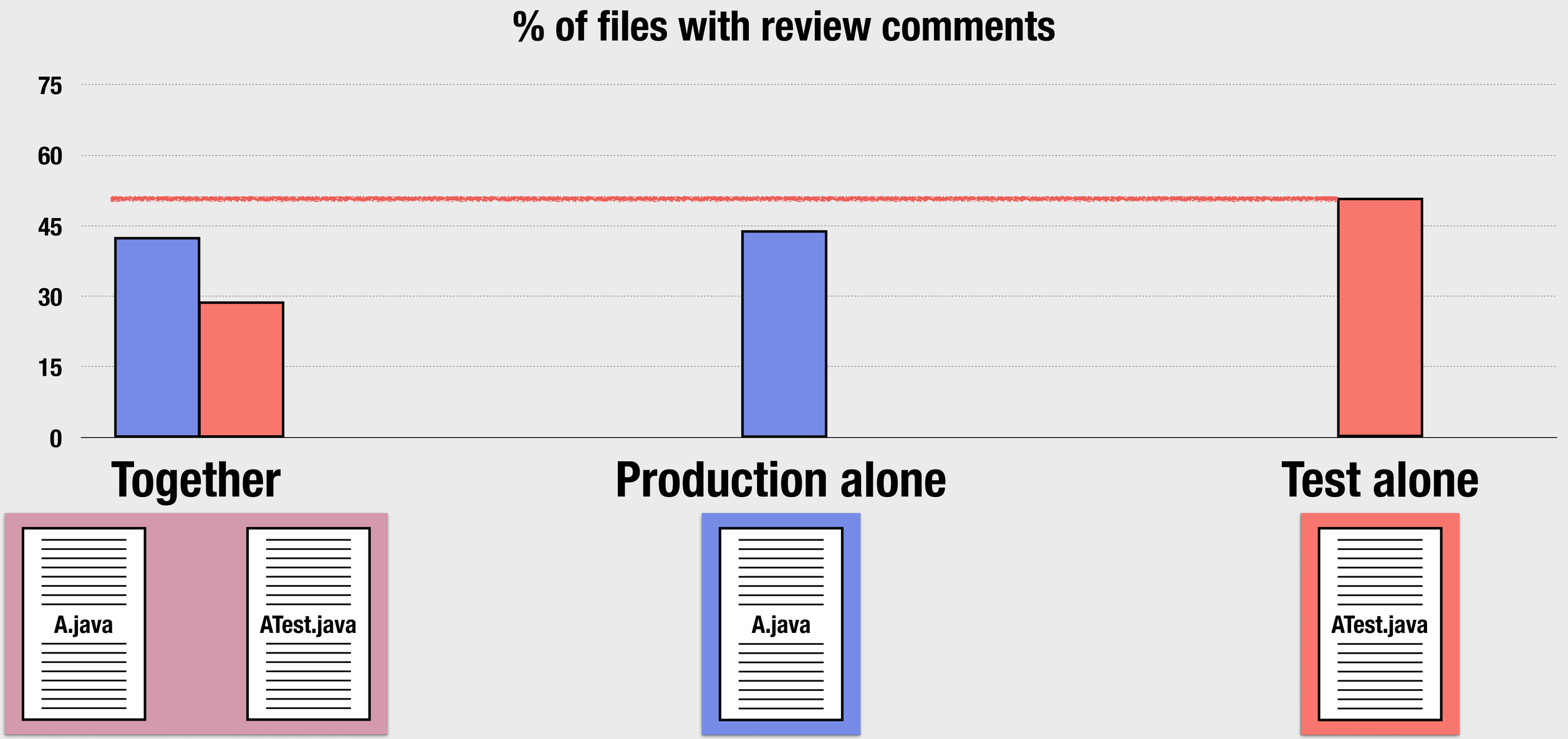
RQ1: How rigorously is test code reviewed? — Results



RQ1: How rigorously is test code reviewed? — Results

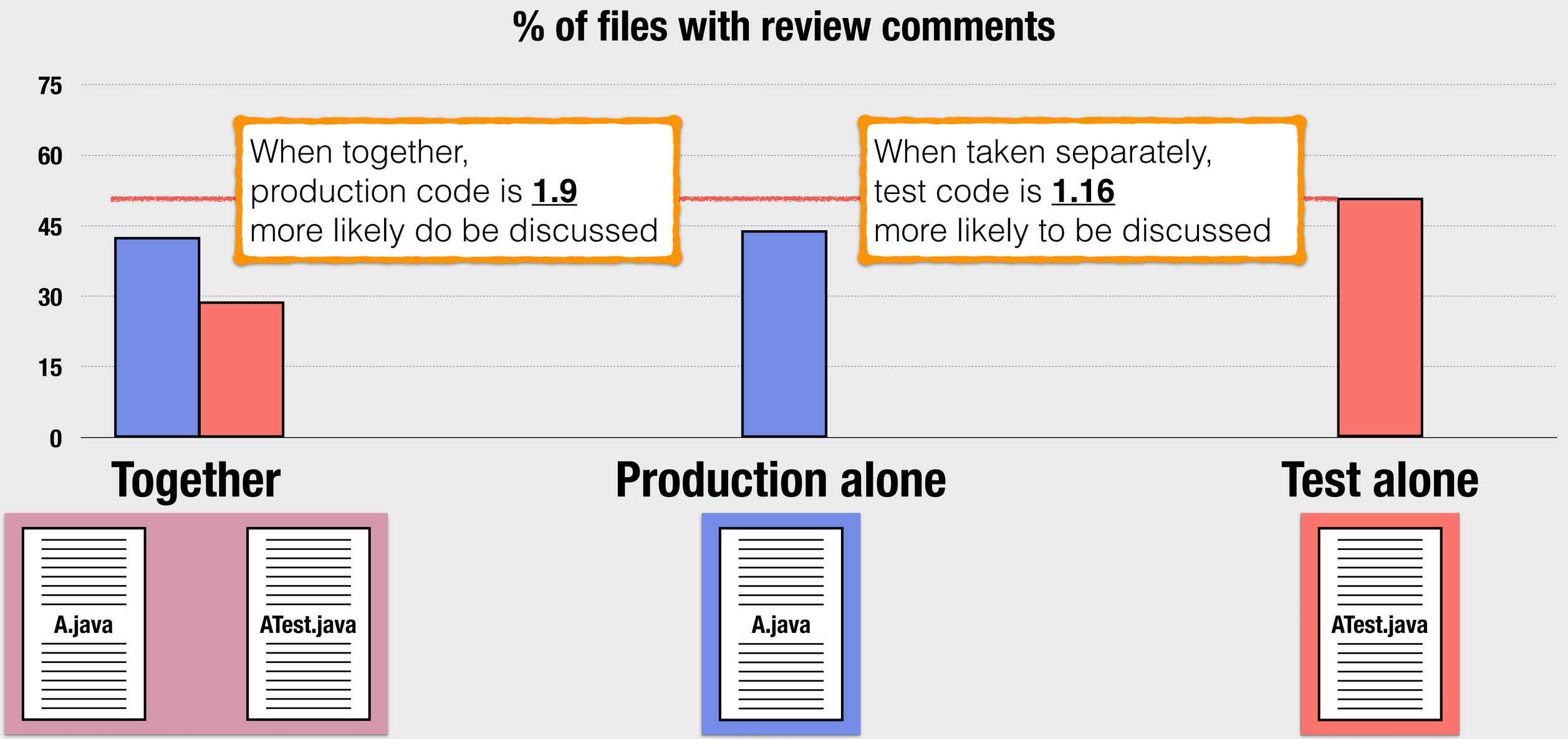


RQ1: How rigorously is test code reviewed? — Results



RQ1: How rigorously is test code reviewed? — Results

RQ1: How rigorously is test code reviewed? — Results



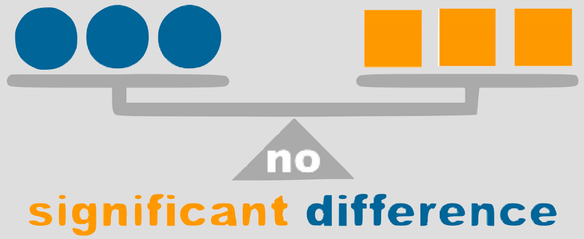
RQ1: How rigorously is test code reviewed? — Results

		Avg # of comments per file	Avg # of reviewers	Avg length of comments
Together	Production	3.00	5.49	19.09
	Test	1.27		15.32
Production alone		1.64	3.95	18.13
Test alone		2.30	5.15	17.01

RQ1: How rigorously is test code reviewed? — Summary

Prod. files are **2 times** more likely to be discussed than test files

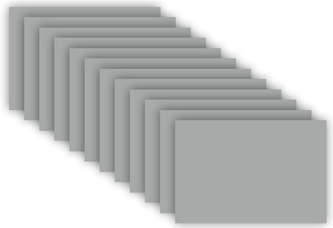
Test files are discussed more when alone



# of comments,
avg length,
avg # reviewers

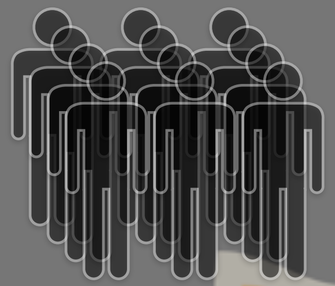
Research Questions

We collected Code Reviews from Gerrit



3 OSS: Eclipse, Qt, OpenStack

12 interviews



RQ1: **How** rigorously is test code reviewed?

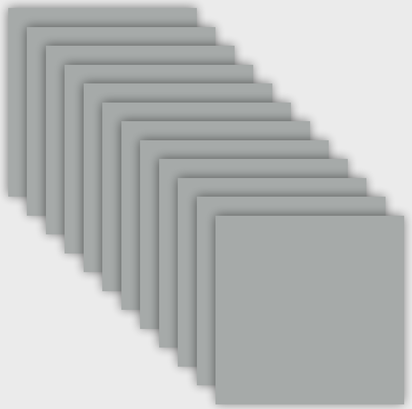
RQ2: **What** do reviewers discuss in test code reviews?

RQ3: Which **practices** do reviewers follow for test files?

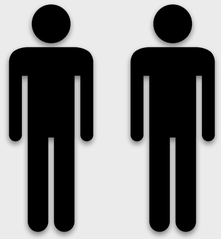
RQ4: What **problems** and **challenges** do developers face when reviewing tests?

RQ2: What do reviewers discuss in test code reviews? — Method

> 1,000,000 comments



600 comments

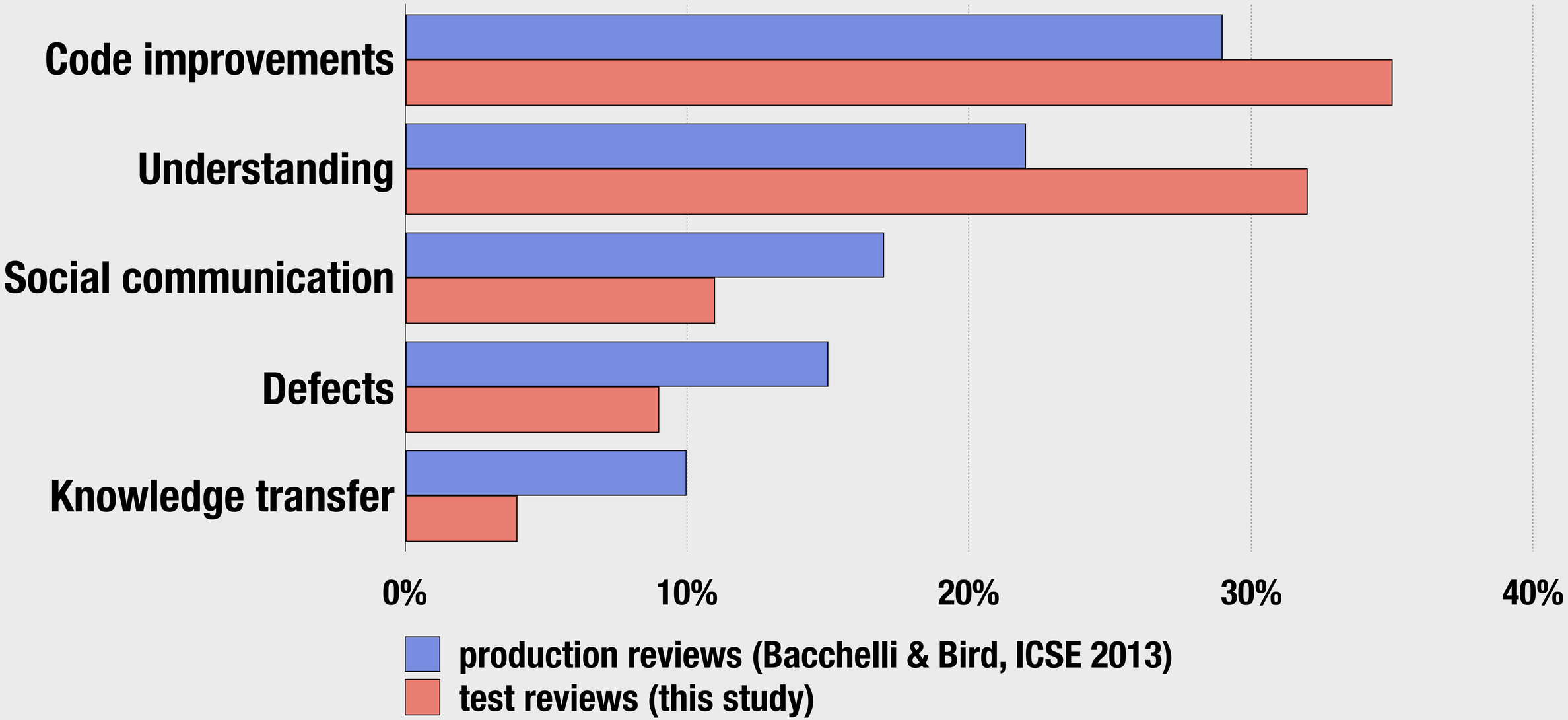


1st round: 6 categories*

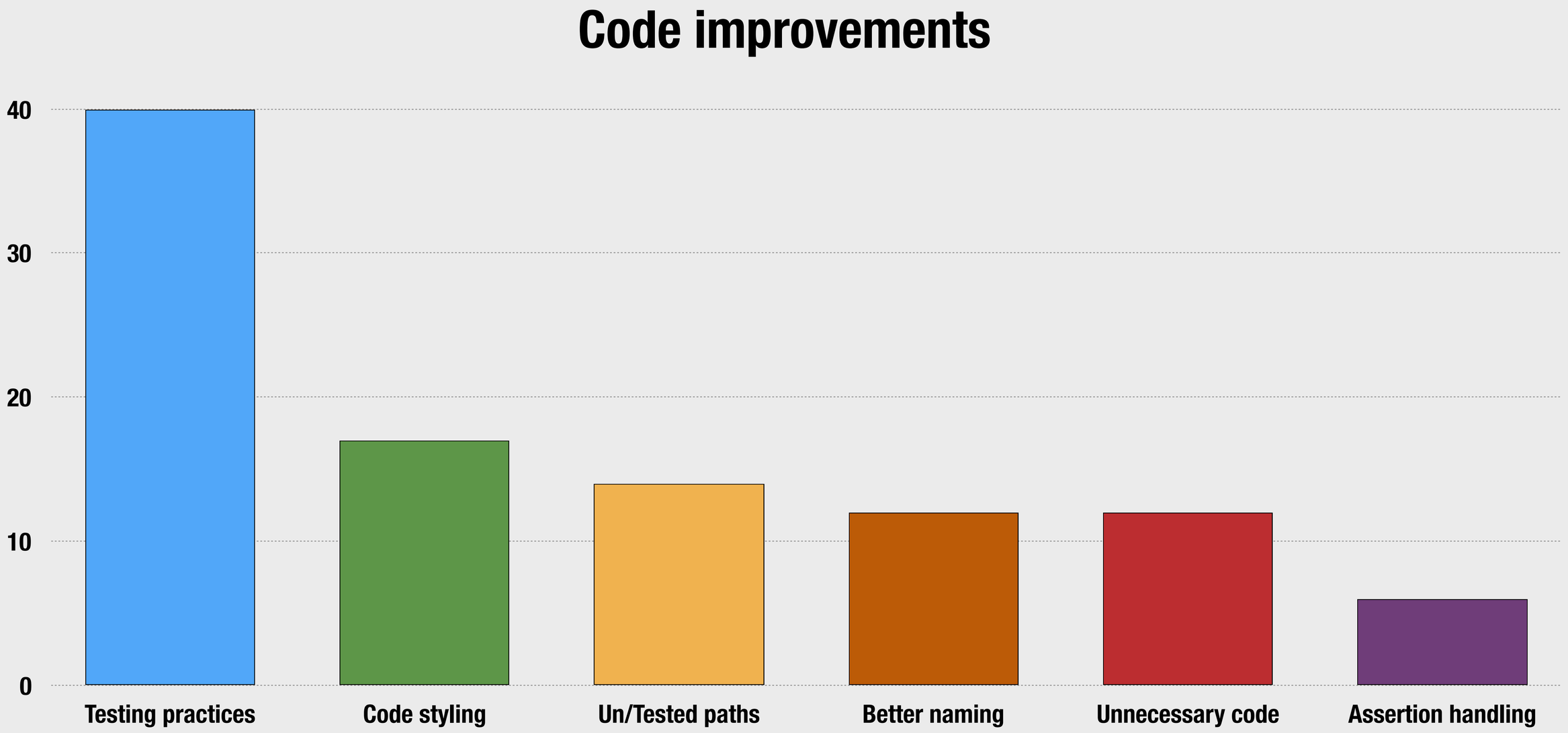
2nd round: for each category, more fine-grained

*Bacchelli & Bird. *Expectations, Outcomes, and Challenges of Modern Code Review*. ICSE 2013

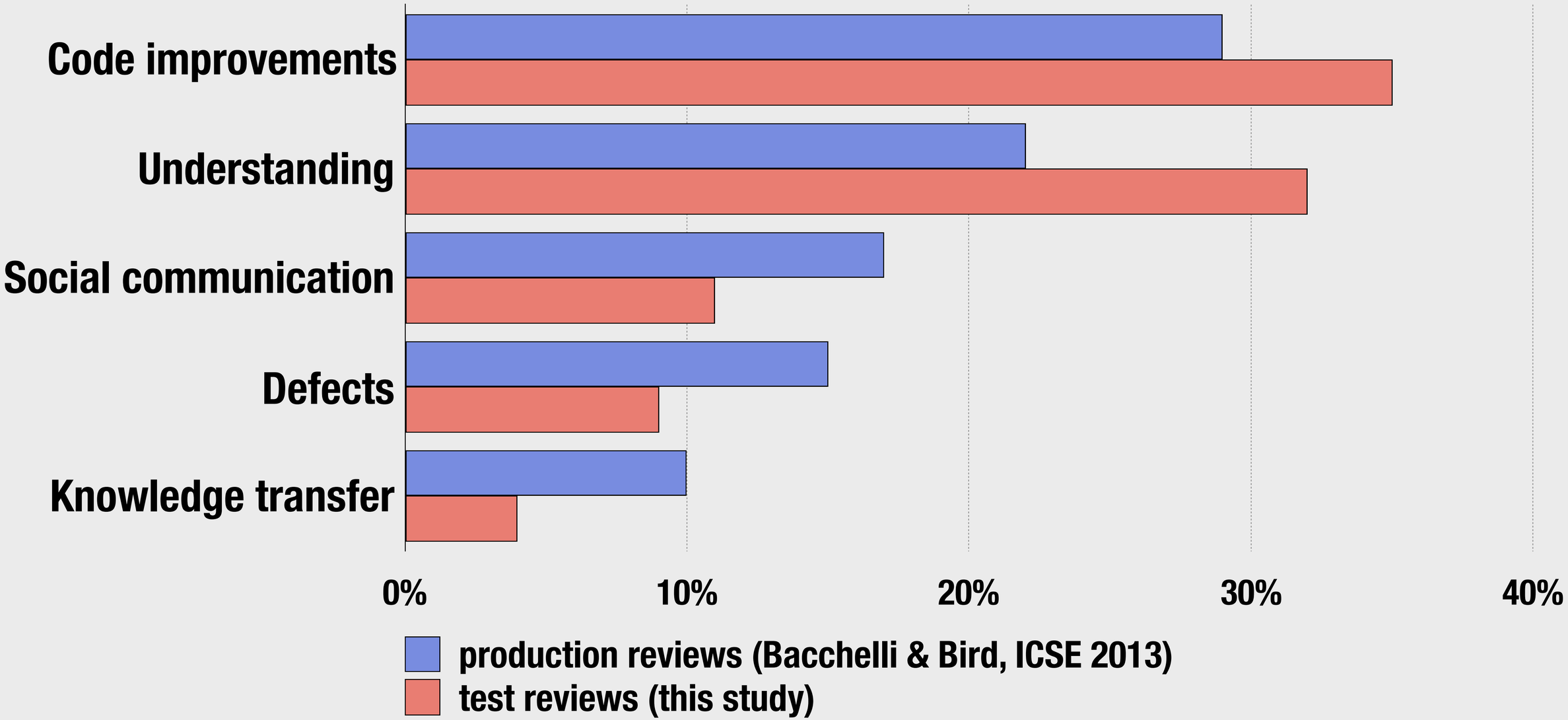
RQ2: What do reviewers discuss in test code reviews? — Results



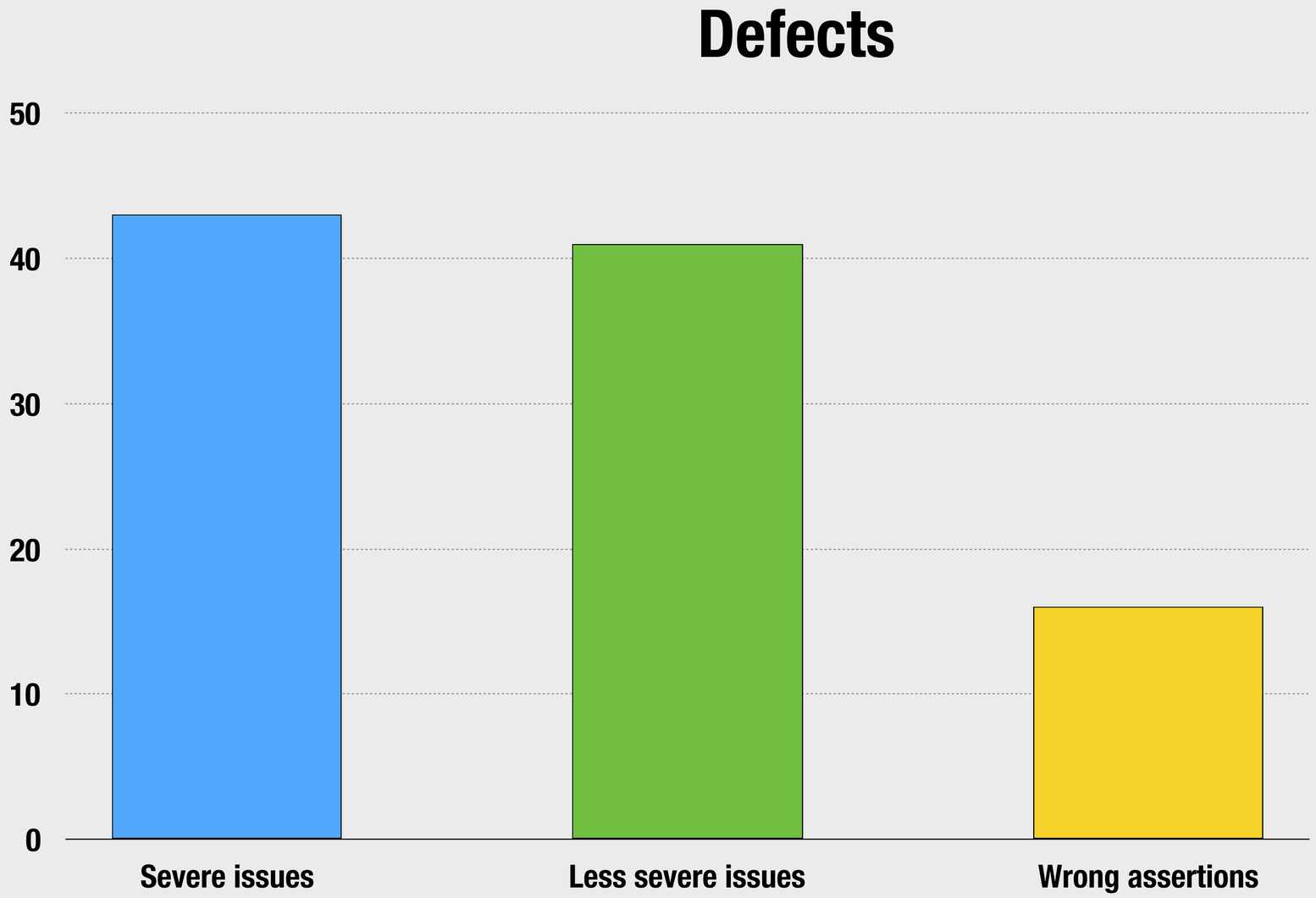
RQ2: What do reviewers discuss in test code reviews? — Results



RQ2: What do reviewers discuss in test code reviews? — Results



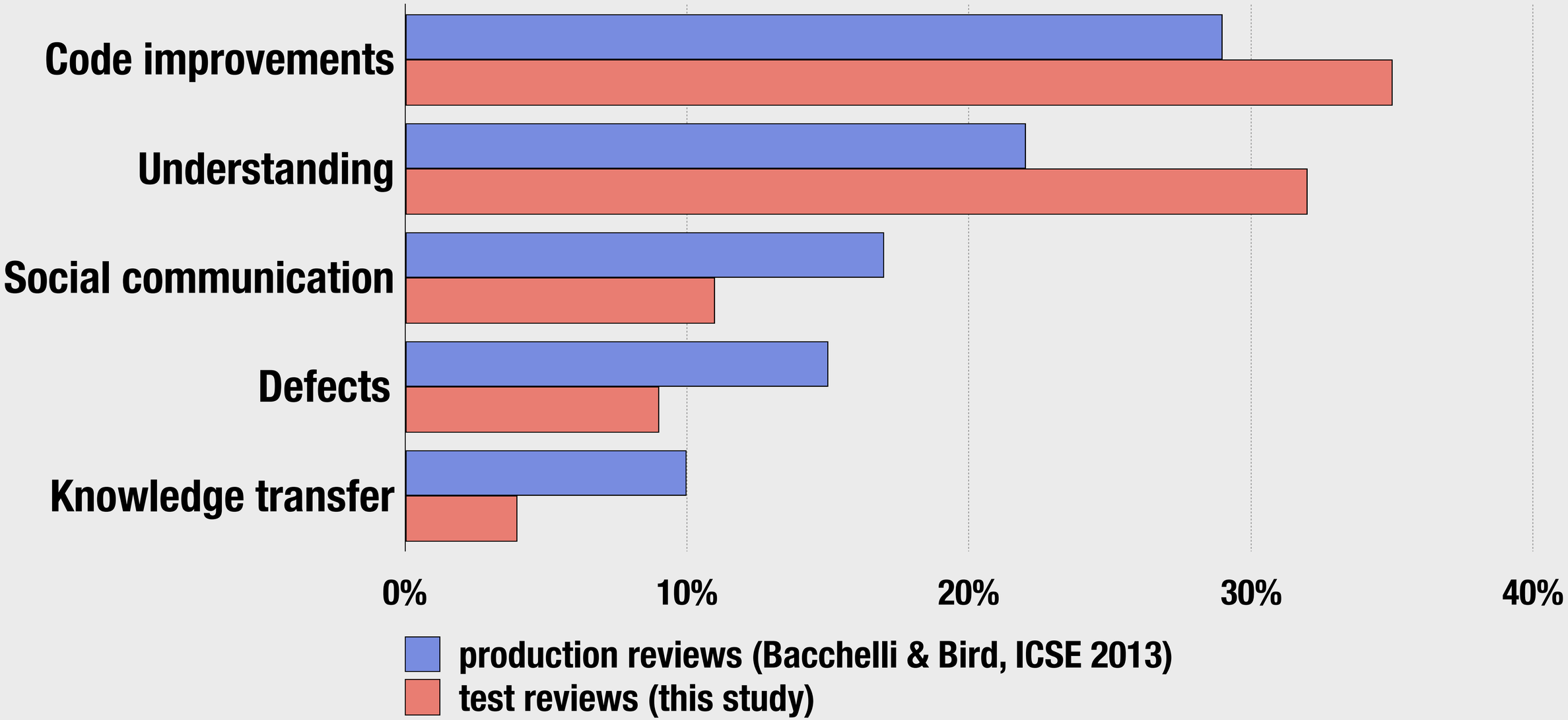
RQ2: What do reviewers discuss in test code reviews? — Results



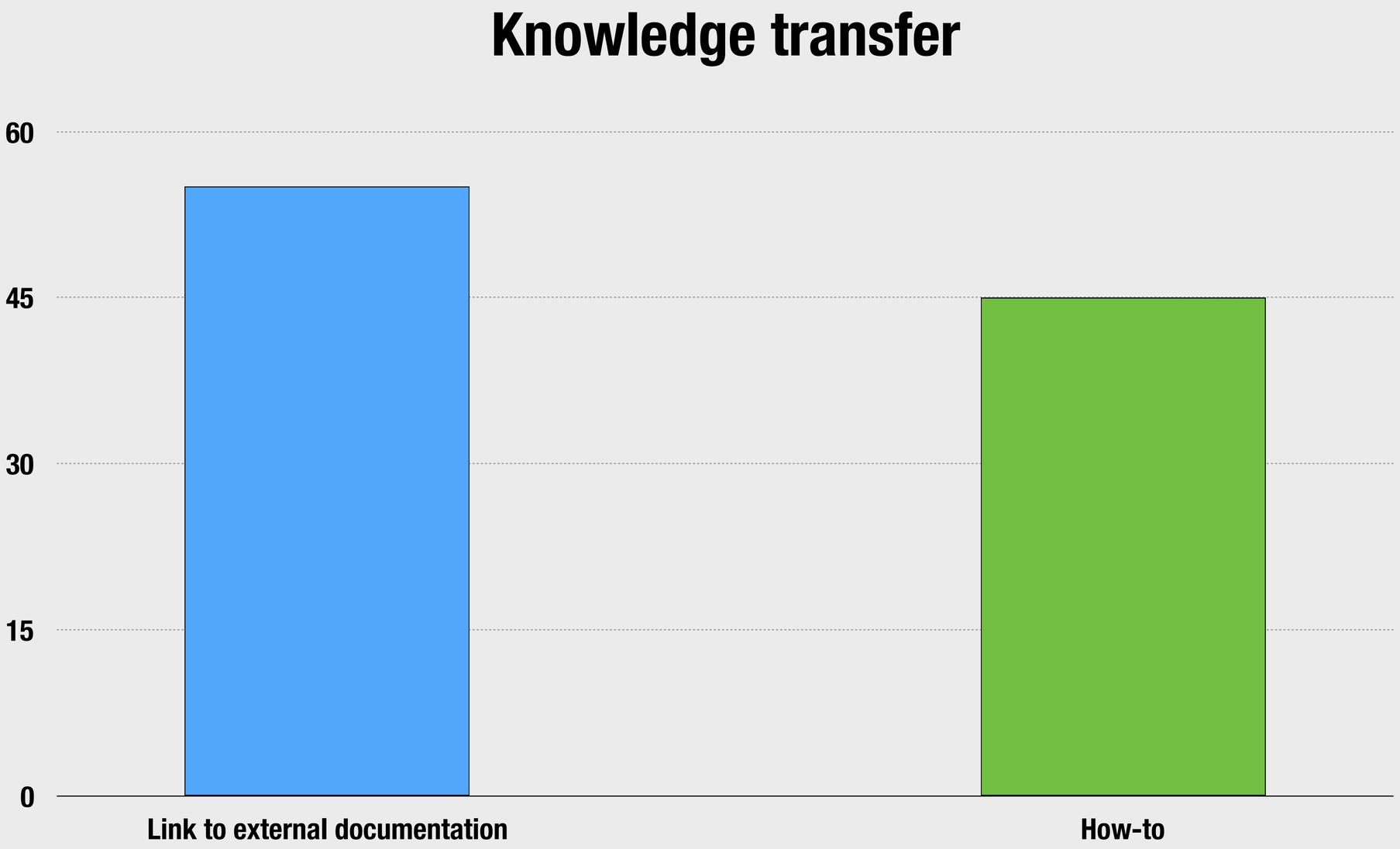
“You need to instantiate LttngKernelTrace here otherwise you will get a class cast exception” —
Severe issue

“This isnt being used, hence pep8 error.” —
Less severe issue

RQ2: What do reviewers discuss in test code reviews? — Results



RQ2: What do reviewers discuss in test code reviews? — Results



RQ2: What do reviewers discuss in test code reviews? — Summary

Testing practices, tested paths and assertions

Defects are **not** among the most discussed topics

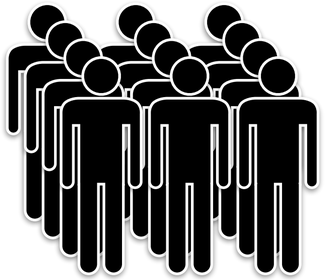
Developers often comment with how-to or ext. documentation

Research Questions

We collected Code Reviews from Gerrit

3 OSS: Eclipse, Qt, OpenStack

12 interviews



RQ1: **How** rigorously is test code reviewed?

RQ2: **What** do reviewers discuss in test code reviews?

RQ3: Which **practices** do reviewers follow for test files?

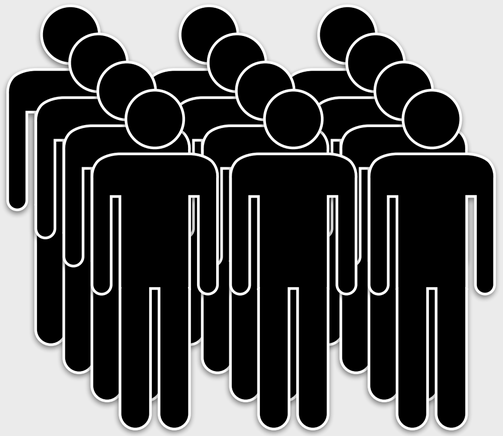
RQ4: What **problems** and **challenges** do developers face when reviewing tests?

RQ3: Which practices do reviewers follow for test files? — Method

Interviews!

RQ3: Which practices do reviewers follow for test files? — Method

12 interviews



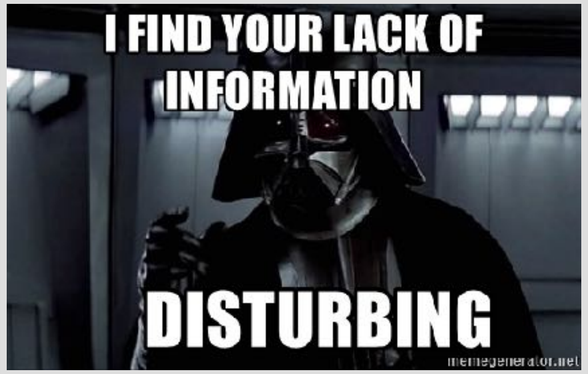
1 Openstack
1 Qt
1 Eclipse

---

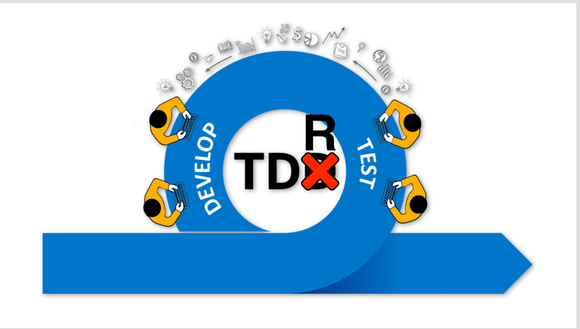
1 Microsoft
1 Ericsson
2 Alura
5 OSS

RQ3: Which practices do reviewers follow for test files? — Summary

Understanding if all paths are covered



Checking out the change and run the tests



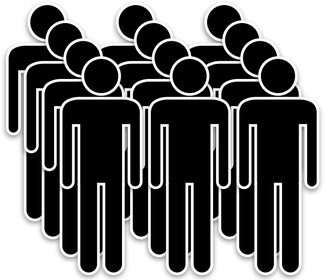
Test Driven Review

Research questions

We collected Code Reviews from Gerrit

3 OSS: Eclipse, Qt, OpenStack

12 interviews



RQ1: **How** rigorously is test code reviewed?

RQ2: **What** do reviewers discuss in test code reviews?

RQ3: Which **practices** do reviewers follow for test files?

RQ4: What **problems** and **challenges** do developers face when reviewing tests?

RQ4: What problems and challenges do developers face when reviewing tests?

Reviewing tests requires context on both tests and production

Management policies prioritize strongly production code

Novice developers do not know the effect of poor testing

Lack of navigation support and in-depth code coverage info

RQ4: What problems and challenges do developers face when reviewing tests?

Reviewing tests requires context on both tests and production

Management policies prioritize strongly production code

Novice developers do not know the effect of poor testing

Lack of navigation support and in-depth code coverage info

Implications and recommendations

Provide the context for reviewing test

Plan enough time for test review

Educate on test reviewing and writing

Detailed code coverage information

RQ1: How rigorously is test code reviewed? — Summary

Prod. files are **2 times** more likely to be discussed

Test files are discussed more when alone

# of comments, avg length, avg # reviewers

RQ2: What do reviewers discuss in test code reviews? — Summary

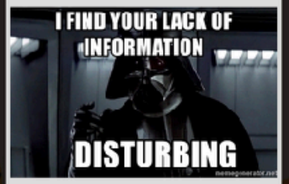
Testing practices, tested paths and assertions

Defects are **not** among the most discussed topics

Developers often comments with how-to or ext. documentation

RQ3: Which practices do reviewers follow for test files? — Summary

Understand if all paths are covered



Check out the change and run the tests

Test Driven Review

RQ4: What problems and challenges do developers face when reviewing tests?

Reviewing tests requires context on both tests and production

Management policies prioritize strongly production code

Novice developers do not know the effect of poor testing

Lack of navigation support and in-depth code coverage info

Implications and recommendations

Provide the context for reviewing test

Plan enough time for test review

Educate on test reviewing and writing

Detailed code coverage information

Code Review

- Developers have limited time to do code review

- Should test files be reviewed?
  - If production files are more prone to contain bugs…reviewer should focus on production!

Should test files be reviewed?

- We conducted a preliminary investigation to see whether test and production code files are equally associated with defects
- We built a statistical model and used as an explanatory variable “being a test”

Should test files be reviewed?

- 3 OSS projects: Qt, Eclipse and Openstack
- Dependent variable: post-release defects
- Explanatory variables: metrics well related to defect proneness: product metrics, process metrics and human factors.
  - Plus, our variable: isTest

Metrics in order of importance

Attribute	Average merit	Average Rank
Churn	0.753	1
Author ownership	0.599	2.2 ± 0.4
Cumulative churn	0.588	2.8 ± 0.4
Total authors	0.521	4
Major authors	0.506	5
Size	0.411	6
Prior defects	0.293	7
Minor authors	0.149	8
Is test	0.085	9

Should test files be reviewed?

- **Yes!**
- **The decision to review a file should not be based on whether the file contains production or test code, as this has no association with defects.**