

Check It Off: Exploring the Impact of a **Checklist Intervention** on the Quality of **Student-authored Unit Tests**

Gina R. Bai
Kai Presler-Marshall
Thomas W. Price
Kathryn T. Stolee

North Carolina State University

JULY 2022

Gina R. Bai – rbai2@ncsu.edu



A Representative Case

Bowling Scorekeeper

The objective is to TEST an application that can calculate the score of a single bowling game.

- There is no graphical user interface.
- You work ONLY with JUnit test cases in this project.
- You have ONE HOUR to work on this project.
- You are free to consult/use any online resources, including documentations, tutorials, Q&A sites, and any Eclipse built-in tools or plug-ins.

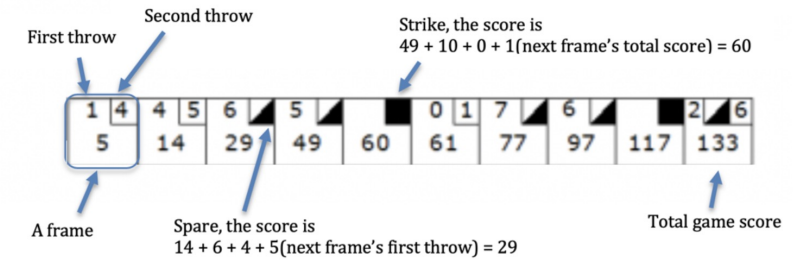
Project Template

You are provided with a completed project that contains three classes: `Frame`, `BowlingGame` and `BowlingException`, each contains some fields and methods. DO NOT CHANGE the names and functionalities of the existing fields and methods.

You are expected to create JUnit test cases to verify the behavior of this implementation as thorough as possible based on the following description of a bowling score keeper. Your program should throw `BowlingException` in all error situations.

Bowling Score Keeper Task Description

The game consists of 10 frames as shown below. In each frame the player has two opportunities to knock down 10 pins. The score for the frame is the total number of pins knocked down, plus bonuses for strikes and spares.



```
1 package tdd.bsk;
2
3 import tdd.bsk.BowlingException;
4
5 public class Frame {
6     private int firstThrow;
7     private int secondThrow;
8
9     public Frame(int firstThrow, int secondThrow) throws BowlingException {
10        if (firstThrow > 10 || firstThrow < 0
11            || secondThrow > 10 || secondThrow < 0
12            || firstThrow + secondThrow > 10 || firstThrow + secondThrow < 0
13        ) {
14            throw new BowlingException();
15        }
16        this.firstThrow = firstThrow;
17        this.secondThrow = secondThrow;
18    }
19
20    // the score of a single frame
21    public int score() {
22        return firstThrow + secondThrow;
23    }
24
25    // returns whether the frame is a strike or not
26    public boolean isStrike() {
27        return firstThrow == 10 && secondThrow == 0;
28    }
29
30    // return whether a frame is a spare or not
31    public boolean isSpare() {
32        return !isStrike() && firstThrow + secondThrow == 10;
33    }
34}
```

What code to test?
And How?

Scenarios? Syntax?...



Kai

A Representative Case



Kai



A Representative Case



Kai

Is the program tested enough?

When can I stop testing?

A Representative Case



Kai



A Representative Case

How to interpret the report?



"I could not see which branches it thought I was missing. It was neither obvious, nor easy, to automate unit test execution."

```
5 public class Frame {
6     private int firstThrow;
7     private int secondThrow;
8
9     public Frame(int firstThrow, int secondThrow) throws BowlingExce
10        if (firstThrow > 10 || firstThrow < 0
11        2 of 4 branches missed. secondThrow > 10 || secondThrow < 0
12        || firstThrow + secondThrow > 10 || firstThrow + sec
13        ){
14        throw new BowlingException();
15    }
16    this.firstThrow = firstThrow;
17    this.secondThrow = secondThrow;
18 }
19
20 // the score of a single frame
21 public int score() {
22     return firstThrow + secondThrow;
23 }
24
```

Element	Coverage	Covered Instructions	Missed Instructions
BowlingScoreKeeper	20.3 %	58	228
...	20.3 %	58	228
...	13.0 %	34	228
...ame.java	0.0 %	0	182
...gGame	0.0 %	0	182
...a	44.2 %	34	43
...	44.2 %	34	43
...ception.java	0.0 %	0	3
...	100.0 %	24	0
...	100.0 %	24	0

Students Need Support in...

- Identifying **what** code to test and **how** to test it
- Creating tests that are semantically and syntactically correct
- Evaluating test code quality
 - completeness & effectiveness (e.g., “when to stop testing”)

Testing Checklists

- ✓ Static
- ✓ Lightweight
- ✓ Transferable

Test Case Checklist

Each test case should:

- be executable (i.e., it has an `@Test` annotation and can be run via "Run as JUnit Test")
- have at least one assert statement or assert an exception is thrown. Example assert statements include: `assertTrue`, `assertFalse`, and `assertEquals` ([click for tutorials](#)). For asserting an exception is thrown, there are different approaches: `try{...; fail();} catch(Exception e){assertThat...;}`, `@Test(expected = exception.class)` in JUnit 4, or `assertThrows` in JUnit 5 ([click for tutorials](#)).
- evaluate/test only one method

Each test case could:

- be descriptively named and commented
- If there is redundant setup code in multiple test cases, extract it into a common method (e.g., using `@Before`)
- If there are too many assert statements in a single test case (e.g., more than 5), you might split it up so each test evaluates one behavior.

Test Suite Checklist

The test suite should:

- have at least one test for each requirement
- appropriately use the setup and teardown code (e.g., `@Before`, which runs before each `@Test`)
- contain a fault-revealing test for each bug in the code (i.e., a test that fails)
- For each requirement, contain test cases for:
 - Valid inputs
 - Boundary cases
 - Invalid inputs
 - Expected exceptions

To improve the test suite, you could:

- measure code coverage using an appropriate tool, such as EclEmma ([installation](#), [tutorial](#)). Inspect uncovered code and write tests as appropriate.

Testing Checklists

Contains

- ✓ Tutorial information
- ✓ Testing strategies

Test Case Checklist

Each test case *should*:

- be executable (i.e., it has an `@Test` annotation and can be run via "Run as JUnit Test")
- have at least one assert statement or assert an exception is thrown. Example assert statements include: `assertTrue`, `assertFalse`, and `assertEquals` ([click for tutorials](#)). For asserting an exception is thrown, there are different approaches: `try{...; fail();} catch(Exception e){assertThat...;}`, `@Test(expected = exception.class)` in JUnit 4, or `assertThrows` in JUnit 5 ([click for tutorials](#)).
- evaluate/test only one method

[Tutorial Info & Syntax](#)

Each test case *could*:

- be descriptively named and commented
- If there is redundant setup code in multiple test cases, extract it into a common method (e.g., using `@Before`)
- If there are too many assert statements in a single test case (e.g., more than 5), you might split it up so each test evaluates one behavior.

Test Suite Checklist

The test suite *should*:

[Test Class Components](#)

- have at least one test for each requirement
- appropriately use the setup and teardown code (e.g., `@Before`, which runs before each `@Test`)
- contain a fault-revealing test for each bug in the code (i.e., a test that fails)
- For each requirement, contain test cases for:
 - Valid inputs
 - Boundary cases
 - Invalid inputs
 - Expected exceptions

[Equivalence Class Partitioning](#)
[Boundary Value Analysis](#)

To improve the test suite, you *could*:

- measure code coverage using an appropriate tool, such as EclEmma ([installation](#), [tutorial](#)). Inspect uncovered code and write tests as appropriate.

Testing Checklists

Addresses

✓ Common mistakes

✓ Common test smells

[Bai et al. ITiCSE '21]

[Bijlsma et al. ICSE-SEET '21]

[Aniche et al. SIGCSE '19]

[Edwards et al. ICSE Companion '14]

Test Case Checklist

Each test case *should*:

- be executable (i.e., it has an `@Test` annotation and can be run via "Run as JUnit Test")
- have at least one assert statement or assert an exception is thrown. Example assert statements include: `assertTrue`, `assertFalse`, and `assertEquals` ([click for tutorials](#)). For asserting an exception is thrown, there are different approaches: `try{...; fail();} catch(Exception e){assertThat...;}`, `@Test(expected = exception.class)` in JUnit 4, or `assertThrows` in JUnit 5 ([click for tutorials](#)).
- evaluate/test only one method

Syntax Errors

No Assertions

Each test case *could*:

- be descriptively named and commented
- If there is redundant setup code in multiple test cases, extract it into a common method (e.g., using `@Before`)
- If there are too many assert statements in a single test case (e.g., more than 5), you might split it up so each test evaluates one behavior.

Bad Naming

Assertion Roulette

Test Suite Checklist

The test suite *should*:

- have at least one test for each requirement
- appropriately use the setup and teardown code (e.g., `@Before`, which runs before each `@Test`)
- contain a fault-revealing test for each bug in the code (i.e., a test that fails)
- For each requirement, contain test cases for:
 - Valid inputs
 - Boundary cases
 - Invalid inputs
 - Expected exceptions

Poor Requirement Coverage

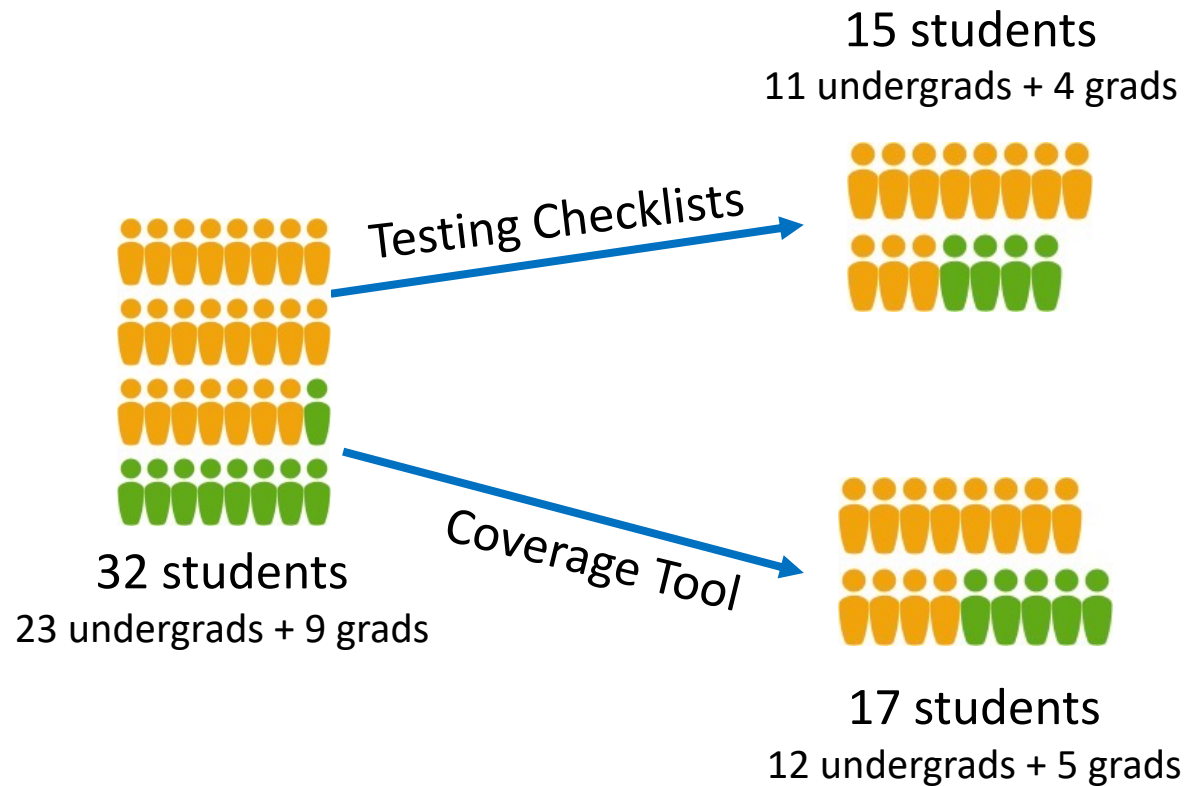
Misinterpretation of Failing Tests

Testing Happy Path Only

To improve the test suite, you *could*:

- measure code coverage using an appropriate tool, such as EclEmma ([installation](#), [tutorial](#)). Inspect uncovered code and write tests as appropriate.

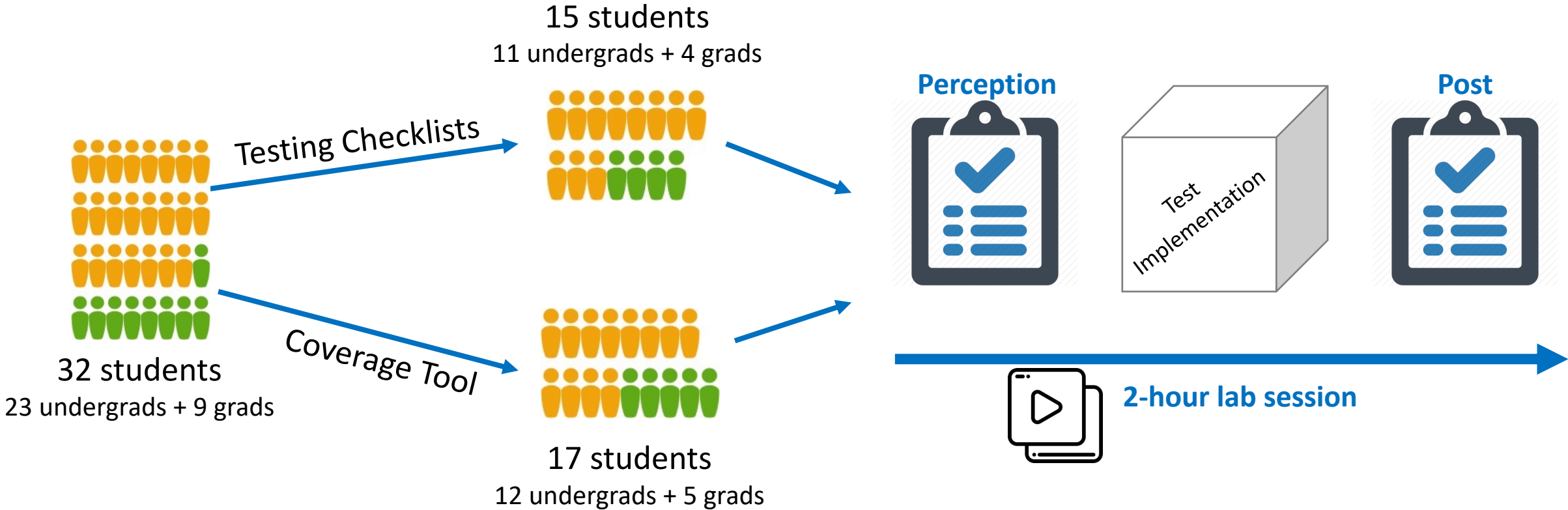
Methodology



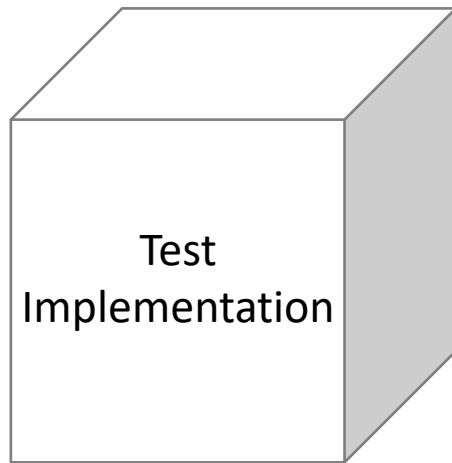
	avg_yrJava	avg_yrUT
Undergrads	3.5	3.0
Grads	0.1	0.4

	avg_yrJava	avg_yrUT
Undergrads	3.8	2.7
Grads	1.4	1.4

Methodology



Measurements of Test Code Quality



- **Completeness**

- Requirements coverage
- Instruction coverage
- Branch coverage

- **Effectiveness**

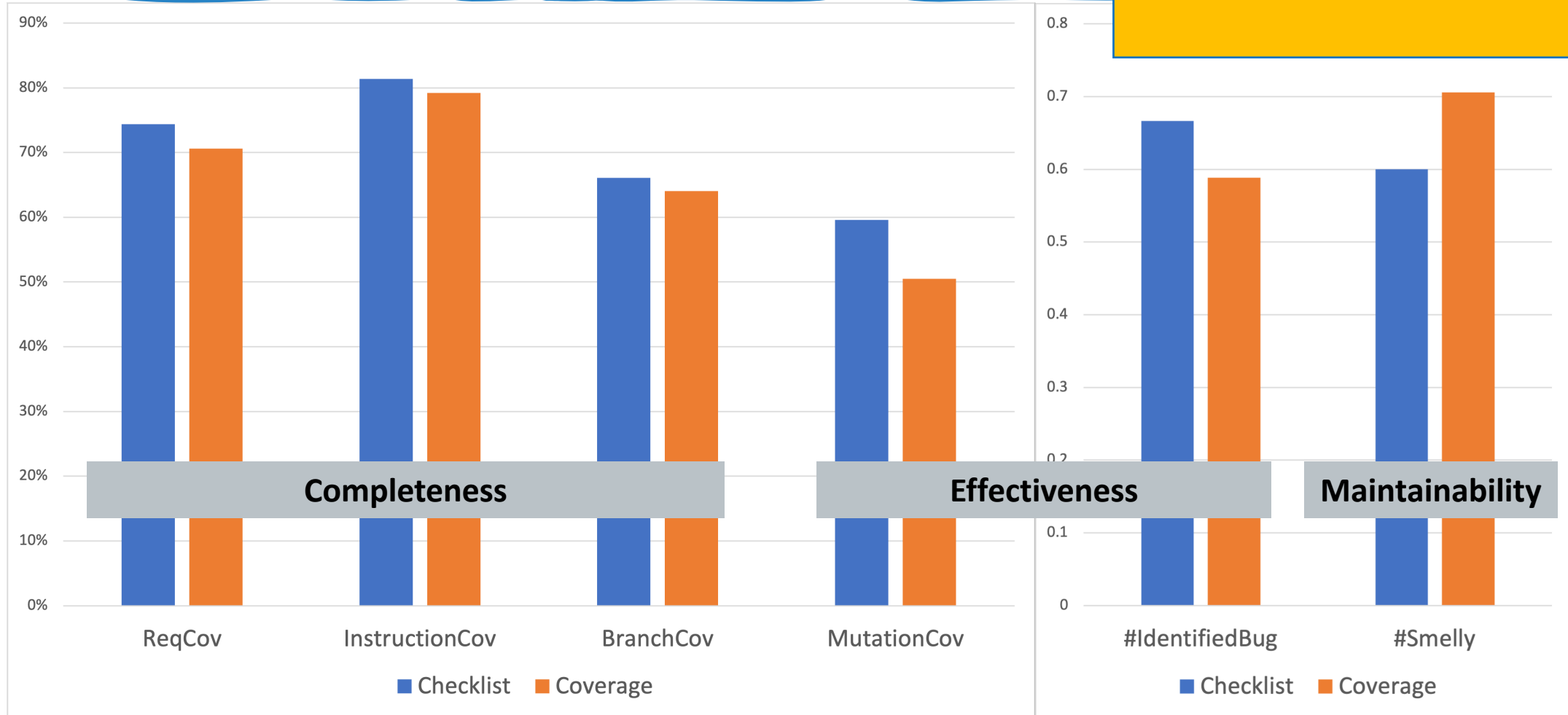
- Mutation coverage
- The number of identified seeded bugs

- **Maintainability**

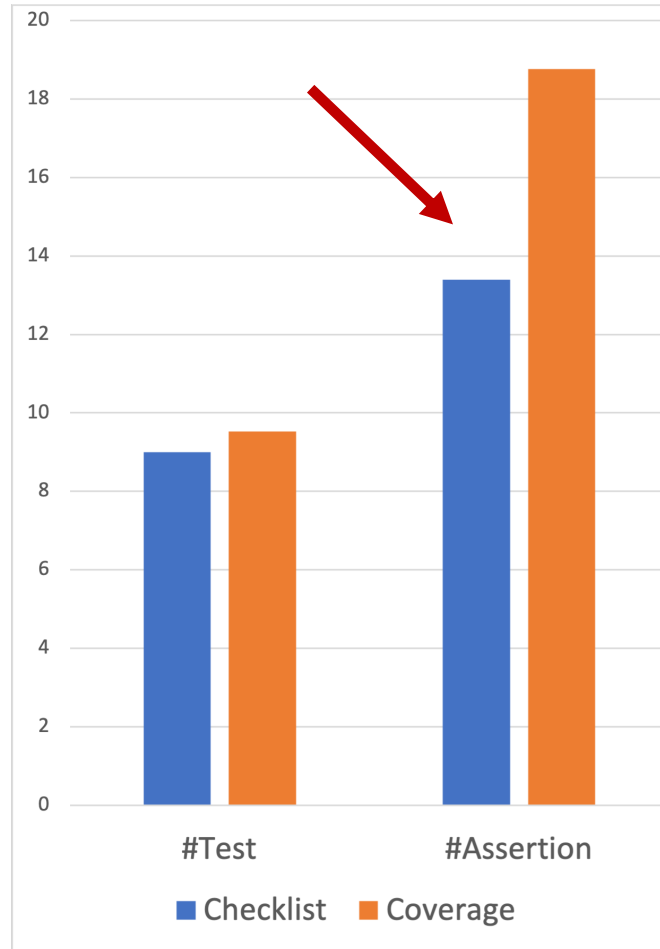
- The number of smelly tests

Checklists vs. Coverage Tools

Tool support does not need to be sophisticated to be effective



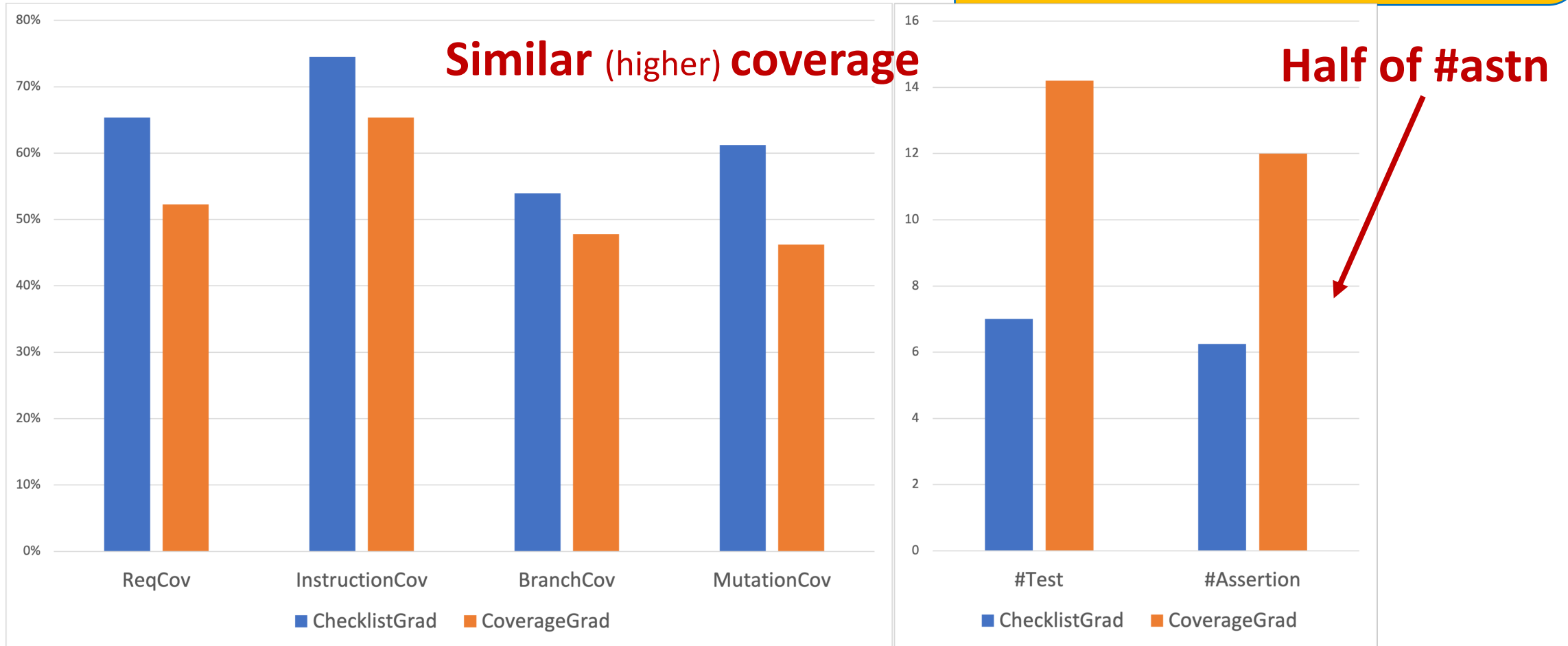
Checklists vs. Coverage Tools



Similar test code quality
with **less** assertions

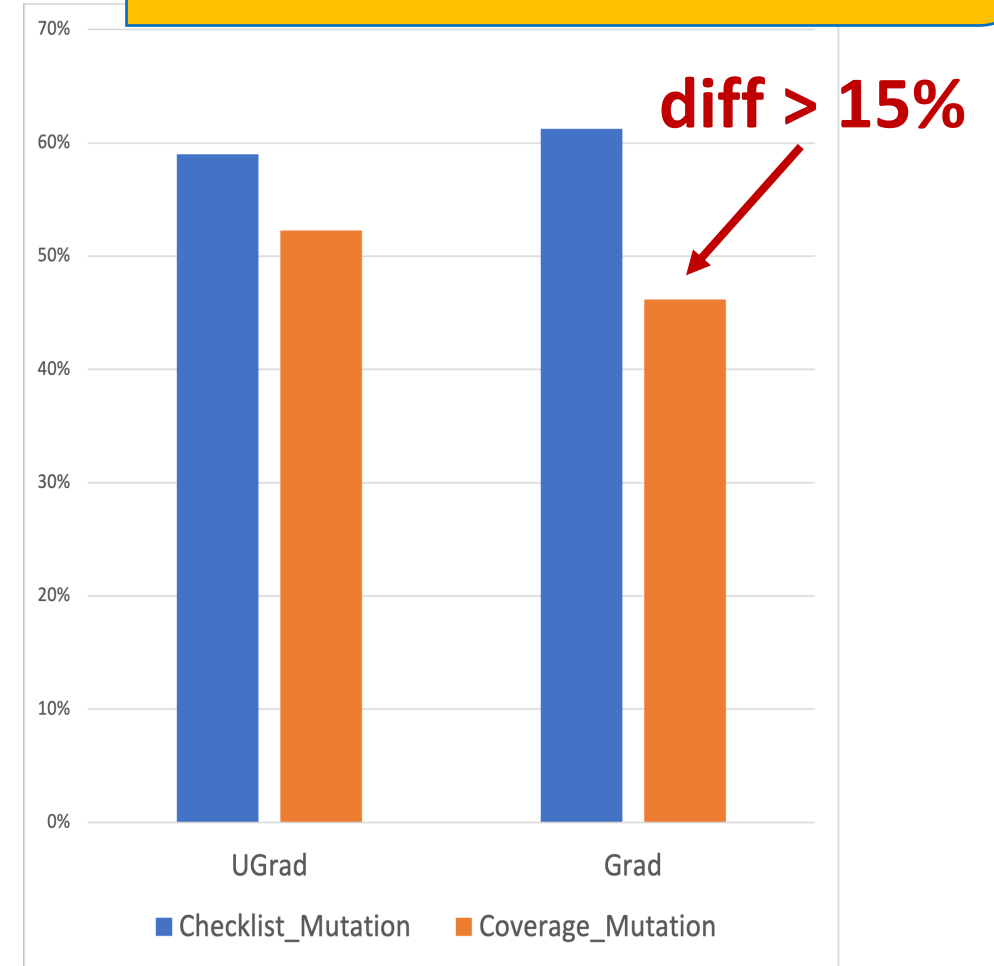
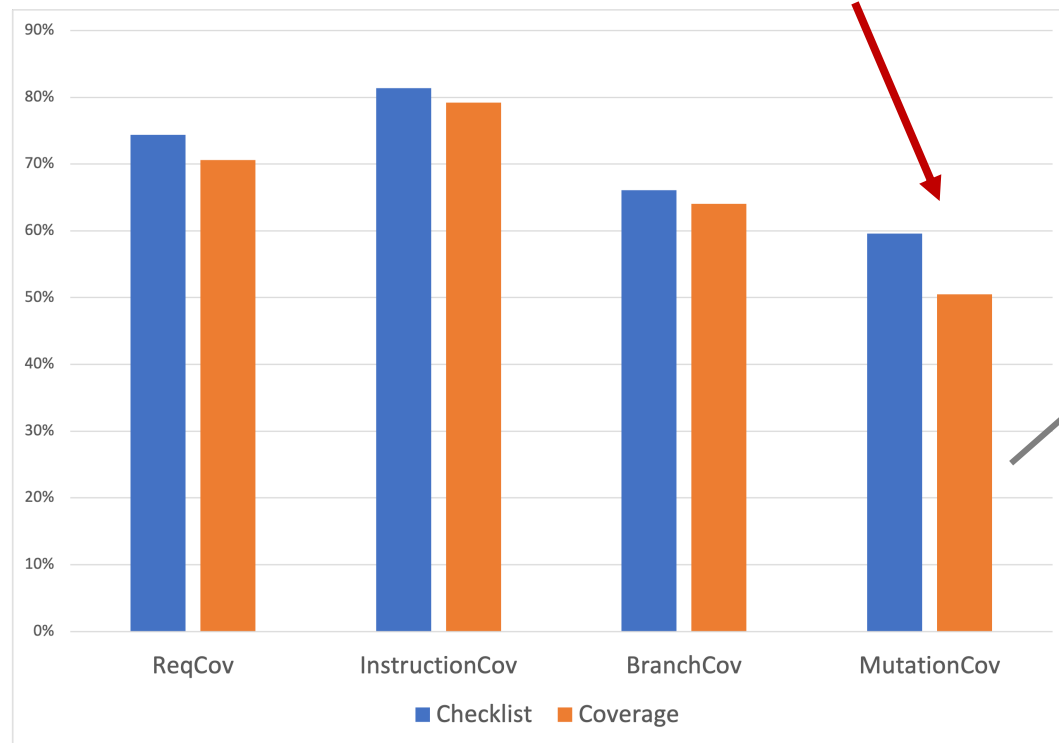
Checklists vs. Coverage T

Novices may benefit more from the checklists



Checklists vs. Coverage T

Using checklists may help students focus more on boundary values



Student Engagement

- ❖ Students found the checklists “very helpful” (Likert-scale: 3.9/5)
- ❖ Most (13/15) students self-reported that they read the checklists before they wrote any unit tests

Future Work

Replication Studies

- With diverse and larger set of students and study tasks

Extending the Checklist Intervention

- Supports automated real-time feedback
 - a progress report
 - coverage reports
 - hints on how to address any shortcoming of the tests
- Intelligent tutoring systems

Adoption of Think-aloud or Eye-tracking

- To learn students' decision-making process

Takeaways

- ❖ Tool support does not need to be sophisticated to be effective
- ❖ Students who have lower prior knowledge in Java and unit testing may benefit more from the checklist