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

*Images from google.com

Gina R. Bai – rbai2@ncsu.edu

Representative Questions

Amy

I was wondering what code should I test? What testing scenarios should I have?

I believe I found a bug in the source code, but I don't know how to show that in JUnit… I found some code examples on StackOverflow, but it's giving me a compile error… Can I just delete it, or just describe it in comments?

Bob

Charlie

One of my tests failed, is it okay? Should I fix the test to make it pass? Does the failure indicate a bug in the source code? Or in my testing code?

[Bai, Smith, Stolee. ITiCSE '21]

Bowling Scorekeeper

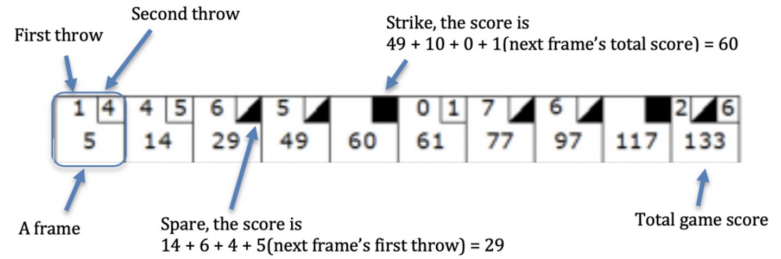
The objective is to TEST an application that can calculate the score of a single bowling game.

- 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.

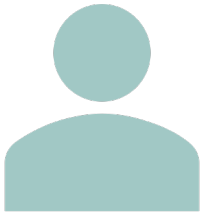
You are expected to create JUnit test cases to verify the behavior of this implementation as thorough as possible based on the following description of a bowling score keeper. Your program should throw `BowlingException` in all error situations.

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.




```java
package tdd.bsk;

import tdd.bsk.BowlingException;

public class Frame {
        private int firstThrow;
        private int secondThrow;

        public Frame(int firstThrow, int secondThrow) throws BowlingException {
                if (firstThrow > 10 || firstThrow < 0
                                || secondThrow > 10 || secondThrow < 0
                                || firstThrow + secondThrow > 10 || firstThrow + secondThrow < 0
                                ){
                throw new BowlingException();
            }
                this.firstThrow = firstThrow;
                this.secondThrow = secondThrow;
        }

        // the score of a single frame
        public int score() {
                return firstThrow + secondThrow;
        }

        // returns whether the frame is a strike or not
        public boolean isStrike() {
                return firstThrow == 10 && secondThrow == 0;
        }

        // return whether a frame is a spare or not
        public boolean isSpare() {
                return !isStrike() && firstThrow + secondThrow == 10;
        }

```

Representative Questions



Daniel

I was wondering how many tests do I need to write? EclEmma says there are uncovered branches, but I could not see which branches it thought I was missing. Do I need to test everything? When can I stop testing?

```
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
```

Problems | Javadoc | Declaration | Coverage

Element	Coverage	Covered Instructions	Missed Instructions
BowlingScoreKeeper	20.3 %	58	228
src	20.3 %	58	228
tdd.bsk	13.0 %	34	228
BowlingGame.java	0.0 %	0	182
BowlingGame	0.0 %	0	182
Frame.java	44.2 %	34	43
Frame	44.2 %	34	43
BowlingException.java	0.0 %	0	3
tdd.bsk.tests	100.0 %	24	0
tests.java	100.0 %	24	0

[Bai, Smith, Stolee. ITiCSE ‘21]

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”)

[Bai, Smith, Stolee. ITiCSE ‘21]

Testing Checklists

- ✓ Static
- ✓ Lightweight
- ✓ Transferable

README.md

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

README.md

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]

README.md

Test Case Checklist

Each test case *should*:

Syntax Errors

- [ ] 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).

No Assertions

- [ ] evaluate/test only one method

Each test case *could*:

Bad Naming

- [ ] 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.

Assertion Roulette

Test Suite Checklist

The test suite *should*:

Poor Requirement Coverage

- [ ] 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:

Misinterpretation of Failing Tests

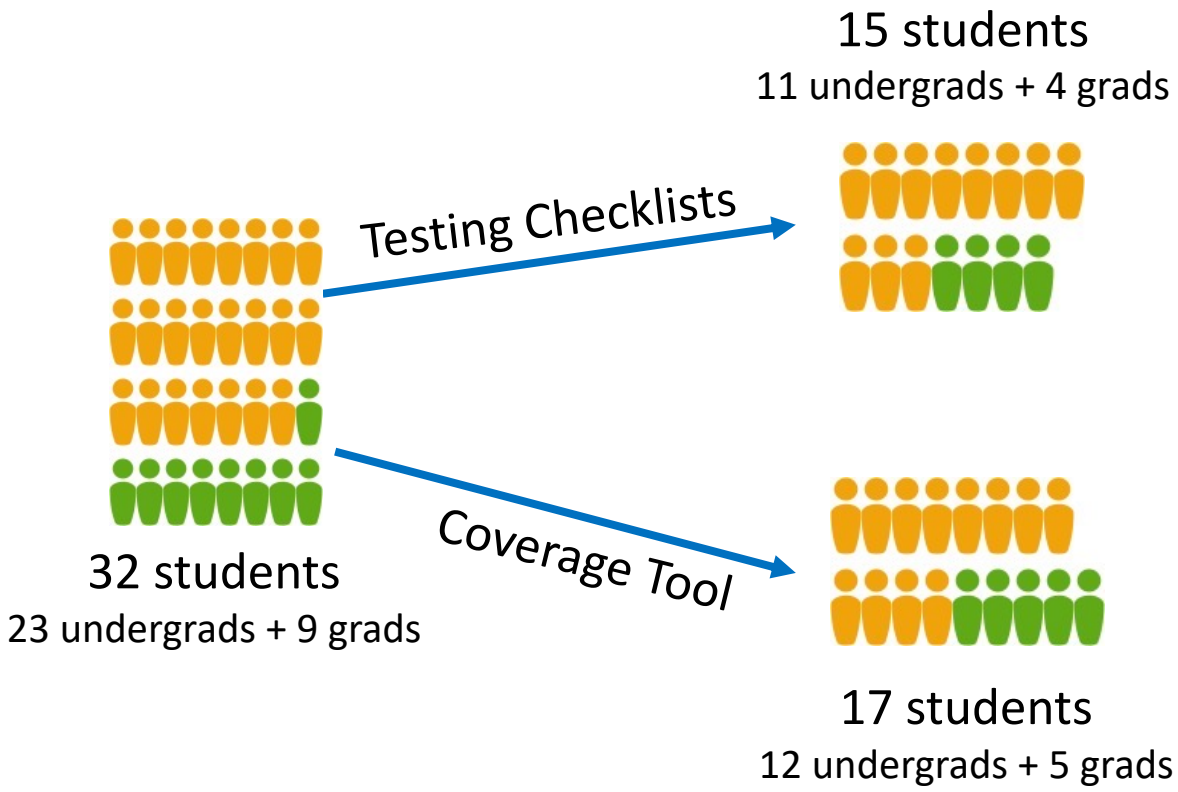
  - [ ] Valid inputs
  - [ ] Boundary cases
  - [ ] Invalid inputs
  - [ ] Expected exceptions

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

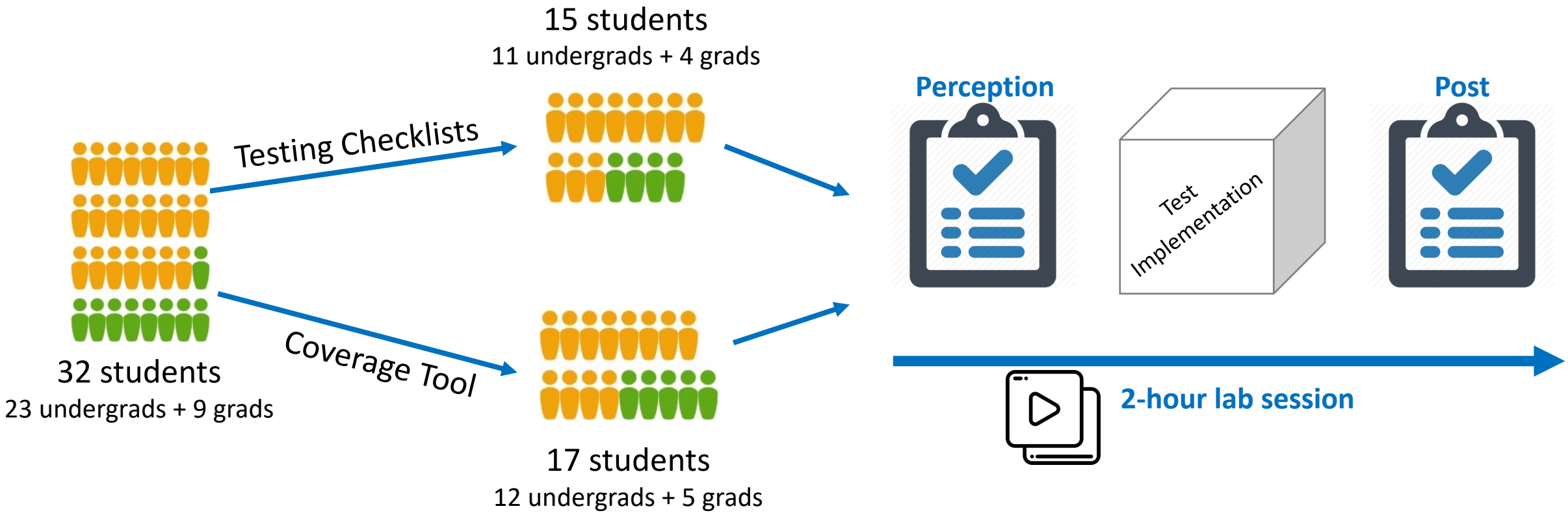


	avg_yrJava	avg_yrUT
Undergrads	3.5	3.0
Grads	**0.1**	**0.4**

	avg_yrJava	avg_yrUT
Undergrads	3.8	2.7
Grads	1.4	1.4

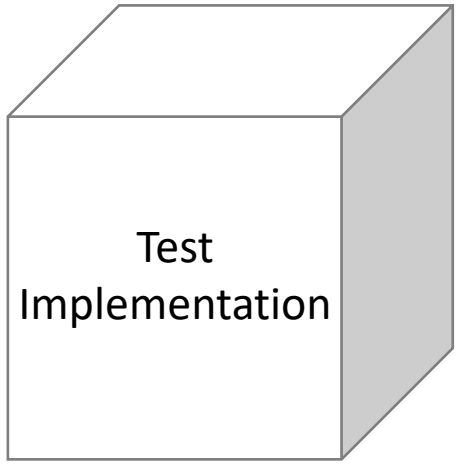
Gina R. Bai – rbai2@ncsu.edu

Methodology



Gina R. Bai – rbai2@ncsu.edu

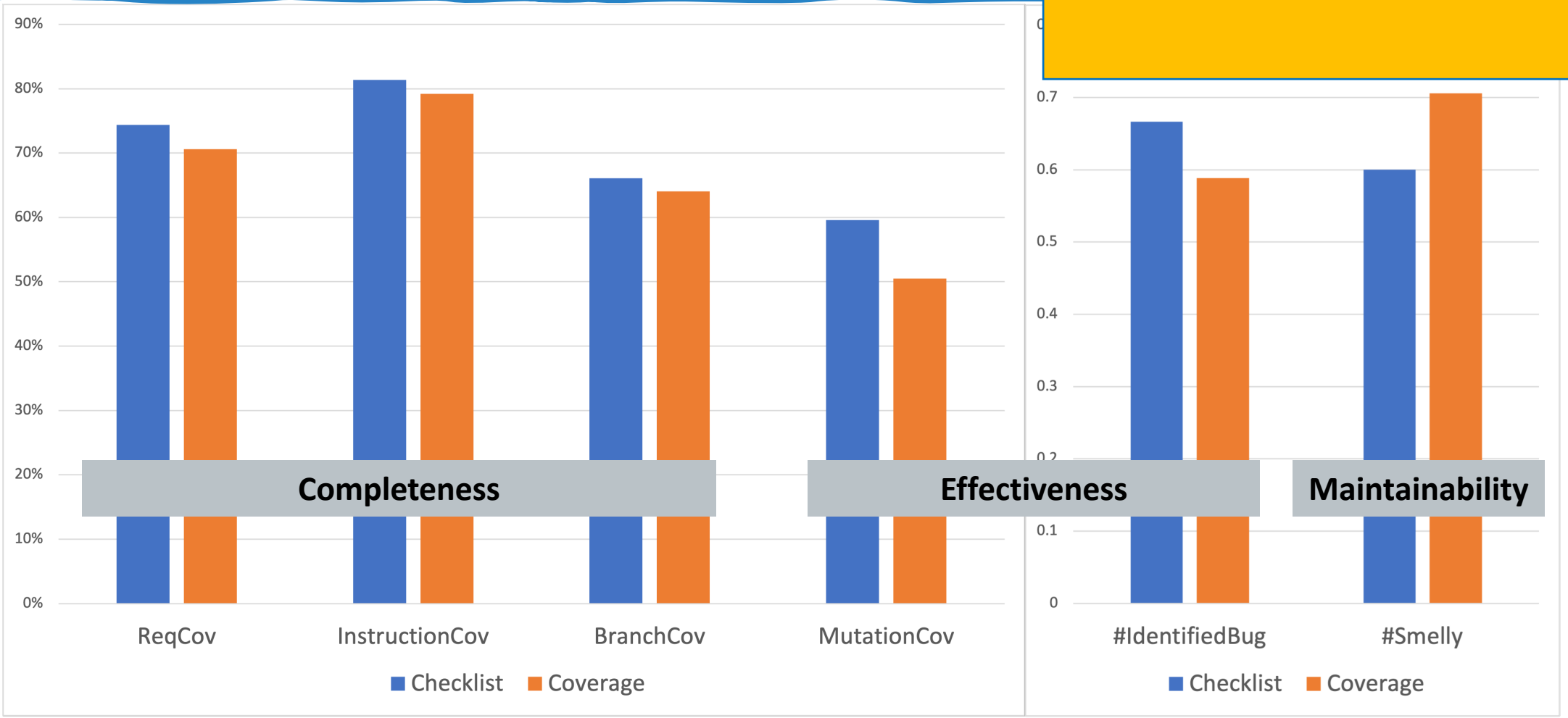
Measurements of Test Code Quality



- **Completeness**
  - Requirements coverage
  - Instruction coverage
  - Branch coverage
- **Effectiveness**
  - Mutation coverage
  - The number of identified seeded bugs
- **Maintainability**
  - The number of smelly tests

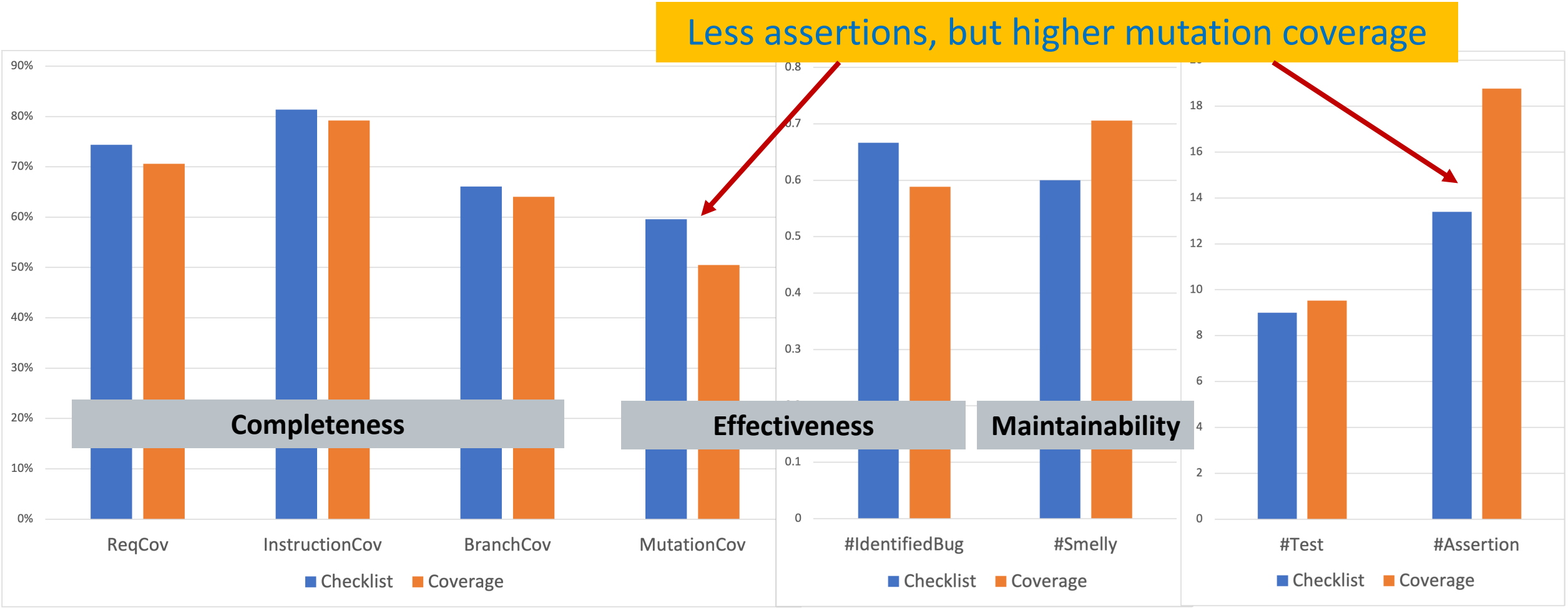
Checklists vs. Coverage T

Tool support does not need to be sophisticated to be effective!



Gina R. Bai – rbai2@ncsu.edu

Checklists vs. Coverage Tools



Gina R. Bai – rbai2@ncsu.edu

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

Gina R. Bai – rbai2@ncsu.edu

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

Gina R. Bai – rbai2@ncsu.edu