CS 4530: Fundamentals of Software Engineering
Module 3.1: Trusting TypeScript (or not!)

Adeel Bhutta, Rob Simmons, and Mitch Wand

Khoury College of Computer Sciences

Learning Goals for this Lesson

At the end of this lesson, you should be able to

- Explain how TypeScript types and documented preconditions influence what tests you need to write
- Explain the difference between the any vs unknown types in TypeScript

What Inputs Should We Test?

What input values do I need to test this function on?

```
/**
 * Prints "hello" repeatedly
 *
 * @param numHellos - number of times to print “hello”,
 *   must be an integer >= 0
 */
function helloNTimes(numHellos: number) {
  for (let i = numHellos; i !== 0; i--) {
    console.log('hello');
  }
}
```

What Inputs Should We Test?

What input values do I need to test this function on?

- Edge cases (definitely 0)
- Probably 1 and some larger number?
  But most numbers > 1 are kind of interchangeable.
  - If we want to sound fancy, we can call these “equivalence classes of inputs.”
- What about -3? 1.4? NaN? null? { lol: 'owned' } ?

```
/**
 * Prints "hello" repeatedly
 *
 * @param numHellos - number of times to print “hello”,
 *   must be an integer >= 0
 */
function helloNTimes(numHellos: number)
```

TypeScript Types Cannot Be Trusted

- TypeScript types are, at the end of the day,
  no better than preconditions mentioned in comments.

```
helloNTimes({ lol: 'owned ' } as unknown as number)
```

- They do at least make it less likely you'll screw up
  accidentally…

TypeScript Types… Can Be Trusted?

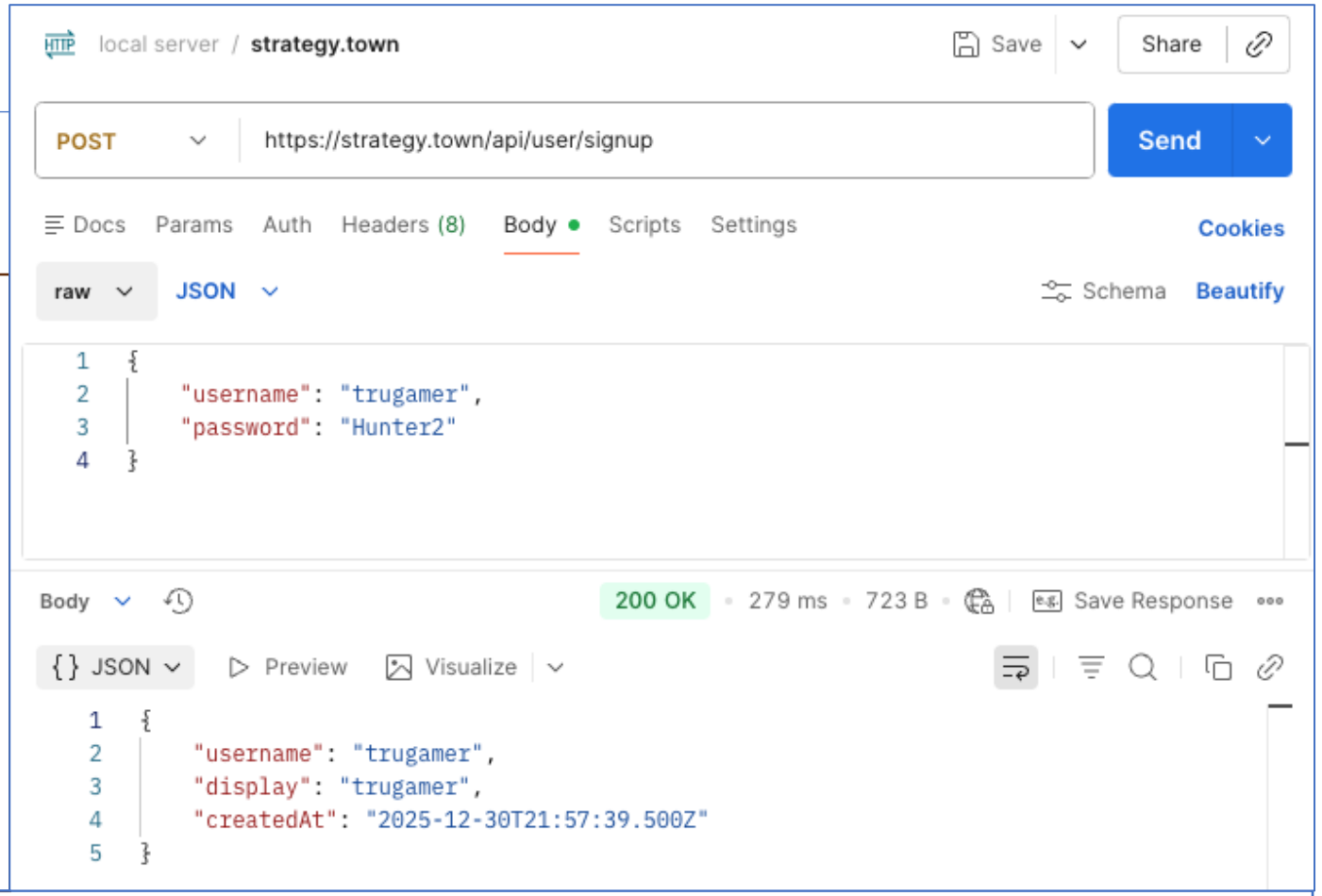
- If you use TypeScript with care, you can rely on the control it gives you over what might get passed to the function
  - If a function is only being called from other sources that respect contracts… then you can rely on the contracts being respected?
  - Don’t have contracts on functions that won’t be respected!

What Trusting Contracts Looks Like

```
/**
 * Adds a message to a chat, updating the chat
 *
 * @param chatId - Ostensible chat id
 * @param user - Authenticated user
 * @param messageId - Valid message id
 * @returns the updated chat info object
 * @throws if the chat id is not valid
 */
export function addMessageToChat(
  chatId: string,
  user: UserWithId,
  messageId: string
): ChatInfo {
```

Untrusted Inputs

Any input given to a web app can also be given by other means…

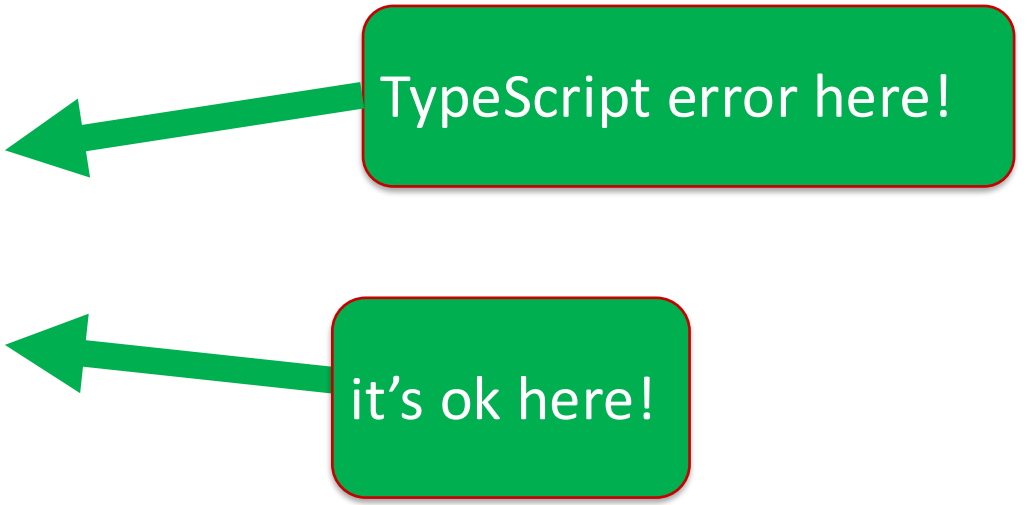



```
curl https://strategy.town/api/user/signup -H 'Content-Type: application/json' \
  --data '{ "username": "trugamer", "password": "Hunter2" }'
```

Untrusted Inputs should be unknown

- The appropriate TypeScript type for an unknown value is unknown

```
function lookAtMe(input: unknown) {
  console.log(input.toUpperCase());
  if (typeof input === "string") {
    console.log(input.toUpperCase());
  }
}
```



- If you use the any type instead, TypeScript will just say "ok, I guess you know what you're doing"

Untrusted Inputs Should be “unknown”

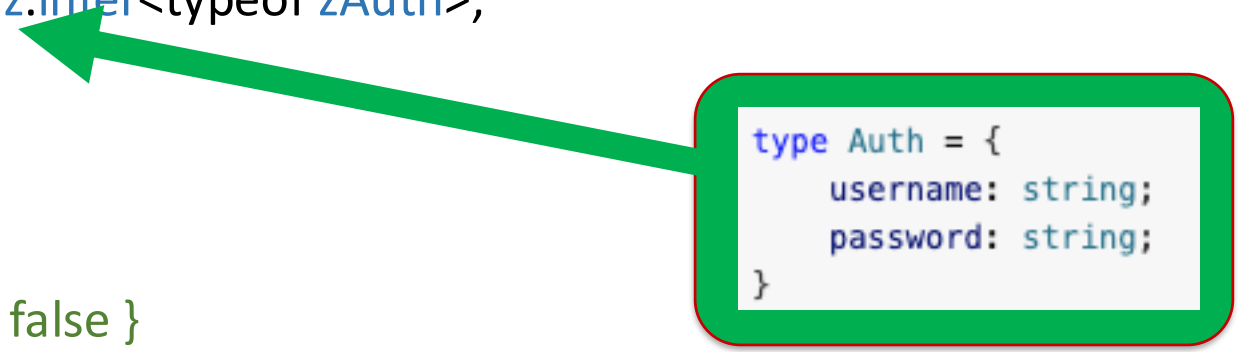
This can get complicated fast…

```
type Auth = { username: string, password: string }

function useAuth(x: unknown) {
  if (
    (typeof x === 'object' && x !== null) &&
    ('username' in x && typeof x.username === 'string’) &&
    ('password' in x && typeof x.password === 'string’)
  ) {
    const auth: Auth = { username: x.username, password: x.password };
    // write the code you care about here!
  }
}
```

Libraries Make Checking Types Easier

Zod is a library that makes testing the structure of inputs less tedious and error-prone.

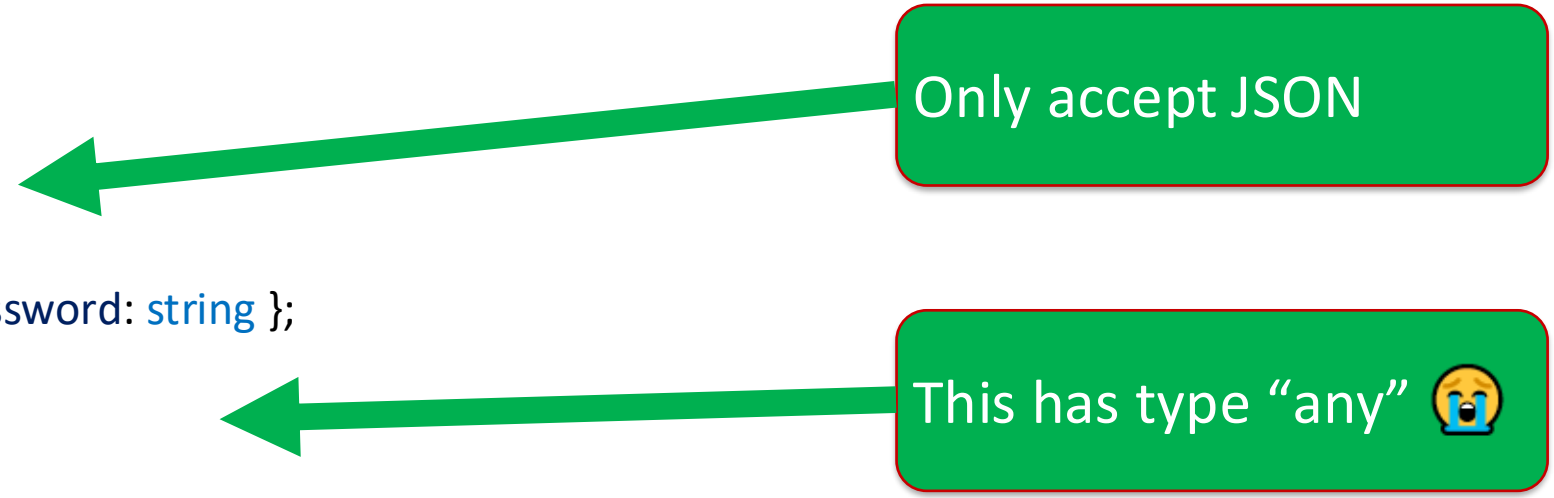
```
import { z } from 'zod';

const zAuth = z.object({ username: z.string(), password: z.string() });
type Auth = z.infer<typeof zAuth>;
```




```
// { success: false }
console.log(zAuth.safeParse({ username: 4, password: null }))
// { success: true, data: { username: "", password: "" } }
console.log(zAuth.safeParse({ username: "", password: "" }))
```

Using “any”: Common, Not Great

```
import express from 'express';
const app = express();
app.use(express.json());

type Auth = { username: string; password: string };
app.post('/', (req, res) => {
  const auth: Auth = req.body;


  if (auth.password !== 'secret') {
    res.status(403).send({ error: 'Wrong password' });
  } else {
    res.send({ message: `WELCOME,${auth.username.toUpperCase()}` });
  }
});

app.listen(8000, () => console.log(`Listening on port 8000`));
```



Zod In A Tiny Web Server

```
import { z } from 'zod';
import express from 'express';
const app = express();
app.use(express.json());

const zAuth = z.object({ username: z.string(); password: z.string() });
app.post('/', (req, res) => {
  const auth = zAuth.safeParse(req.body);
  if (auth.error) {
    res.status(400).send({ error: 'Unexpected message' });
  } else if (auth.data.password !== 'secret') {
    res.status(403).send({ error: 'Wrong password' });
  } else {
    res.send({ message: `WELCOME,${auth.data.username.toUpperCase()}` });
  }
});

app.listen(8000, () => console.log(`Listening on port 8000`));
```

Review

- One view of TypeScript is that it's a handy way of documenting, and *imperfectly* checking, the contracts (preconditions and postconditions) of your code
- Do you need to test inputs that violate your contracts? It depends!
- You can never trust that the input to a web server will obey any sort of contract — important to test!

TypeScript Types… Can Be Trusted?

- If you use TypeScript with care, you can rely on the control it gives you over what might get passed to the function
  - If a function is only being called from other sources that respect contracts… then you can rely on the contracts being respected?
  - Don’t have contracts on functions that won’t be respected!
- Sometimes you *cannot* control your input, like when it’s coming directly from a user or a network request
- The unknown type (NOT the any type) helps in these situations

TypeScript (can) Make Testing Trickier!

- TypeScript can lull you into a false sense of security that you *don't* need to check whether your number-accepting function can handle a string.
- It makes sense *sometimes* to treat your precondition comments as not-needing-to-be tested
- It makes sense *often* to treat your TypeScript types as not-needing-to-be-tested, especially if you use TypeScript appropriately (avoid the use of the any type!)

(Un)authenticated Inputs

With difficulty, you can check an unknown value

```
type Auth = { username: string, password: string }

function isAuth(x: unknown) : x is Auth {
  return (
    typeof x === 'object' &&
    x !== null &&
    ('username' in x && typeof x.username === 'string') &&
    ('password' in x && typeof x.password === 'string')
  );
}
```

"unknown," NOT "any," is the type for unauthenticated inputs

If a value passes this function, TypeScript will just assume it's an Auth, *even if you write the function wrong*

Why do we test?

- Test Driven Development
  - If we start with tests, passing the tests lets us know we're done meeting the conditions of satisfaction
- Acceptance Testing
  - Does the customer agree we've met the conditions of satisfaction?
- Regression Testing
  - Did something change since some previous version?
  - Prevent bugs from (re-)entering during maintenance.
  - "Good" test suite detects bugs that (inevitably) get added as software is developed over time: tests are for *the future*

What makes for a good test (suite)?

- Desirable properties of test suites:
  - Find bugs
  - Run automatically
  - Are relatively cheap to run
  - Don't depend on the order of tests.
- Desirable properties of individual tests:
  - Understandable and debuggable
  - No false alarms (not “flaky”)

Code Coverage

- The industry standard answer for “have I written enough tests”
- Measures “how much of your code” is exercised by your tests
- If none of your test even *execute* a piece of code, it’s definitely not being tested!

Code Coverage

- *Line* and *Statement* coverage: coarsest measure.
- Testing x = 0 exercises lines 1 and 2
- Testing x = 10 exercises lines 1, 4, 5, and 6.

```
1| if (x === 0) {
2|  return 3;
3| }
4| const y = x > 4 ? 2 : 3;
5| const z = x % 2 == 0 ? 1 : 2;
6| return x / (y - z);
```

Code Coverage

- *Branch* coverage: most widely used in industry.
- Testing with x > 4 and x <= 4 necessary to handle both branches on line 4.
- Testing with odd and even numbers necessary to handle both branches on line 5.
- The values -2, 0, 1, and 10 get full branch coverage.

```
1| if (x === 0) {
2|  return 3;
3| }
4| const y = x > 4 ? 2 : 3;
5| const z = x % 2 == 0 ? 1 : 2;
6| return x / (y - z);
```

Code Coverage

- The values -2, 0, 1, and 10 get full branch coverage…
- …but 5 causes line 6 to divide by zero!
  - In JavaScript/TypeScript, this doesn't cause an exception, there's a number called "NaN" for "not a number"
- *Path* coverage covers all combinations of branches; it is infeasible in practice.

```
1| if (x === 0) {
2|  return 3;
3| }
4| const y = x > 4 ? 2 : 3;
5| const z = x % 2 == 0 ? 1 : 2;
6| return x / (y - z);
```

Code Coverage

- Total code coverage — by any metric — does not mean no bugs
  - Running code doesn't mean checking that it's doing the right thing! (important to Assemble/Act/Assess)
- Coverage checking can be invaluable at identifying when you *think* you're testing something but you're not, which is a real problem in practice.
  - Test-Driven Development also valuable for this problem: it's important that tests switch from failing to succeeding *when you expect them to*

Code Coverage

- Vitest makes it easy to check code coverage

```
  calculator/add
    ✓ should return a number when parameters are passed to `add()`
    ✓ should return sum of `2` when 1 + 1 is passed to `add()`

  calculator/subtract
    ✓ should return a number when parameters are passed to `subtract()`
    ✓ should return sum of `1` when 2 - 1 is passed to `subtract()`


  4 passing (4ms)

-------------|----------|----------|----------|----------|-------------------|
File         |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
-------------|----------|----------|----------|----------|-------------------|
All files    |      100 |      100 |      100 |      100 |                   |
 Add.ts      |      100 |      100 |      100 |      100 |                   |
 Subtract.ts |      100 |      100 |      100 |      100 |                   |
-------------|----------|----------|----------|----------|-------------------|
```

Code Coverage: Review

- A lot of what you've been taught about "edge cases" is can be understood in terms of trying to imagine inputs that will provide code coverage for an unknown function.
- What inputs should we start with for code coverage if:
  - We're taking numbers that are supposed to represent a 5-digit zip code?
  - We're taking strings that are supposed to represent a 5-digit zip code?
  - We're taking an array of strings that represent?

Adversarial Testing

- It can be helpful to think of testing as a game in which you play against an adversary.
- Your adversary plays by producing multiple versions of code that you agree is buggy, and multiple versions of code you agree is correct.
- Your win if your tests catch all the buggy code, and pass all the correct code.

Adversarial Testing

Original code (correct)

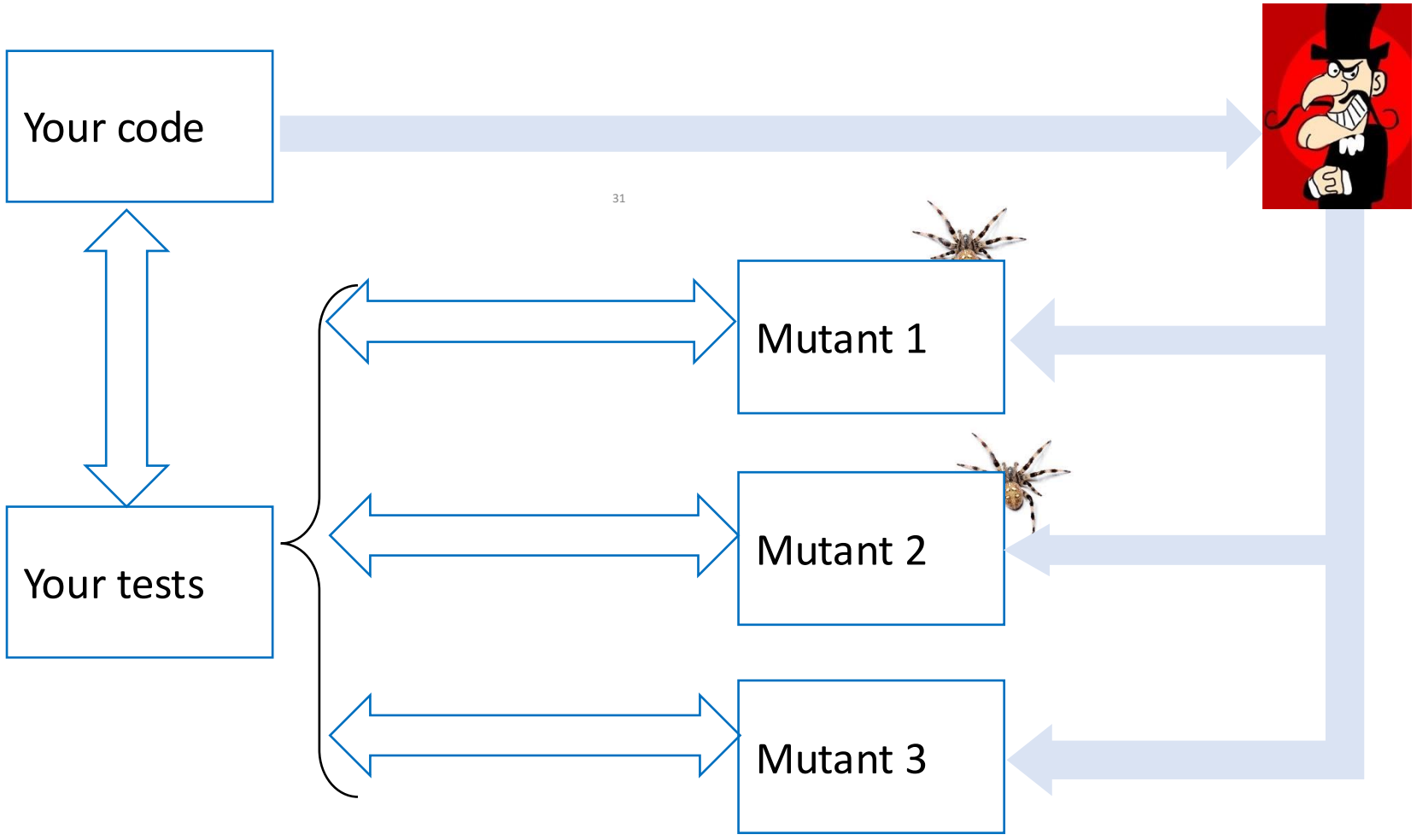
```
// find the first item in the list that is
// greater than or equal to the target.
export default function search(list:number[], target:number) {
    return list.find((item) => item >= target);
}
```

Mutated code (buggy)

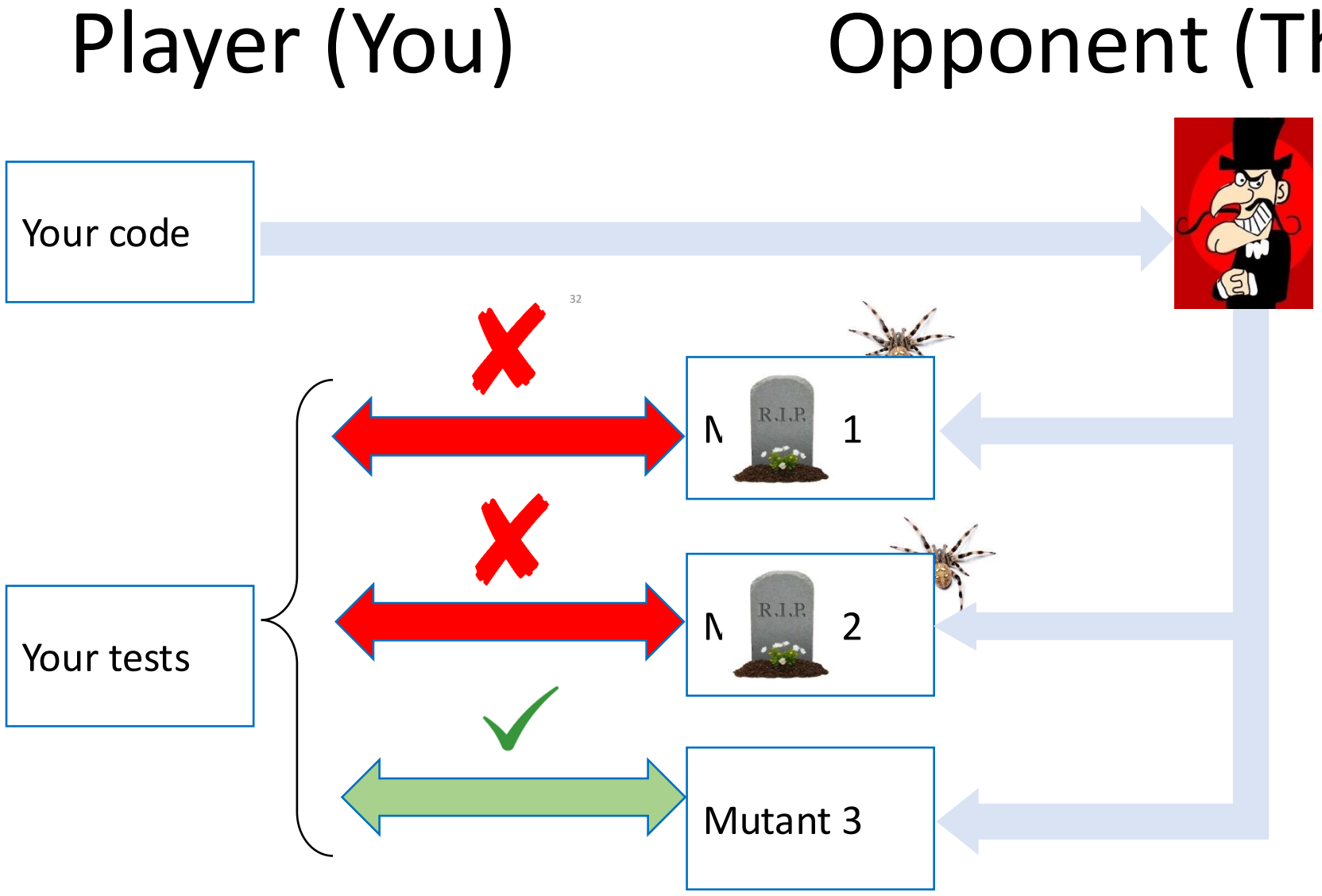
```
// find the first item in the list that is
// greater than or equal to the target.
export default function search(list:number[], target:number) {
    return list.find((item) => item > target)
}
```

Adversarial Testing: The Opening

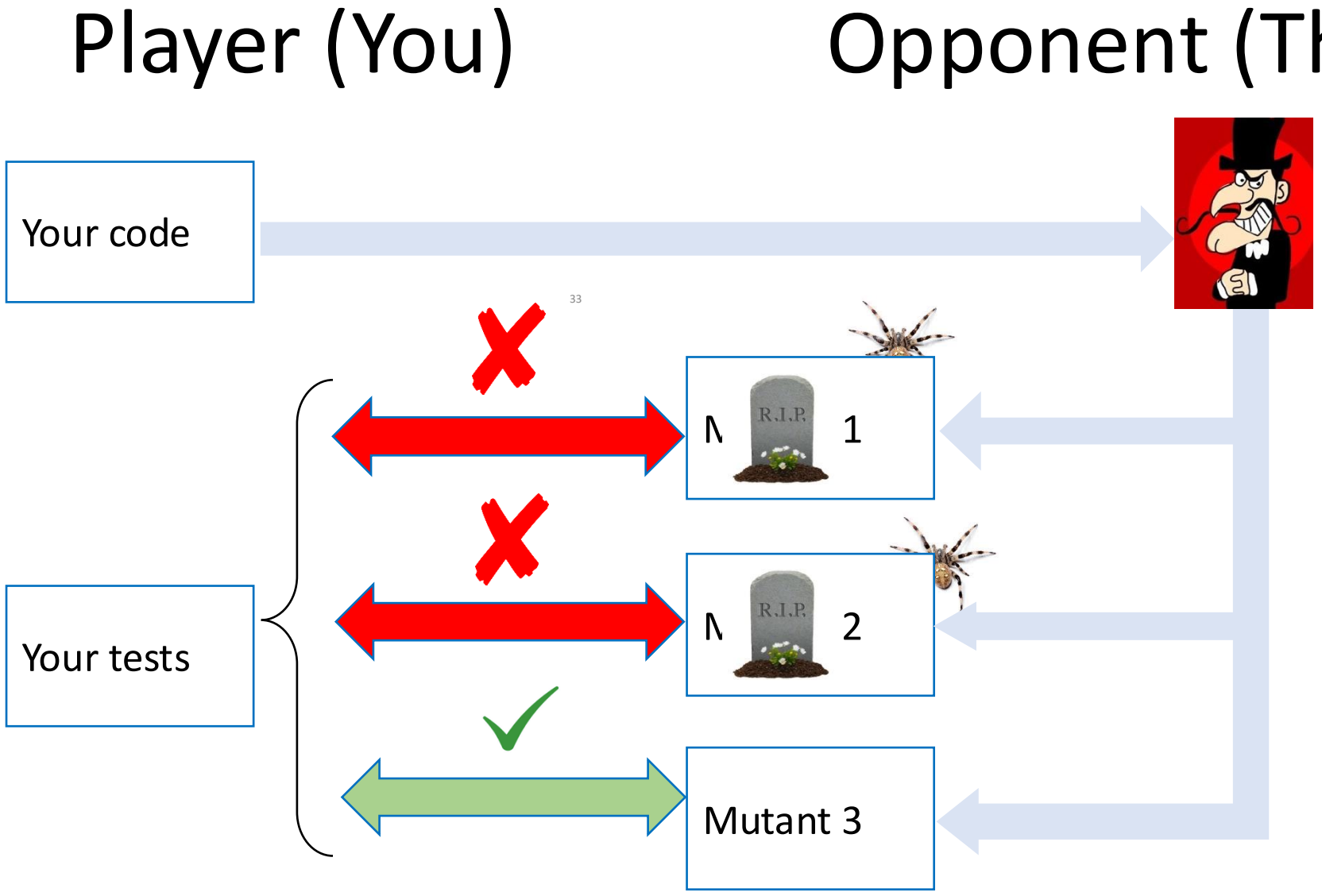
Player (You) Opponent (Them)



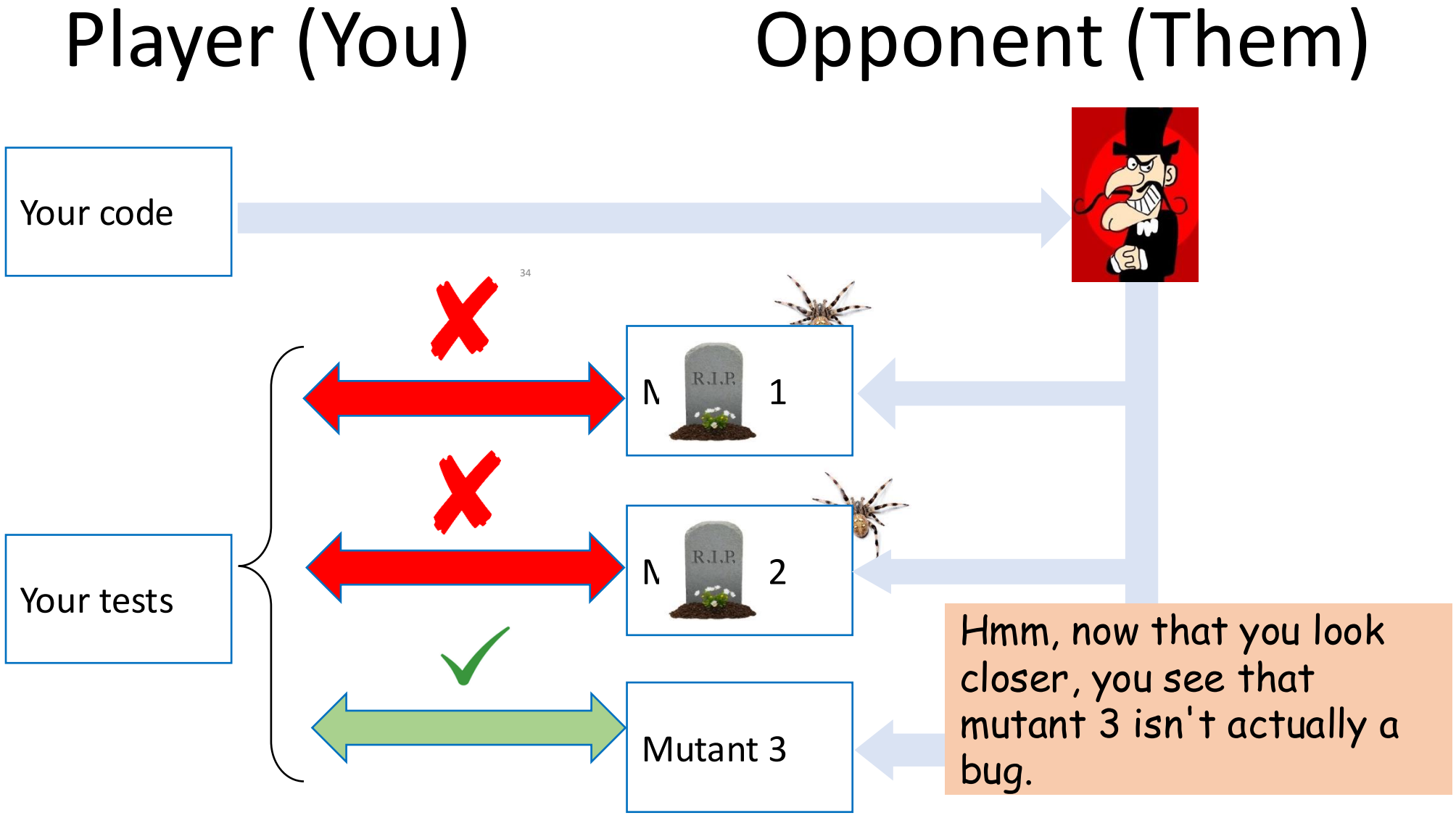
Adversarial Testing: A round of play



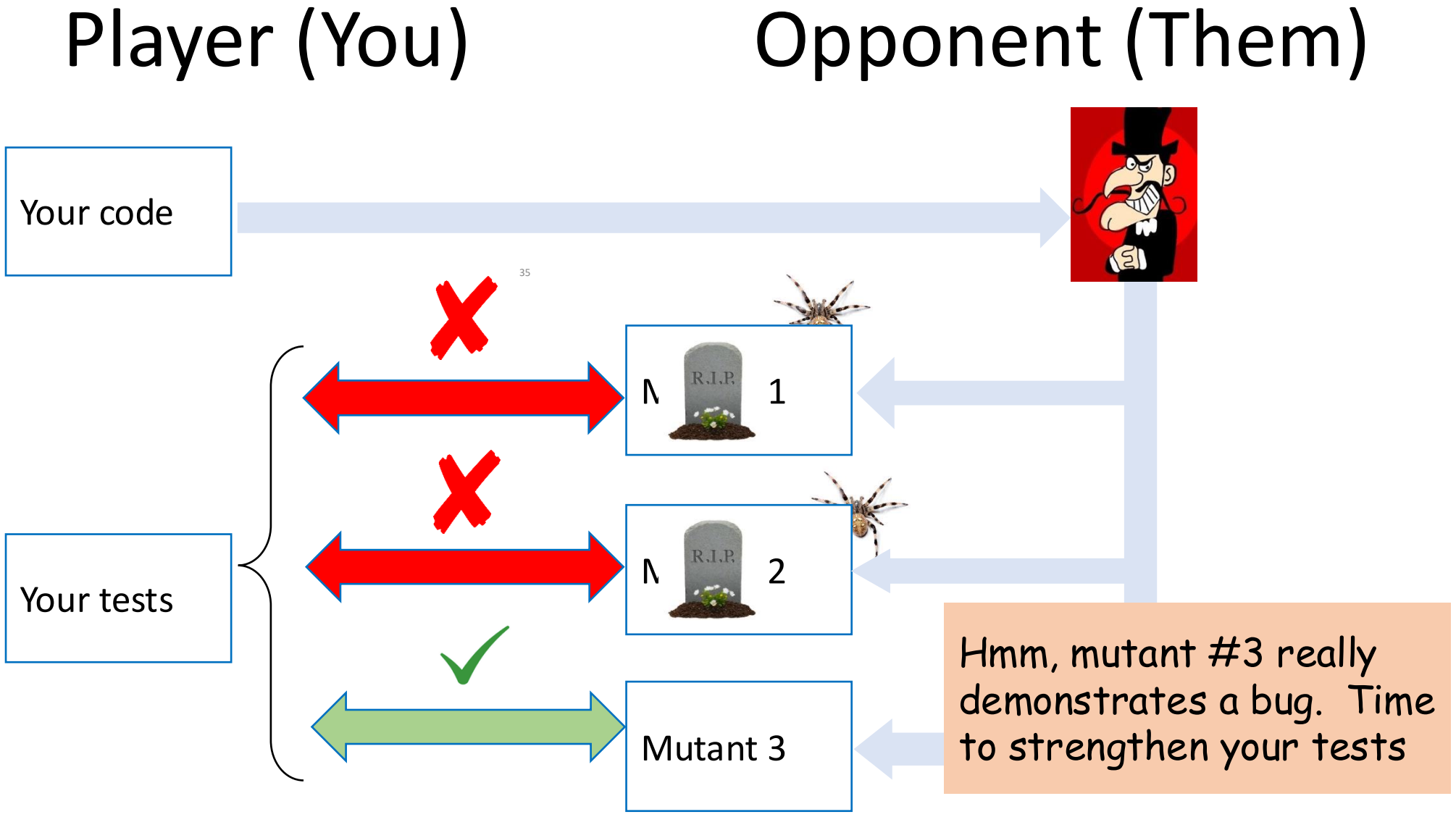
Adversarial Testing: A round of play



Adversarial Testing: a winning position

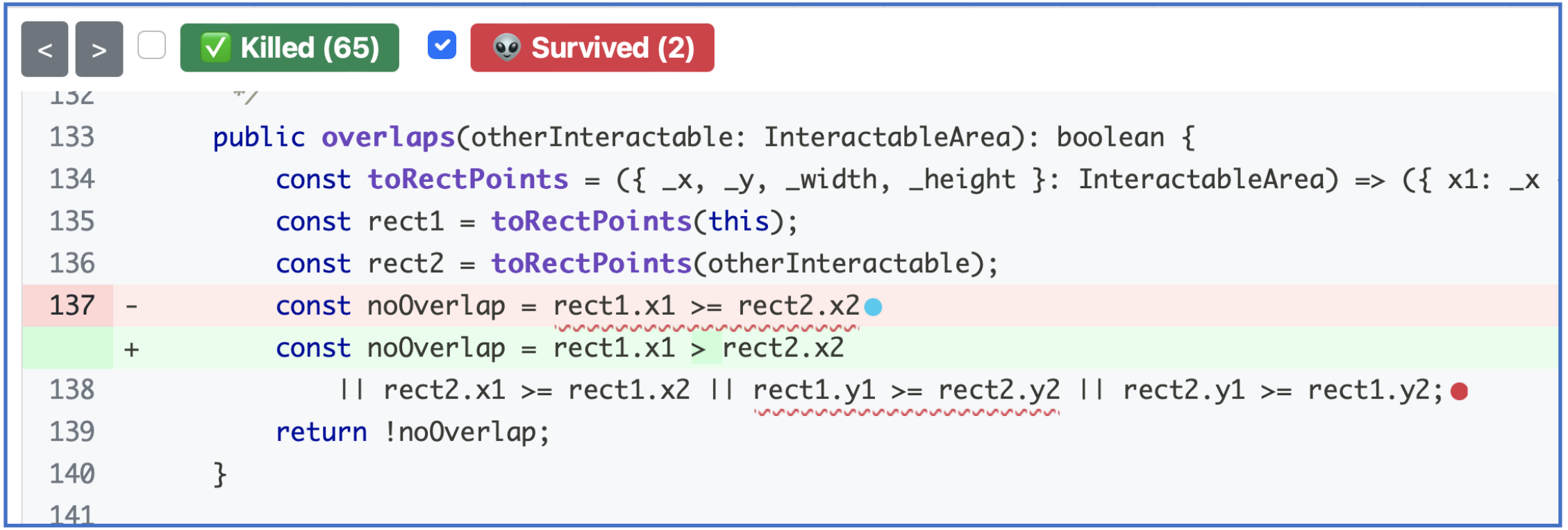


Adversarial Testing: you lose



Adversarial Testing

Stryker is a *mutation tester* for JavaScript — an automated adversary!



Adversarial Testing

Stryker is a mutation tester for JavaScript — an automated adversary!

Sometimes it loses the game because mutants aren't bugs.

```
62        public static fromMapObject(mapObject: ITiledMapObject, broadcas
63            const { name, width, height } = mapObject;
64            if (!width || !height) {
65  -             throw new Error(`Malformed viewing area ${name}`);
    +             throw new Error(``);
66            }
67            const rect: BoundingBox = { x: mapObject.x, y: mapObject.y, w
68            return new ConversationArea({ id: name, occupantsByID: [] },
69        }
```

Are mutants a Valid Substitute for Real Faults? Probably yes.

- Do mutants really represent real bugs?
- Researchers have studied the question of whether a test suite that finds more mutants also finds more real faults
- Conclusion: For the 357 real faults studied, yes
- This work has been replicated in many other contexts, including with real faults from student code

Are Mutants a Valid Substitute for Real Faults in Software Testing?

René Just†, Darioush Jalali†, Laura Inozemtseva*, Michael D. Ernst†, Reid Holmes*, and Gordon Fraser‡
†University of Washington, Seattle, WA, USA, {rjust, darioush, mernst}@cs.washington.edu
*University of Waterloo, Waterloo, ON, Canada, {lminozem, rtholmes}@uwaterloo.ca
‡University of Sheffield, Sheffield, UK, gordon.fraser@sheffield.ac.uk

ABSTRACT

A good test suite is one that detects real faults. Because the set of faults in a program is usually unknowable, this definition is not useful to practitioners who are creating test suites, nor to researchers who are creating and evaluating tools that generate test suites. In place of real faults, testing research often uses mutants, which are artificial faults — each one a simple syntactic variation — that are systematically seeded throughout the program under test. Mutation analysis is appealing because large numbers of mutants can be automatically-generated and used to compensate for low quantities or the absence of known real faults.

Unfortunately, there is little experimental evidence to support the use of mutants as a replacement for real faults. This paper investigates whether mutants are indeed a valid substitute for real faults, i.e., whether a test suite's ability to detect mutants is correlated with its ability to detect real faults that developers have fixed. Unlike prior studies, these investigations also explicitly consider the confounding effects of code coverage on the mutant detection rate.

Our experiments used 357 real faults in 5 open-source applications that comprise a total of 321,000 lines of code. Furthermore, our experiments used both developer-written and automatically-generated test suites. The results show a statistically significant correlation between mutant detection and real fault detection, independently of code coverage. The results also give concrete suggestions on how to improve mutation analysis and reveal some inherent limitations.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Measurement

Keywords

Test effectiveness, real faults, mutation analysis, code coverage

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2635929

1. INTRODUCTION

Both industrial software developers and software engineering researchers are interested in measuring test suite effectiveness. While developers want to know whether their test suites have a good chance of detecting faults, researchers want to be able to compare different testing or debugging techniques. Ideally, one would directly measure the number of faults a test suite can detect in a program. Unfortunately, the faults in a program are unknown a priori, so a proxy measurement must be used instead.

A well-established proxy measurement for test suite effectiveness in testing research is the *mutation score*, which measures a test suite's ability to distinguish a program under test, the *original version*, from many small syntactic variations, called *mutants*. Specifically, the mutation score is the percentage of mutants that a test suite can distinguish from the original version. Mutants are created by systematically injecting small artificial faults into the program under test, using well-defined *mutation operators*. Examples of such mutation operators are replacing arithmetic or relational operators, modifying branch conditions, or deleting statements (cf. [18]).

Mutation analysis is often used in software testing and debugging research. More concretely, it is commonly used in the following use cases (e.g., [3, 13, 18, 19, 35, 37–39]):

Test suite evaluation The most common use of mutation analysis is to evaluate and compare (generated) test suites. Generally, a test suite that has a higher mutation score is assumed to detect more real faults than a test suite that has a lower mutation score.

Test suite selection Suppose two unrelated test suites T_1 and T_2 exist that have the same mutation score and $|T_1| < |T_2|$. In the context of test suite selection, T_1 is a preferable test suite as it has fewer tests than T_2 but the same mutation score.

Test suite minimization A mutation-based test suite minimization approach reduces a test suite T to $T \setminus \{t\}$ for every test $t \in T$ for which removing t does not decrease the mutation score of T.

Test suite generation A mutation-based test generation (or augmentation) approach aims at generating a test suite with a high mutation score. In this context, a test generation approach augments a test suite T with a test t only if t increases the mutation score of T.

Fault localization A fault localization technique that precisely identifies the root cause of an artificial fault, i.e., the mutated code location, is assumed to also be effective for real faults.

These uses of mutation analysis rely on the assumption that mutants are a valid substitute for real faults. Unfortunately, there is little experimental evidence supporting this assumption, as discussed in greater detail in Section 4. To the best of our knowledge, only three previous studies have explored the relationship between mutants and

Adversarial Testing & Overspecification

We don't *want* our test to fail on non-bug mutants!

Example: is a mutant that changes the specific error message a valid change, or a bug, according to the English spec?

```
test("should throw an error if no such item", () => {
    const list = [1, 2, 3];
    const target = 4;
    expect(search(list, target)).toThrowError("No such item");
});
```

Maybe better? `toThrowError();`

Defensiveness has a cost

When you can’t control

```
type Auth = { username: string, password: string }

/**
 * Checks whether an unknown value is an `Auth`
 */
function isAuth(x: unknown) {
  return (
    typeof x === 'object' &&
    x !== null &&
    ('username' in x && typeof x.username === 'string’) &&
    ('password' in x && typeof x.password === 'string’)
  );
}
```

“unknown,” NOT “any,” is the type for unauthenticated inputs

Module Outline

- Lesson 3.1 Writing tests for TDD
- Lesson 3.2 Assessing Test Coverage
- Lesson 3.3 Adversarial Coverage Testing