2019
HOLYJS
MOSCOW

Types, Tests, and why flat-earthers are bad at QA

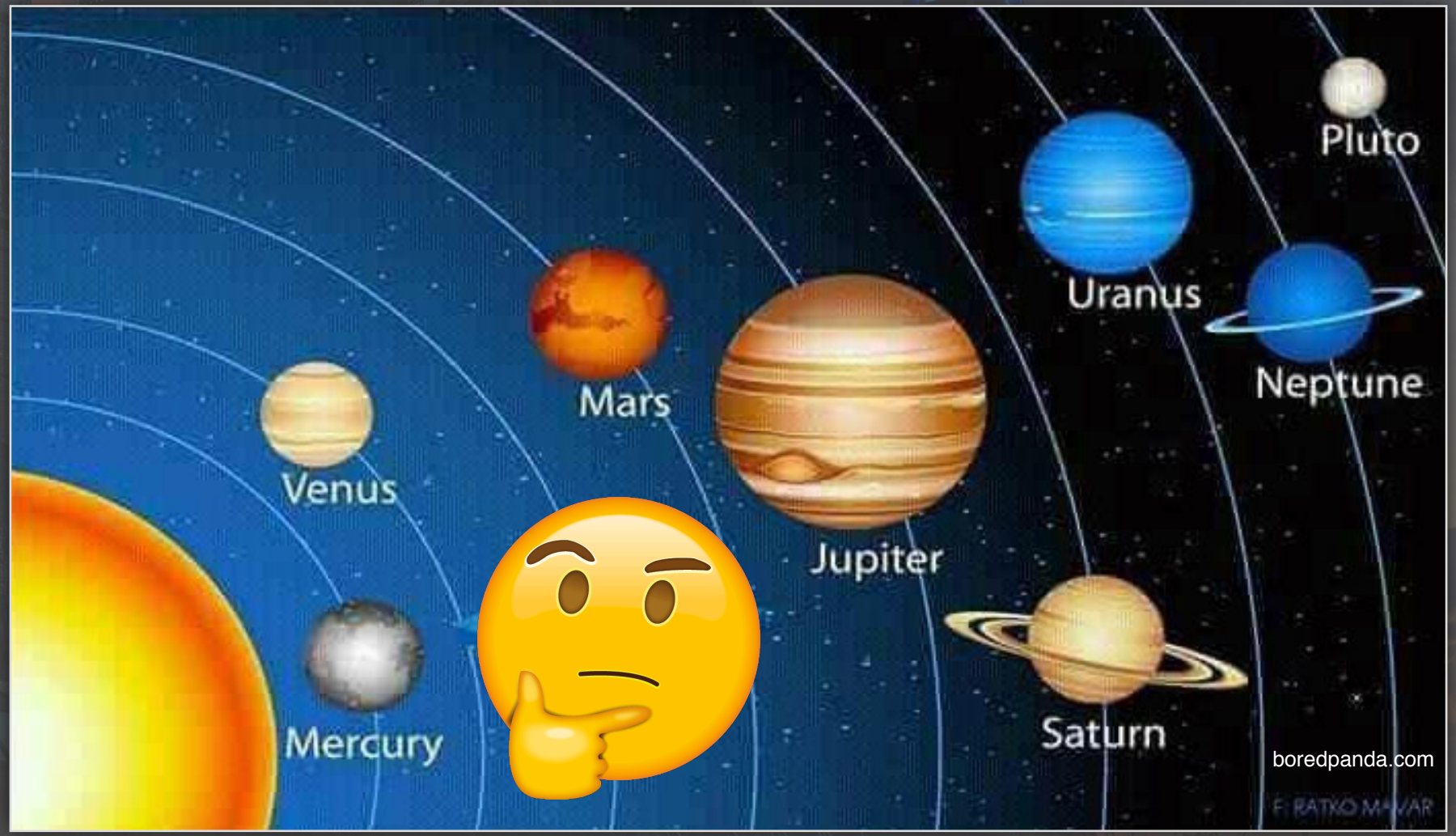
Twitter: @thewizardlucas

GitHub: @lucasfcosta

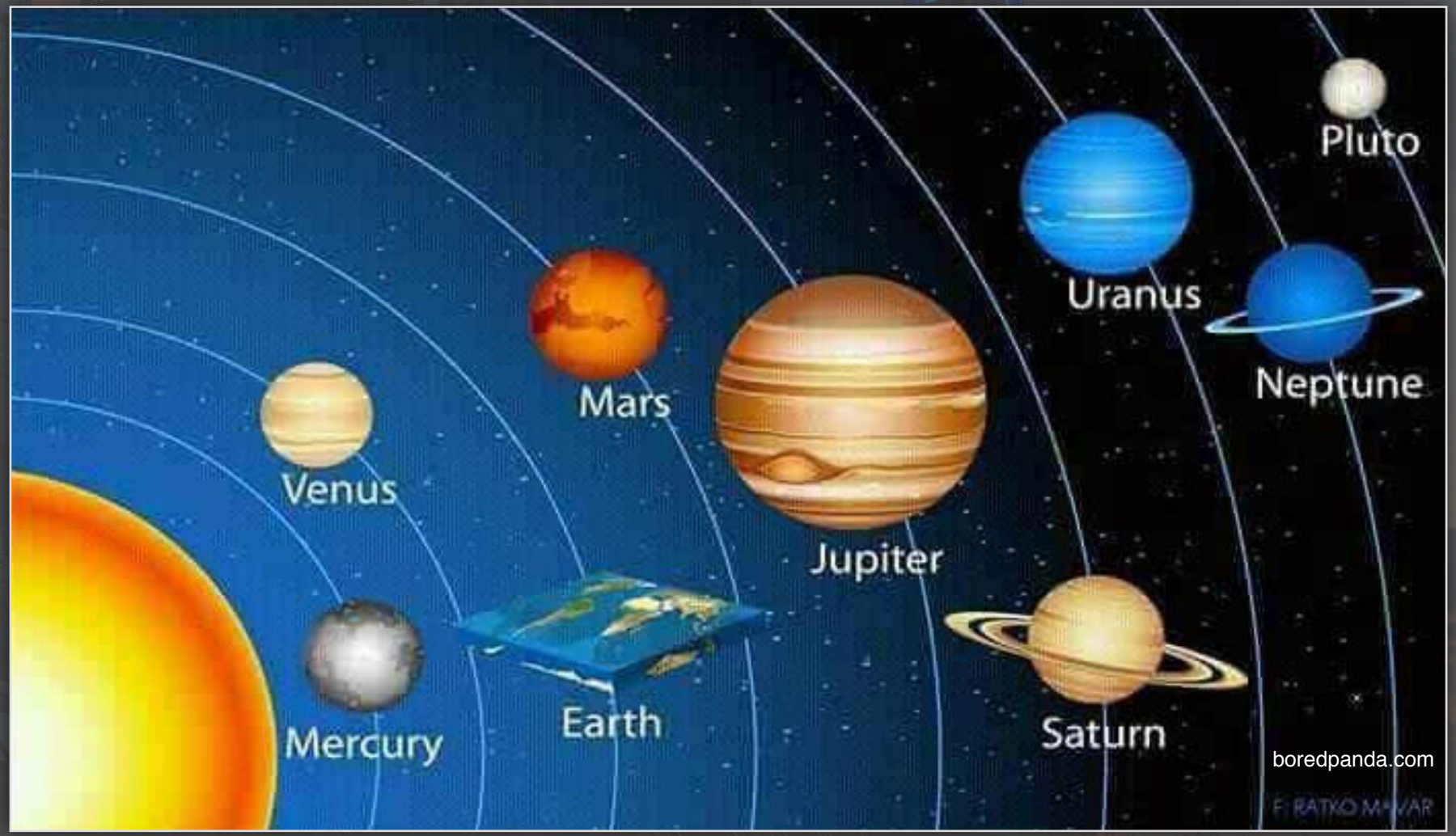
WWW: lucasfcosta.com

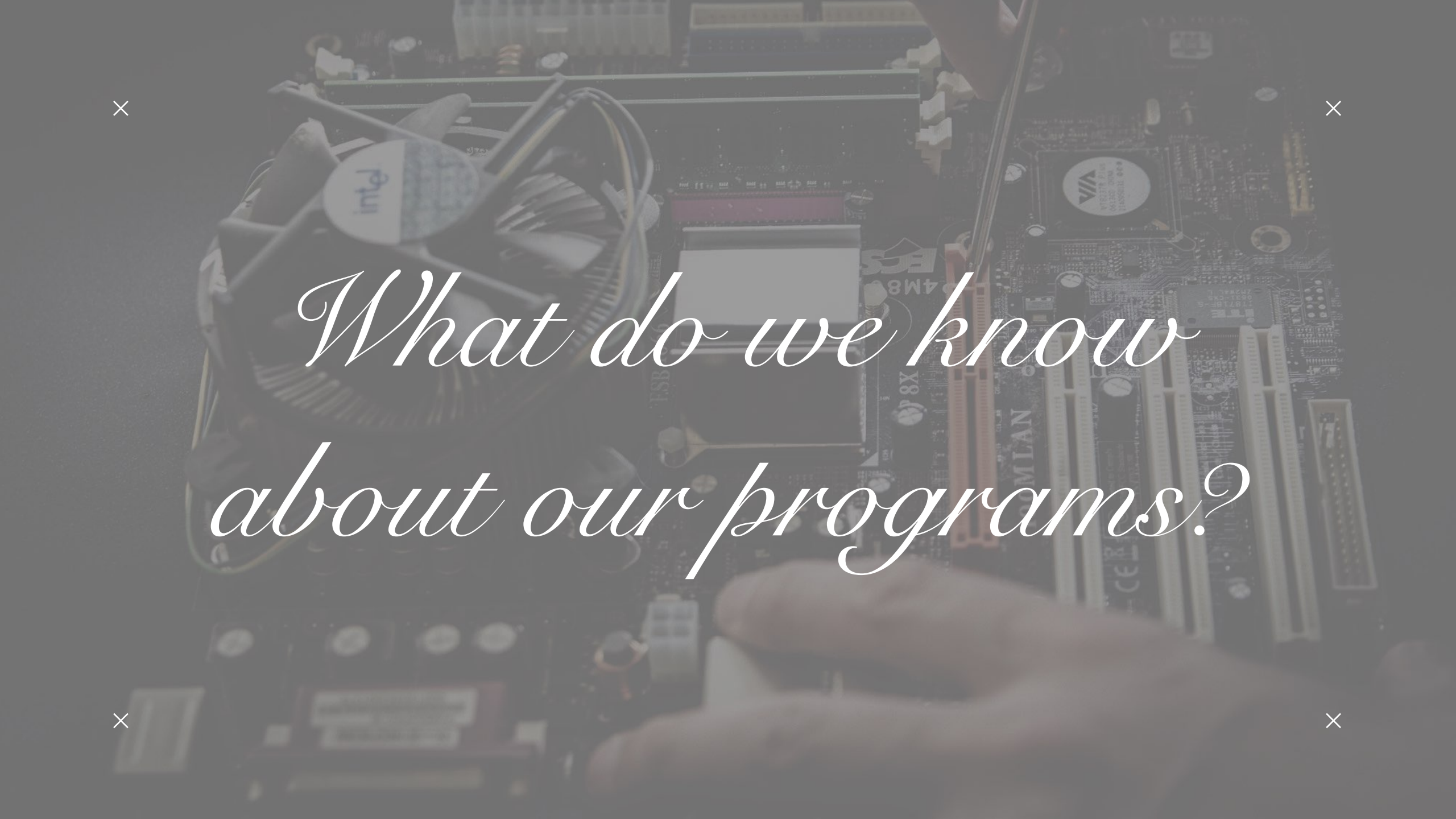
converge

2019
HOLYJS
MOSCOW



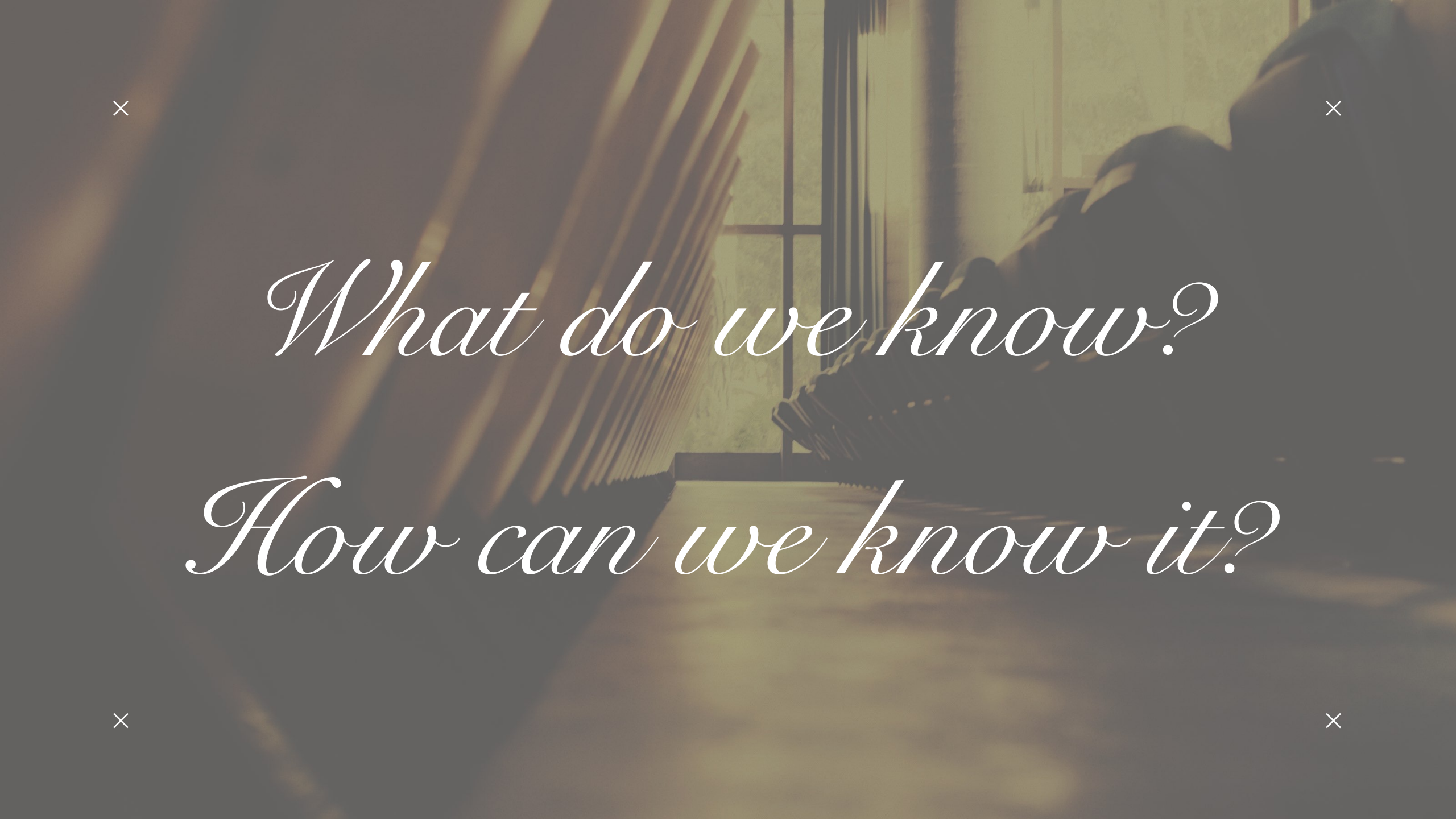
2019
HOLYJS
MOSCOW





What do we know about our programs?


What do we know?


What do we know?
How can we know it?

We must know our program to make sure it does what it is supposed to do.

How do we get to know the world?

EMPIRICISM

How do we get to know the world?

EMPIRICISM

How do we get to know the world?

RATIONALISM

TYPES

EMPIRICISM

How do we get to know our programs?

RATIONALISM

TESTS

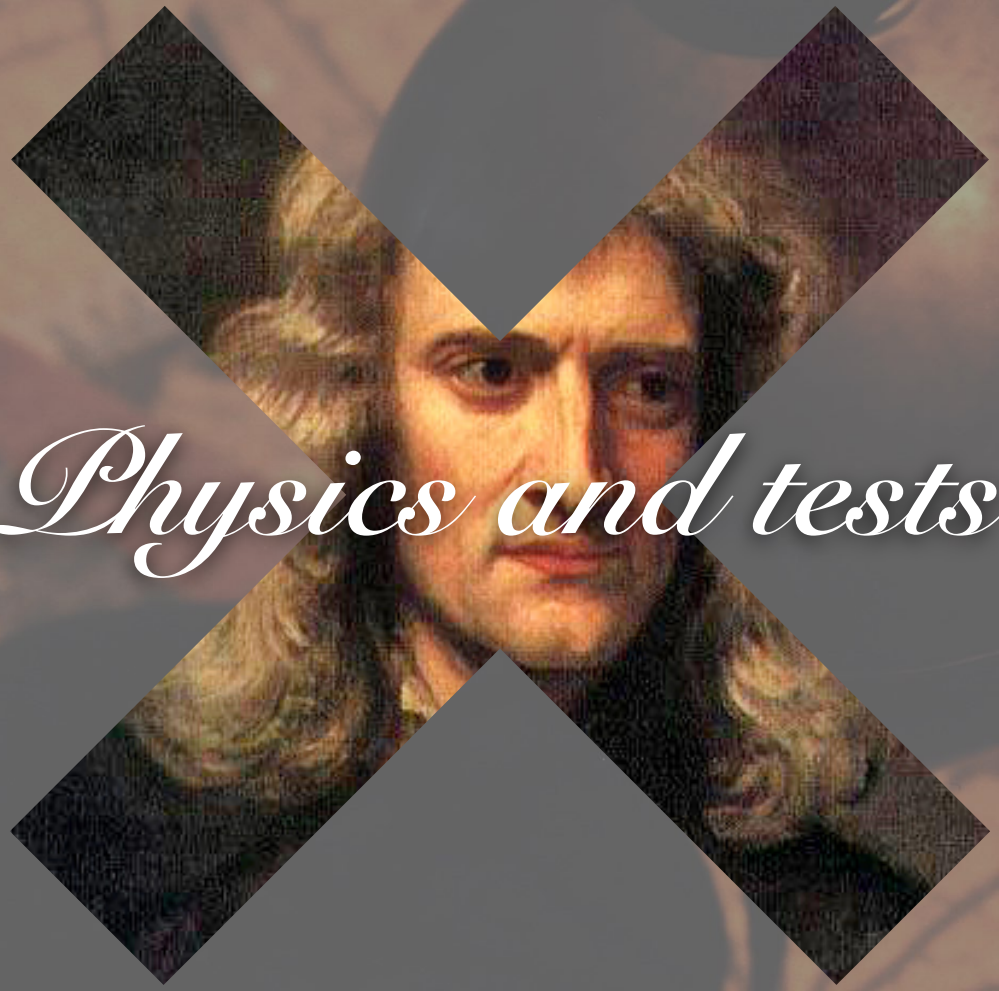

Programming
Epistemology

Physics and Mathematics

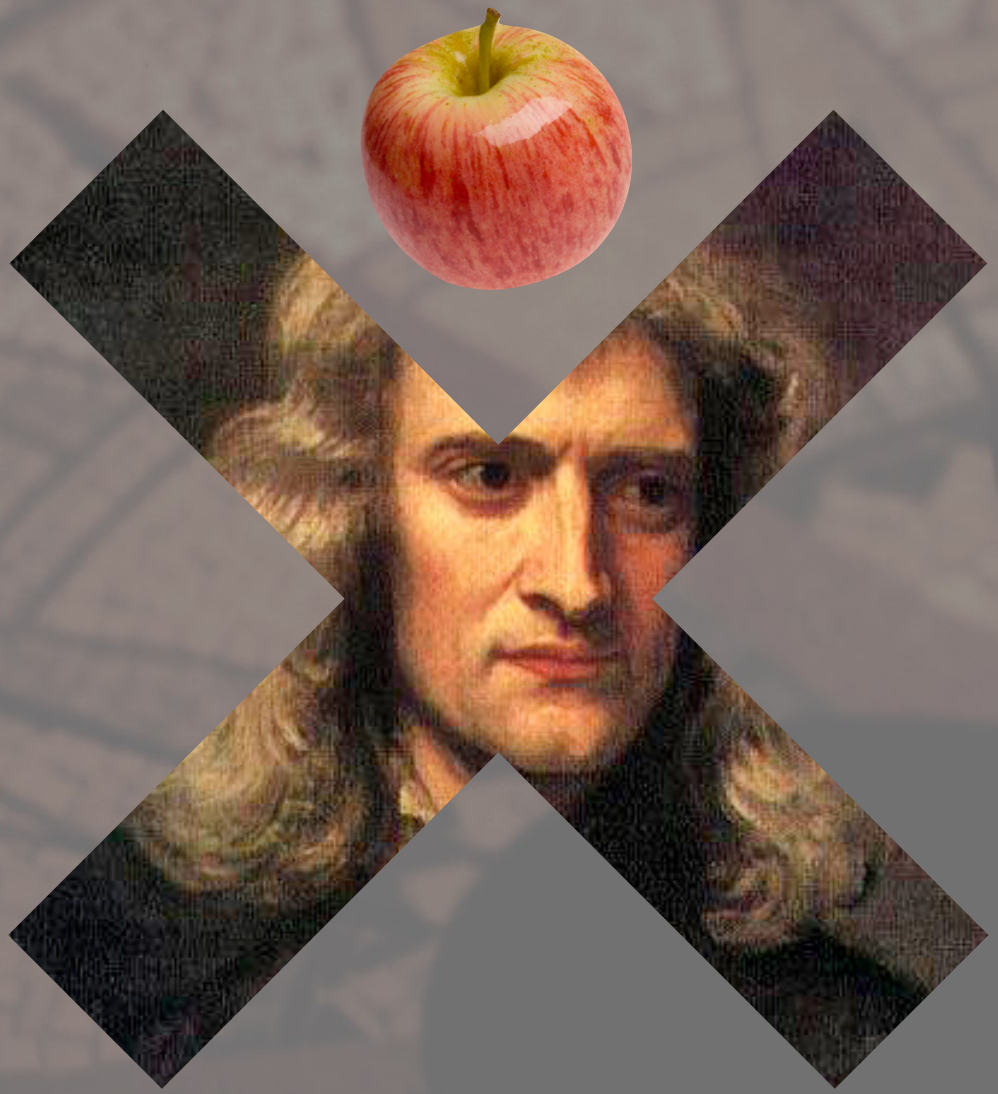
The scientific method, mathematical proofs, tests, and types.

Tests and Types

The scientific method, mathematical proofs, tests, and types.



Physics and tests



The Scientific Method

Being proven wrong and the unattainable truth

2019
HOLYJS
MOSCOW

Francis Bacon

2019
HOLYJS
MOSCOW

Inductivism

A method of reasoning

Francis Bacon

2019
HOLYJS
MOSCOW

Inductivism

A method of reasoning

Observe similar effects and similar causes and generalise.

Francis Bacon

2019
HOLYJS
MOSCOW

Inductivism

A method of reasoning

Theory

↑

Hypothesis

↑

Look for patterns

↑

Observe

Francis Bacon

2019
HOLYJS
MOSCOW

Inductivism

A method of reasoning

Observe similar effects and similar causes and generalise.

The premises are viewed as supplying some evidence for the truth of the conclusion.

Francis Bacon

2019
HOLYJS
MOSCOW

Inductivism

A method of reasoning

Observe similar effects and similar causes and generalise.

The premises are viewed as supplying some evidence for the truth of the conclusion.

No confirmations of an explanation make the explanation necessarily be true.

Francis Bacon

2019
HOLYJS
MOSCOW

Inductivism

A method of reasoning

Observe similar effects and similar causes and generalise.

The premises are viewed as supplying some evidence for the truth of the conclusion.

No confirmations of an explanation make the explanation necessarily be true.

Francis Bacon

2019
HOLYJS
MOSCOW

Inductivism

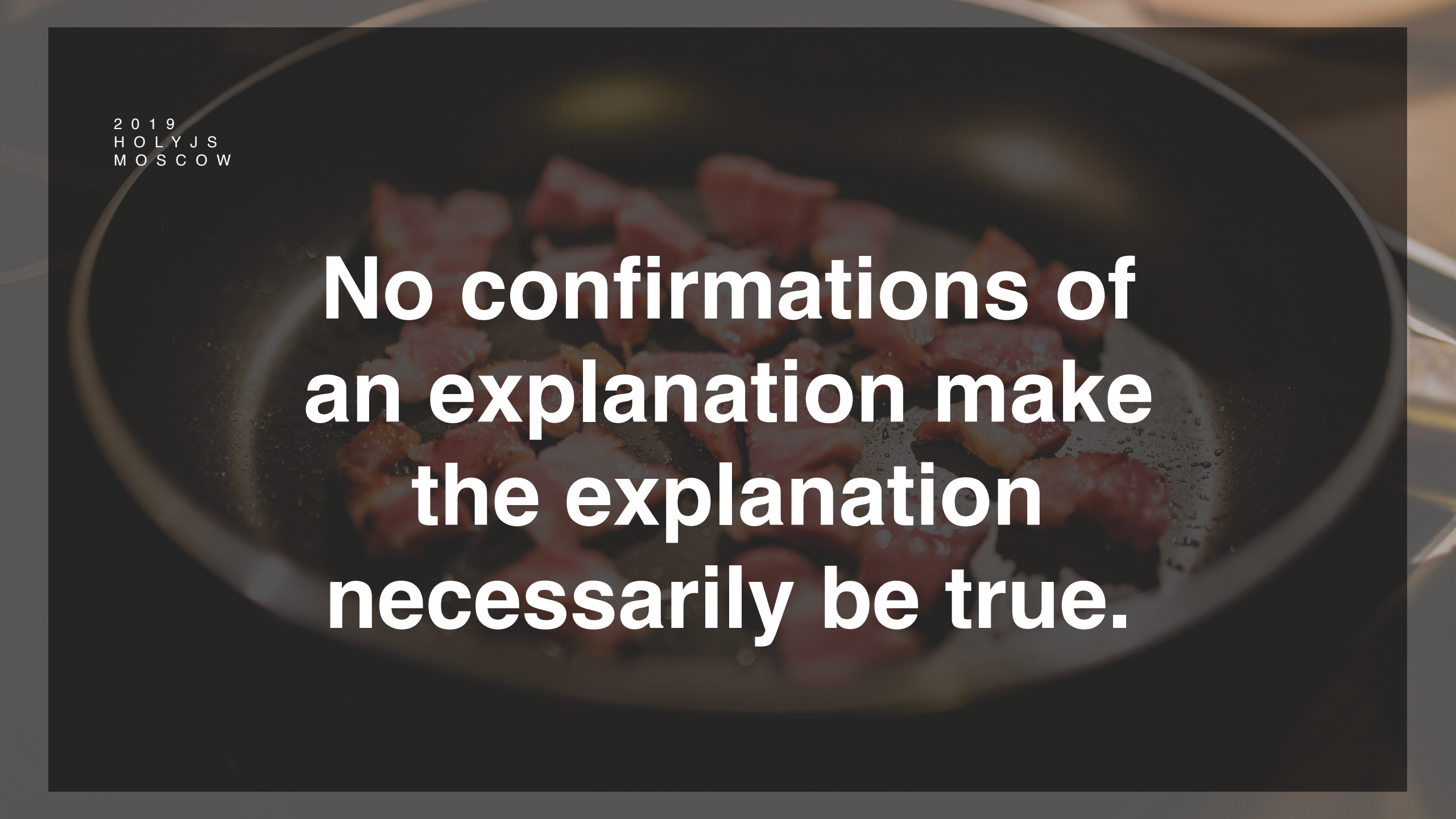
A method of reasoning

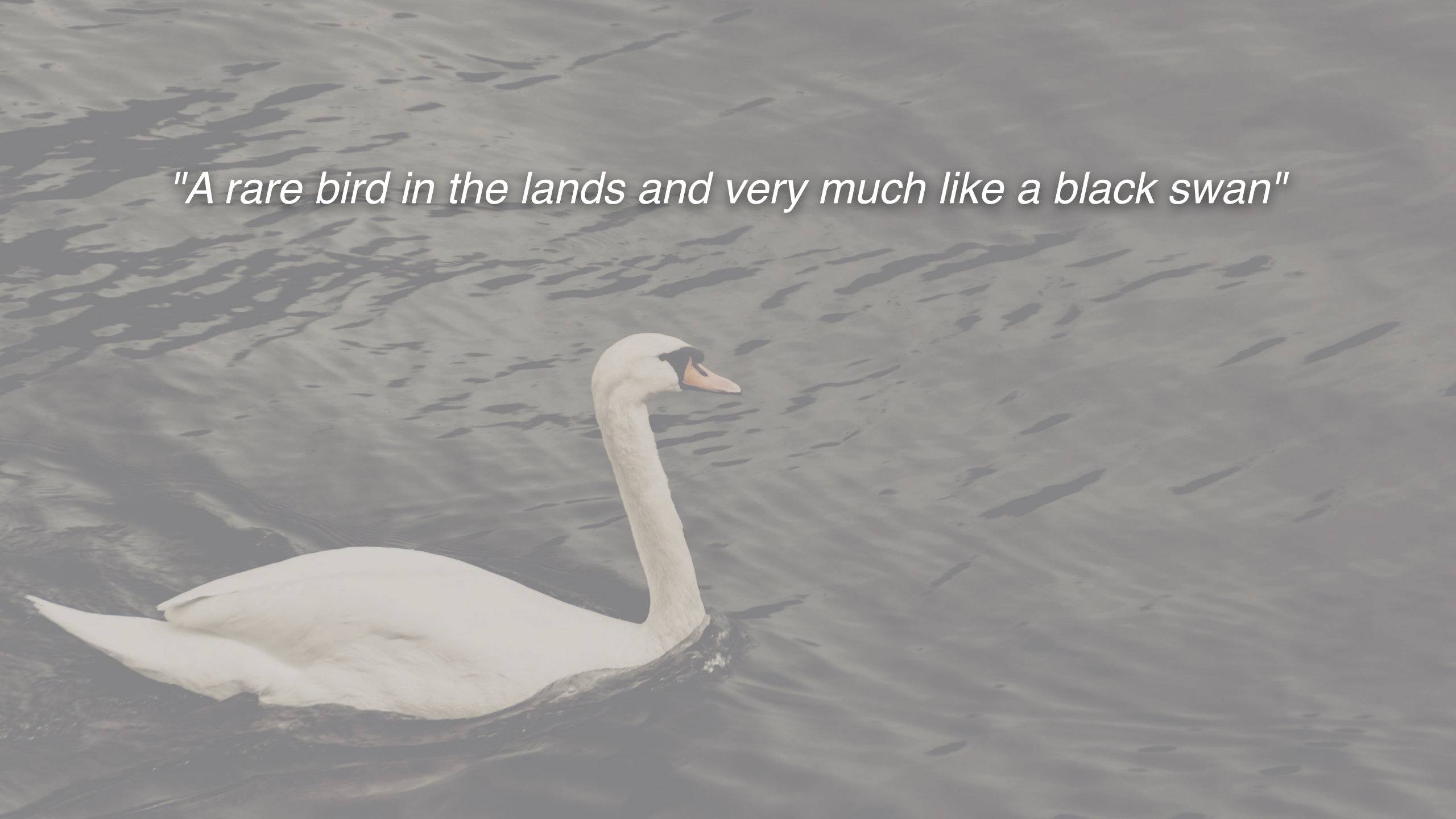
Observe similar effects and similar causes and generalise.

The premises are viewed as supplying some evidence for the truth of the conclusion.

Francis Bacon

No confirmations of an explanation make the explanation necessarily be true.


2019
HOLYJS
MOSCOW
No confirmations of an explanation make the explanation necessarily be true.


"A rare bird in the lands and very much like a black swan"

Systems of thought are way more fragile than we think.

Systems of thought are way more fragile than we think.

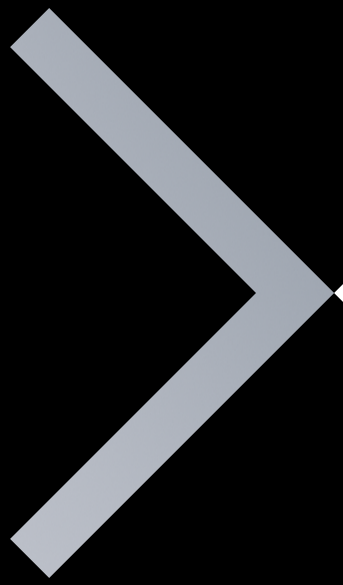
A million successful experiments cannot prove a theory correct, but one failed experiment can prove a theory wrong.
– POPPER, Karl


Truth is unattainable

Science is a successive rejection of falsified theories

We are constantly replacing theories by others which have greater explanatory power.

More evidence gets us closer to the truth



More tests gets us closer to correct software

More tests gets us closer to correct software

We believe what's most probable, not what's necessarily true.

More tests gets us closer to correct software

We believe what's most probable, not what's necessarily true.

Blindly trusting science is dangerous.
But so is not trusting in it at all.

More tests gets us closer to correct software

We believe what's most probable, not what's necessarily true.

Blindly trusting tests is dangerous.
But so is not trusting in it at all.

More tests gets us closer to correct software

We believe what's most probable, not what's necessarily true.

Blindly trusting tests is dangerous.
But so is not trusting in it at all.

Science is not dogmatic.

More tests gets us closer to correct software

We believe what's most probable, not what's necessarily true.

Blindly trusting tests is dangerous.
But so is not trusting in it at all.

Tests are not permanent.

How we do Science

How do we know what we know?

THE SCIENTIFIC METHOD 

How we do Science

How do we know what we know?

THE SCIENTIFIC METHOD

Form a conjecture, state an explanation

How we do Science

How do we know what we know?

THE SCIENTIFIC METHOD

- Form a conjecture, state an explanation
- Deduce predictions from the hypothesis

How we do Science

How do we know what we know?

THE SCIENTIFIC METHOD

- Form a conjecture, state an explanation
- Deduce predictions from the hypothesis
- Test, make experiments

How we do Science

How do we know what we know?

THE SCIENTIFIC METHOD

- Form a conjecture, state an explanation
- Deduce predictions from the hypothesis
- Test, make experiments
- Observe whether your theory matches reality

How we write tests

How do we know what we know?

THE SCIENTIFIC METHOD

Form a conjecture, state an explanation

Deduce predictions from the hypothesis

Test, make experiments

Observe whether your theory matches reality

Be willing
to be wrong

It's easy to find confirmation
for a theory if you are looking for it.

Be willing to be wrong

It's easy to find confirmation for a theory if you are looking for it.

Your assertions can have a true or false value, which you don't know in advance

Be willing to be wrong

It's easy to find confirmation
for a theory if you are looking for it.

Your assertions can have a true or false value, which you don't know in advance

Your hypothesis must be **falsifiable**

Be willing to be wrong

It's easy to find confirmation
for a theory if you are looking for it.

- Your assertions can have a true or false value, which you don't know in advance
- Your hypothesis must be **falsifiable**
- Your observations should **not** strive for confirmation, but for disconfirmation

Be willing to be wrong

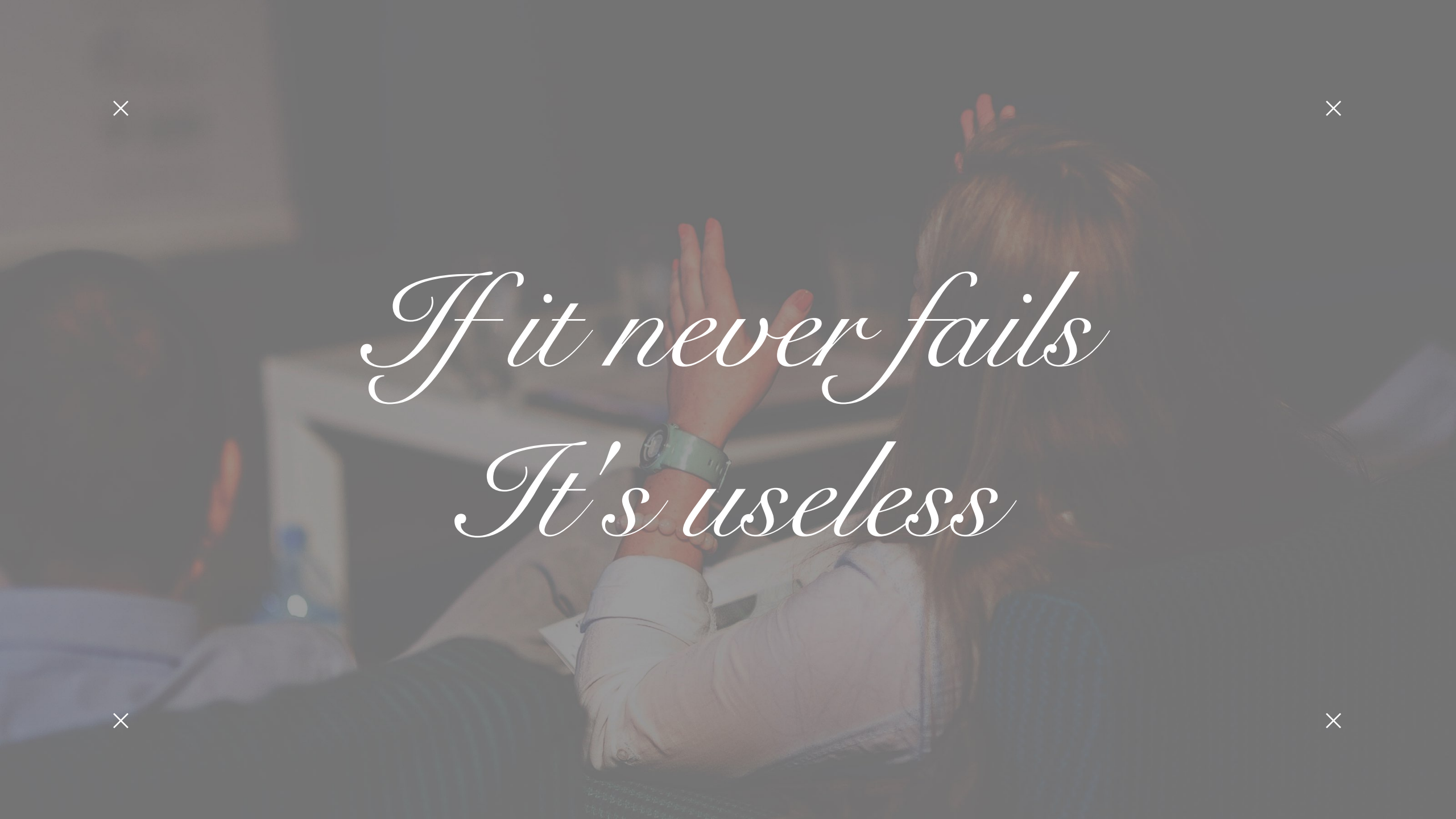
It's easy to find confirmation
for a theory if you are looking for it.

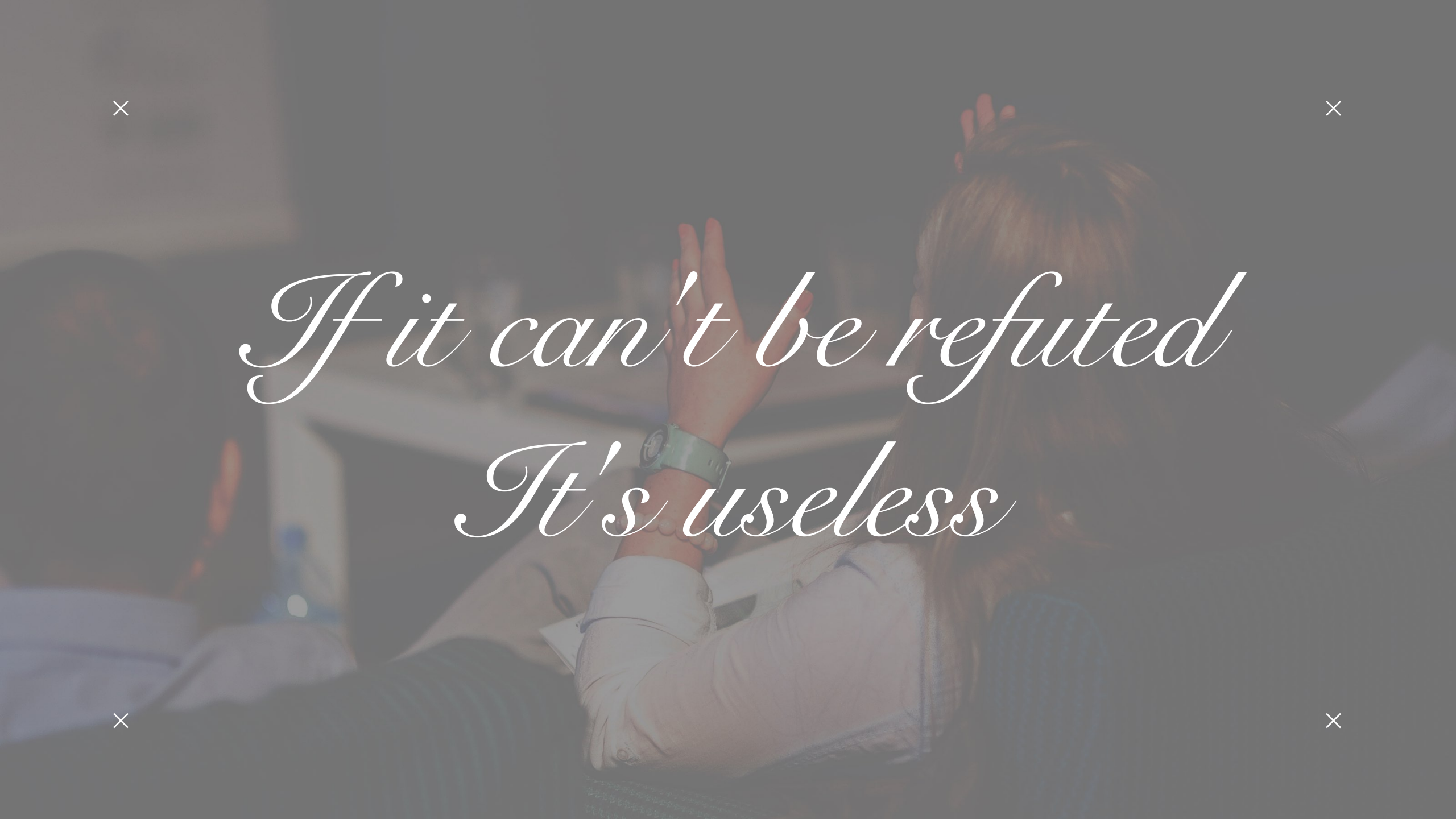
Your assertions can have a true or false value, which you don't know in advance

Your hypothesis must be **falsifiable**

Your observations should **not** strive for confirmation, but for disconfirmation

Your tests must be **risky**, they need to be able to falsify your theory.


If it never fails
It's useless

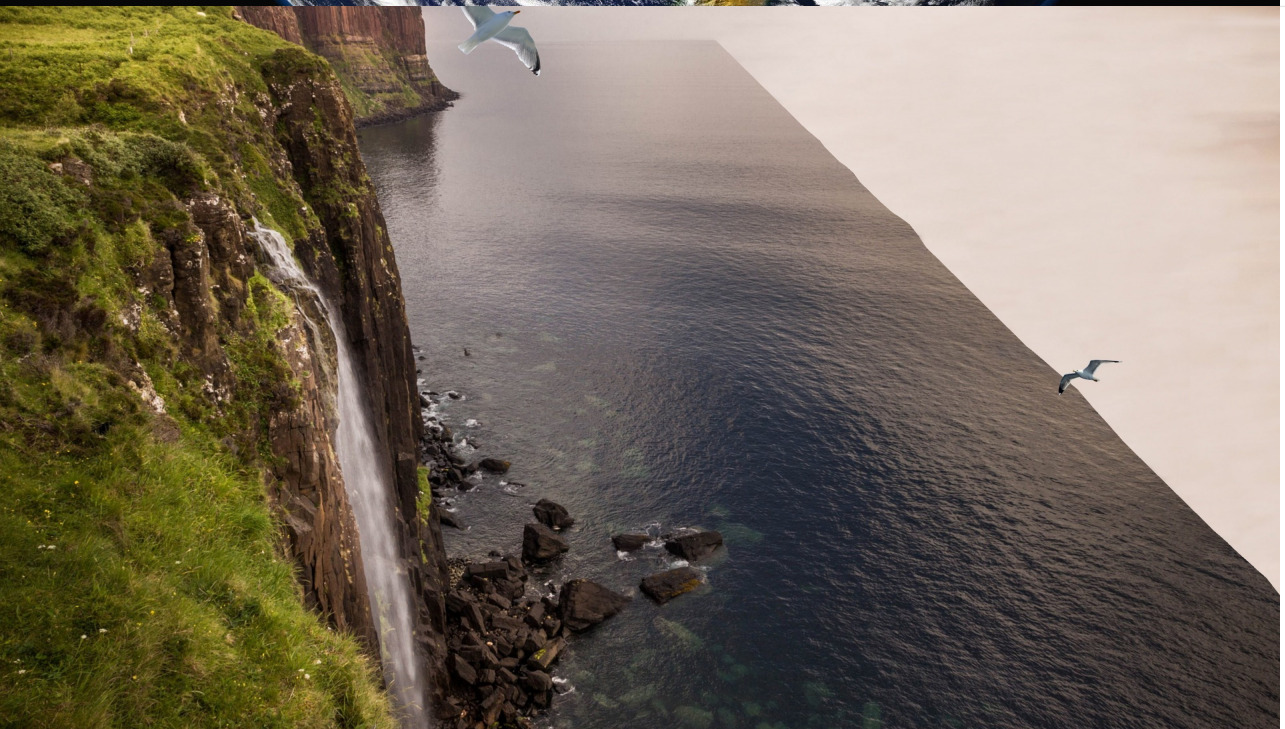

If it can't be refuted
It's useless

×

SCIENCE DISPROVES

It's not possible to prove a theory correct as we can't test all possible scenarios taking into account all possible variables.

Tracing appearances, not unveiling reality.



×

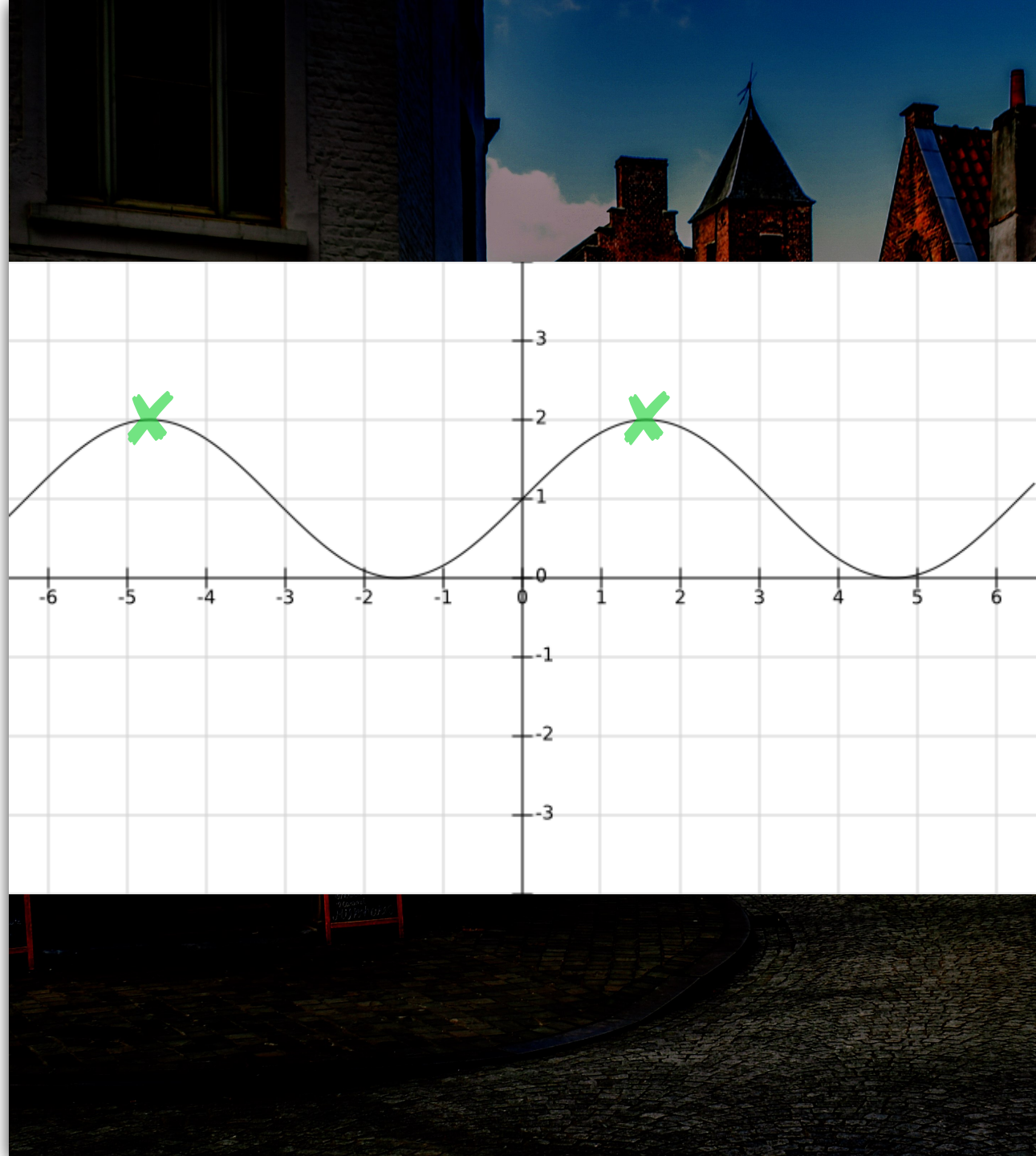
PSEUDOSCIENCE PROVES

Pseudoscience does not look for arguments contrary to its affirmations.

Irrefutable theories are not scientific.

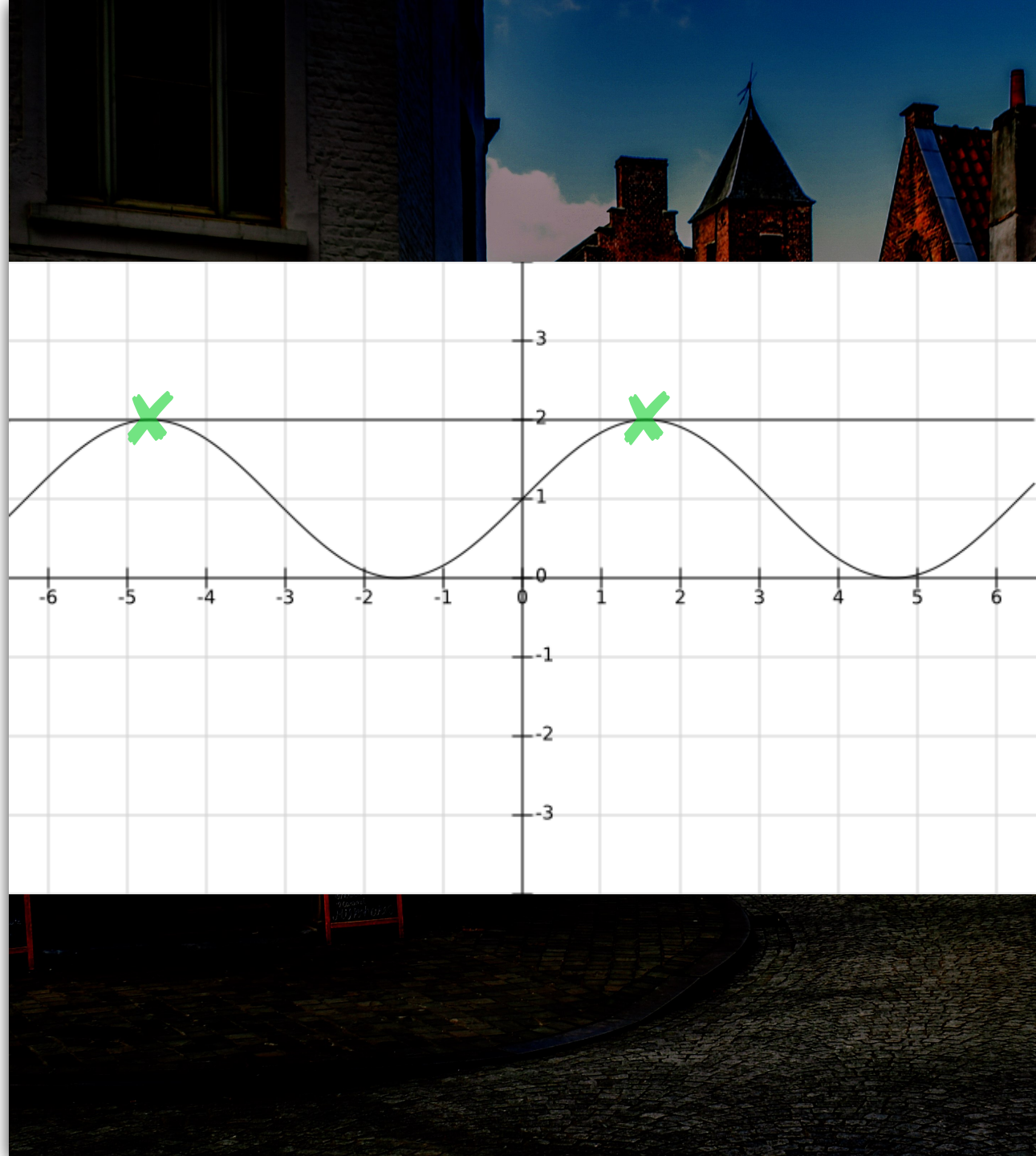
In the same way that physicists cannot prove they are right with experiments, tests can't prove that assumptions about our code are right

A function with two tests



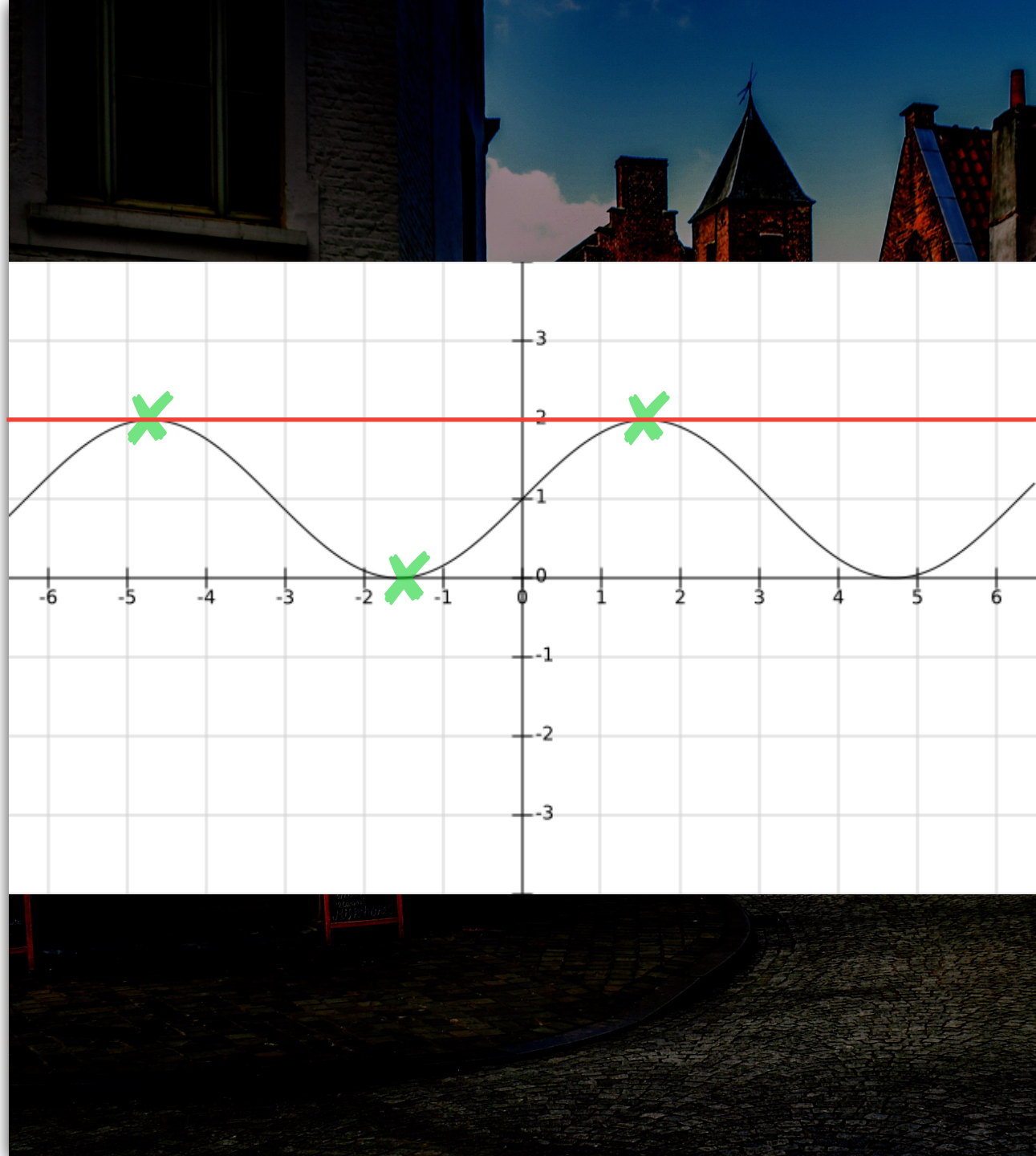
Both functions match our predictions and pass our tests.

Tests do not guarantee correctness.



With more tests we can rule out other intersecting implementations

More tests, more evidence



Tests can't
prove that our code
is correct.

Tests can't
prove that our code
is correct.

Tests can only
prove that it isn't

Tests can't
prove that our code
is correct.

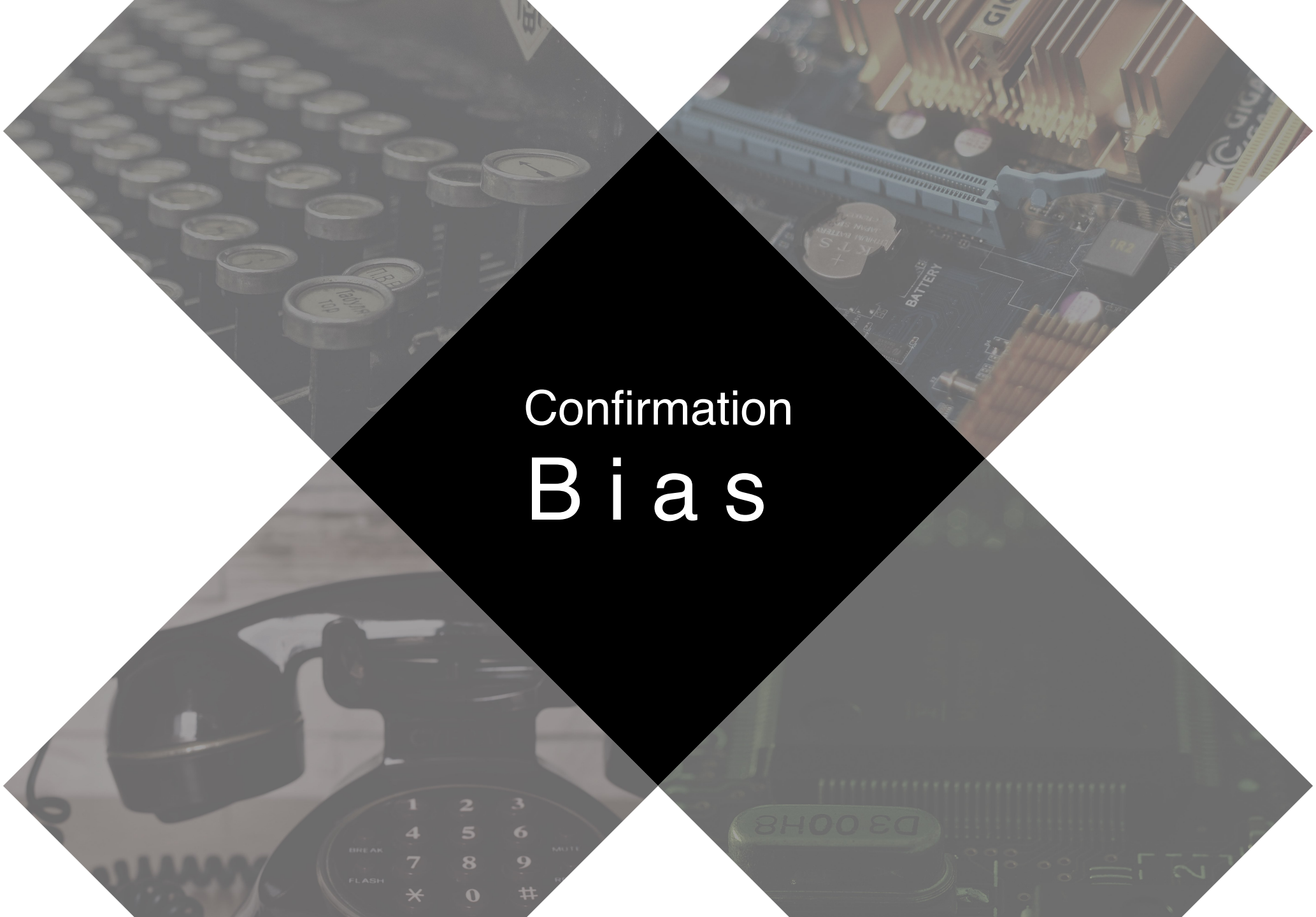
Tests provide
evidence

Tests can only
prove that it isn't

Experiments can't prove that our code is correct.

Experiments provide evidence

Experiments can only prove that it isn't


Confirmation
B i a s

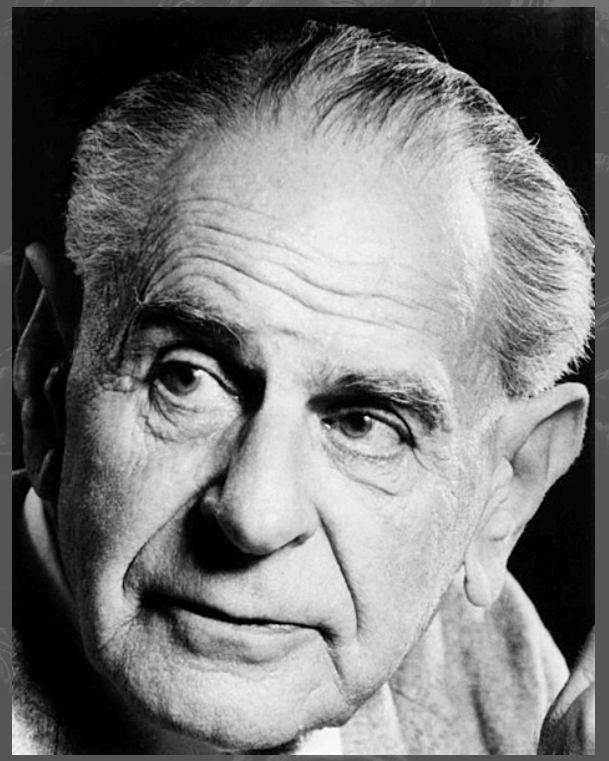
Experiments depend on observation

We observe reality and try to find evidence which contradicts our findings

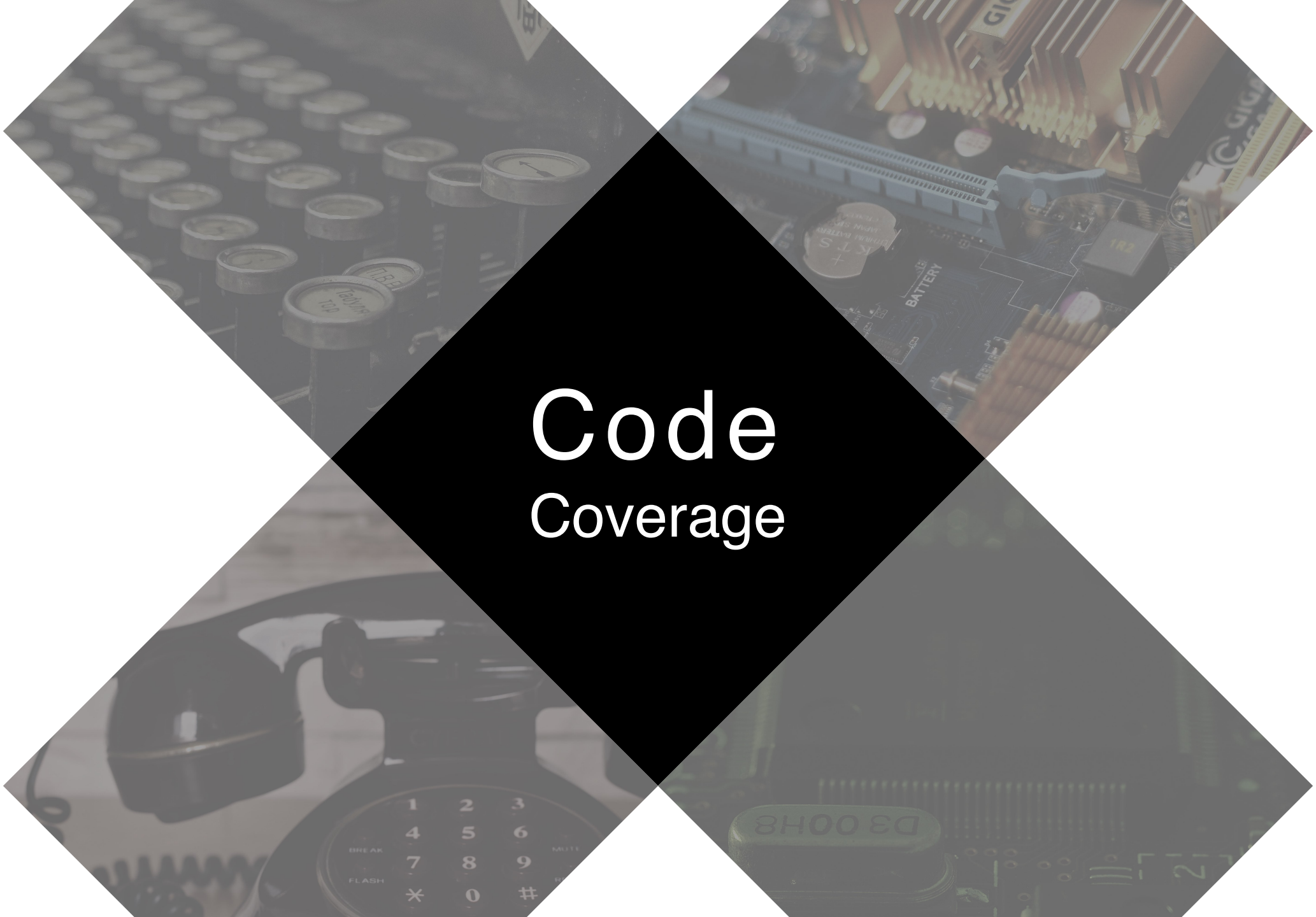
Tests depend on assertions

A test which does not contain assertions simply verifies whether the code can be executed.

Observation is always selective. It needs a chosen object, a definite task, an interest, a point of view, a problem. [...] It presupposes similarity and classification, which in their turn presuppose interests, points of view, and problems.



Karl R. Popper
Conjectures and Refutations: The Growth of Scientific Knowledge



Code
Coverage

James O. Coplien

jcoplien

I define 100% coverage as having examined all possible combinations of all possible paths through all methods of a class, having reproduced every possible configuration of data bits accessible to those methods, at every machine language instruction along the paths of execution.

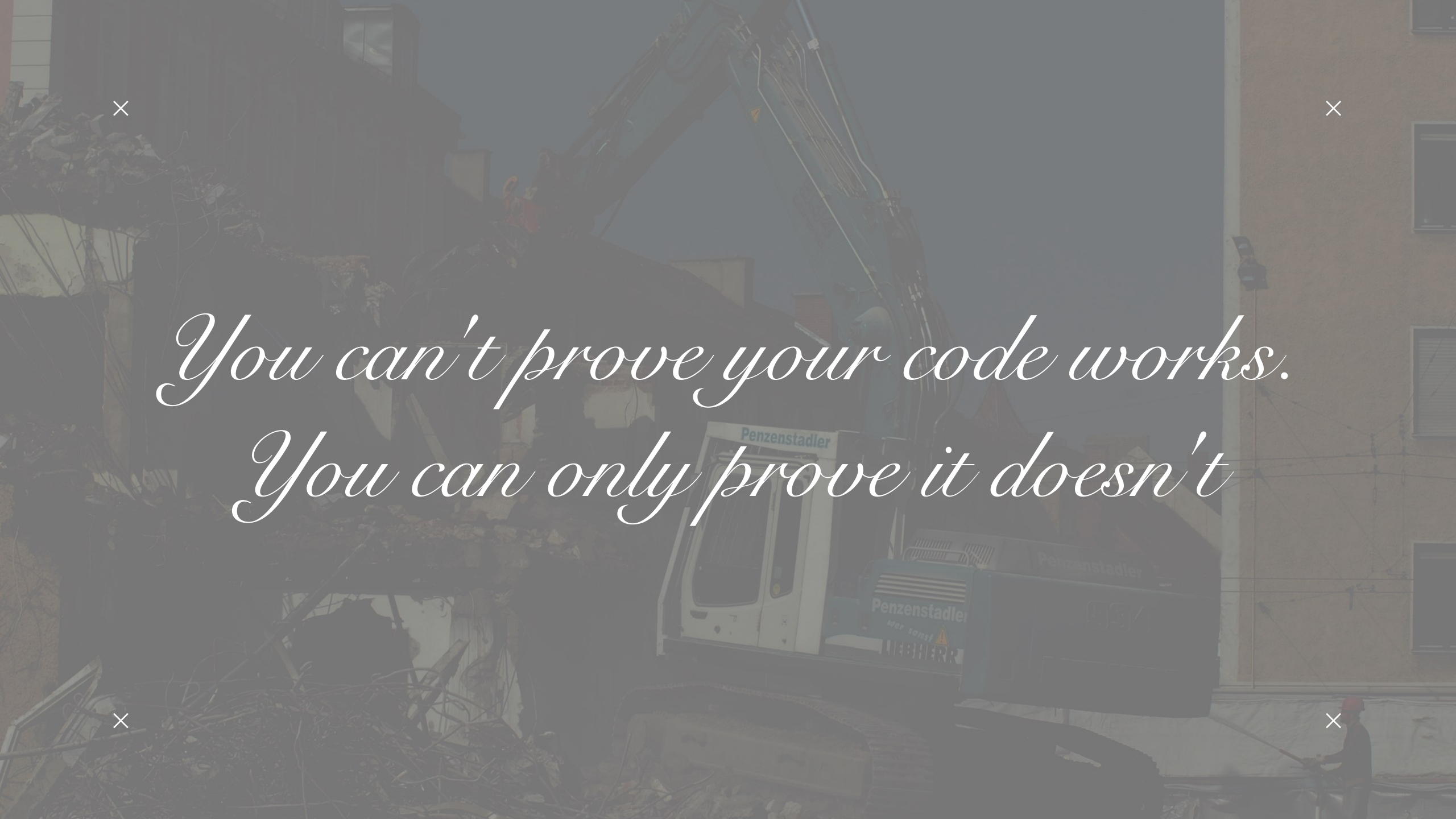
James O. Coplien
jcoplien

James O'Coplien
Why most unit testing is waste

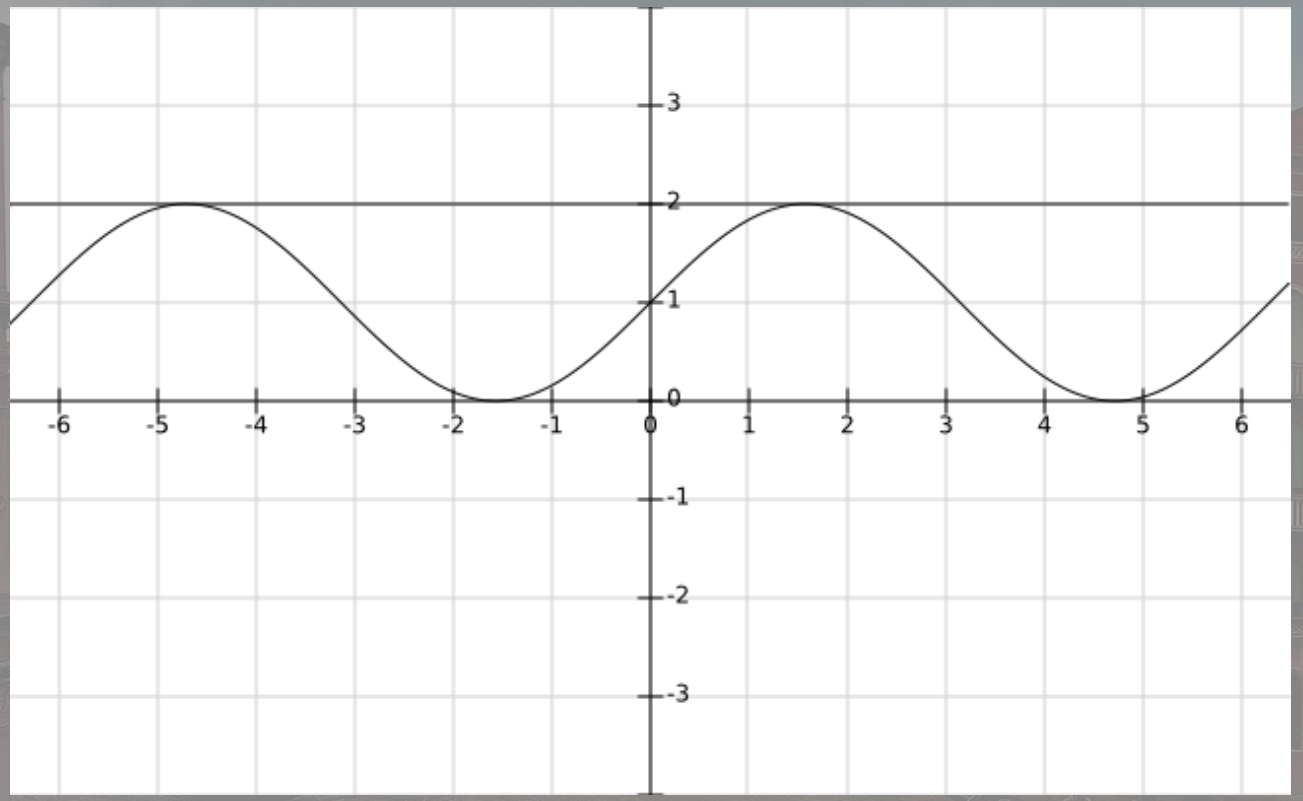
I define 100% coverage as having examined all possible combinations of all possible paths through all methods of a class, having reproduced every possible configuration of data bits accessible to those methods, at every machine language instruction along the paths of execution. ***Anything else is a heuristic about which absolutely no formal claim of correctness can be made.***

James O'Coplien
Why most unit testing is waste

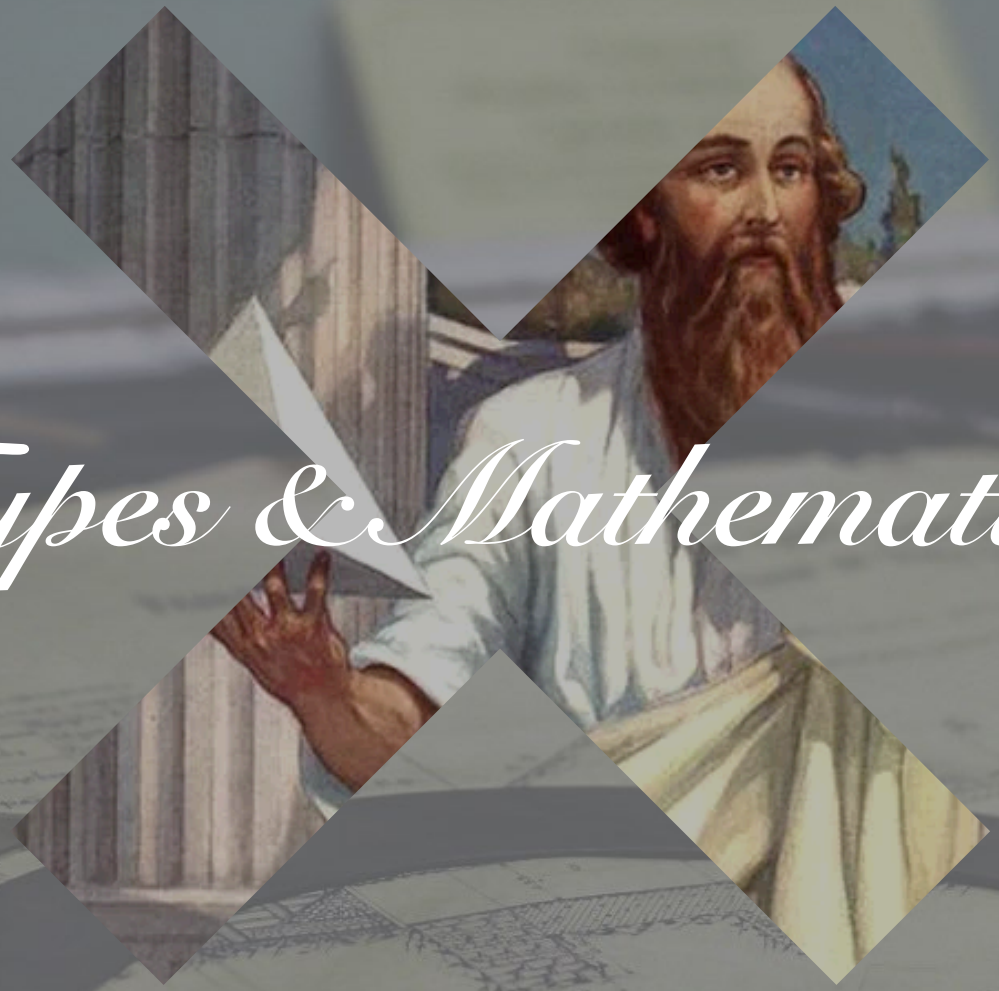
James O. Coplien
jcoplien



You can't prove your code works.
You can only prove it doesn't



Go through every single point...



Types & Mathematics

Mathematical truth

CONJECTURE

A statement which does not have a proof, but is believed to be true

Mathematical truth

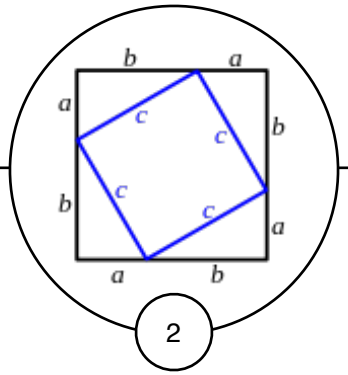
CONJECTURE

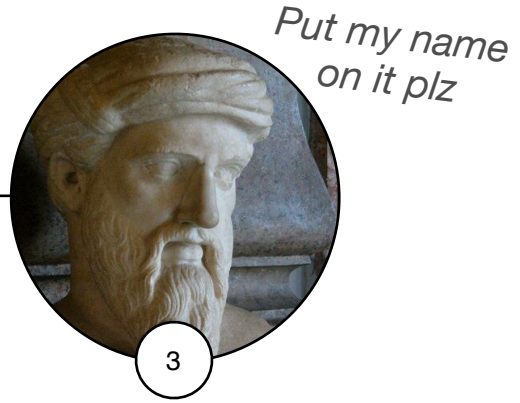
A statement which does not have a proof, but is believed to be true

PROOF

A series of steps in reasoning for demonstrating a mathematical statement is true.

Mathematical truth





CONJECTURE

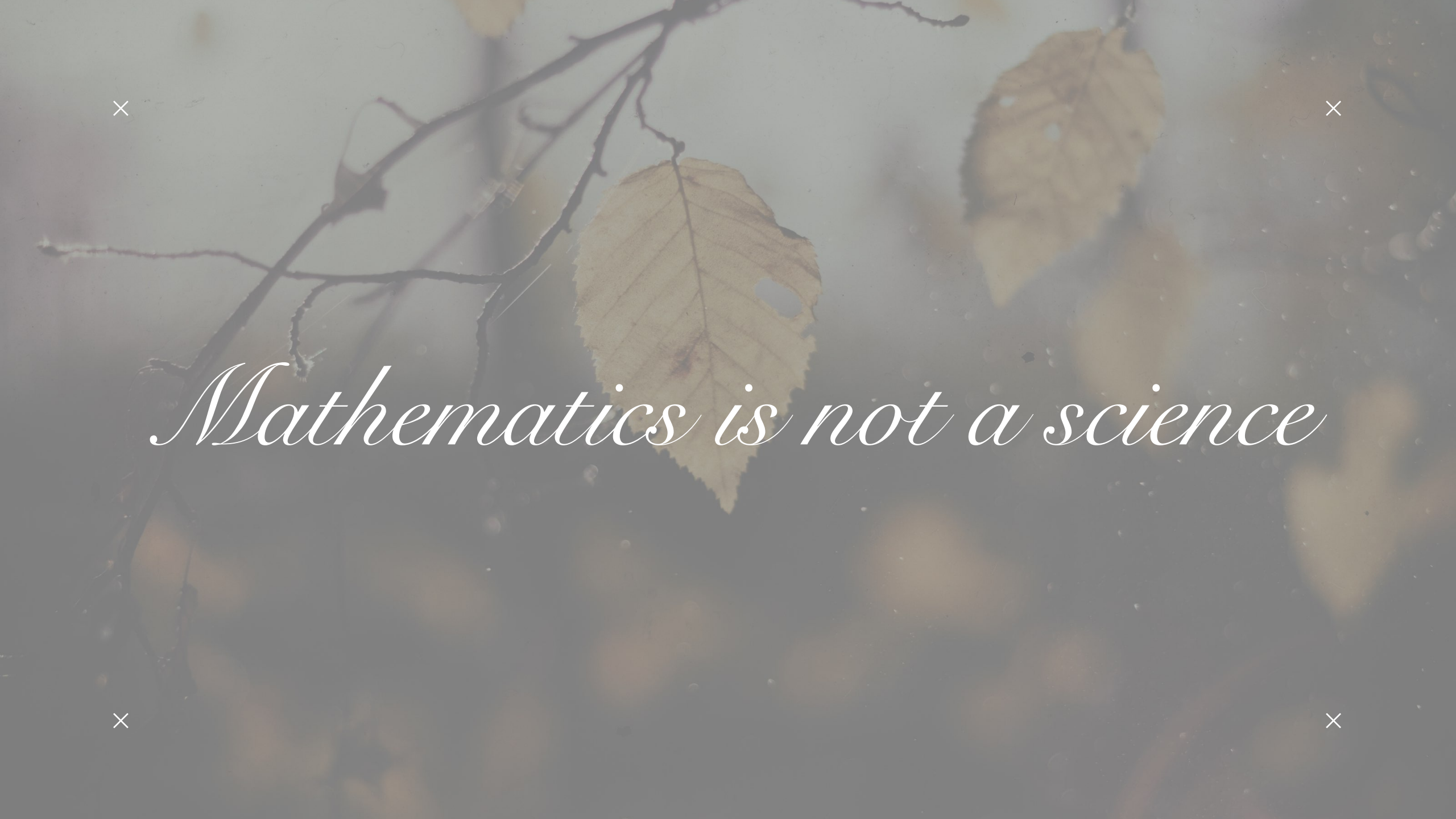
A statement which does not have a proof, but is believed to be true

PROOF

A series of steps in reasoning for demonstrating a mathematical statement is true.

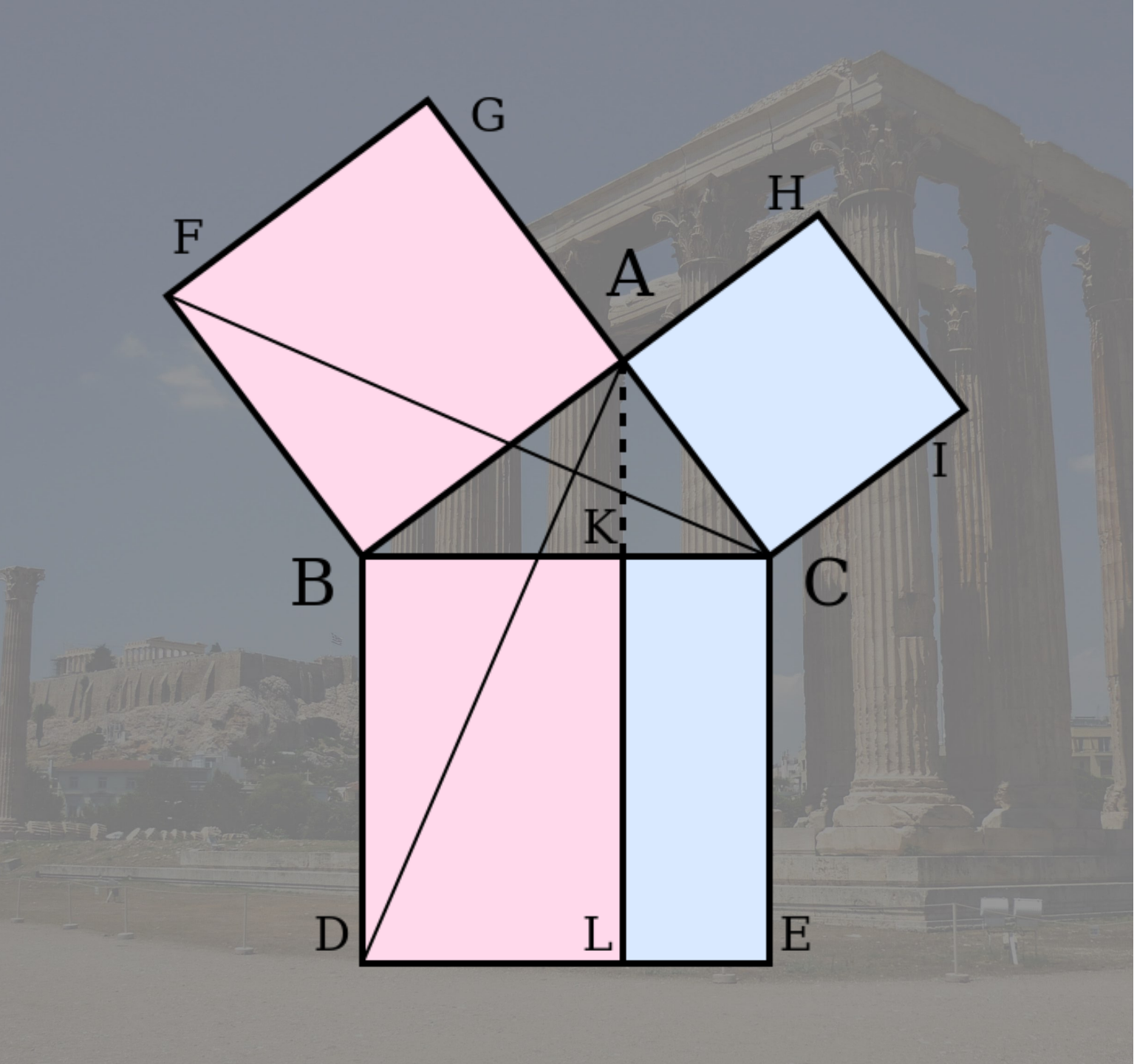
THEOREM

A mathematical statement that is proved using rigorous mathematical reasoning


Mathematics is not a science

Mathematics is not a science from our point of view, in the sense that it is not a natural science. The test of its validity is not experiment.

Richard P. Feyman
The Feynman Lectures on Physics Vol. 1

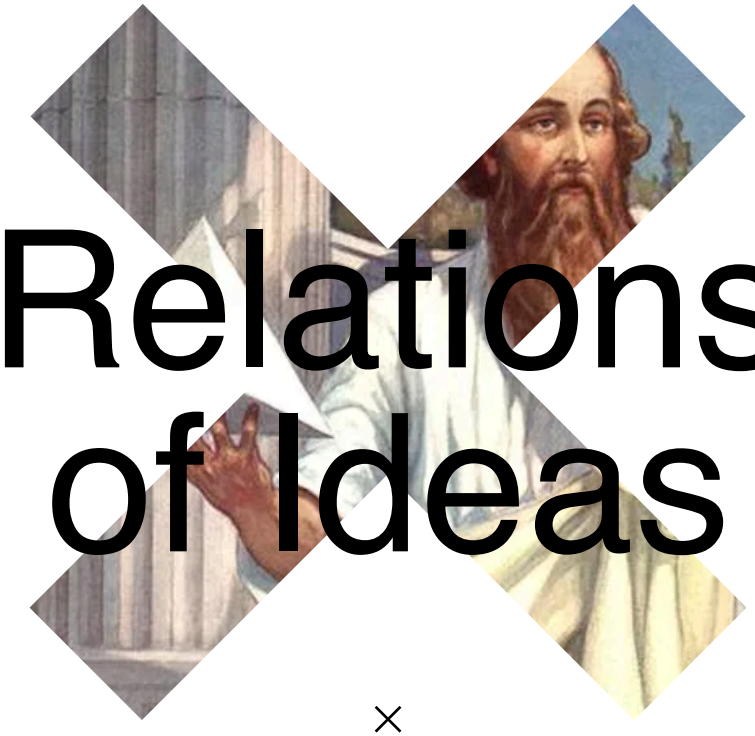


The Quest for Truth

Types and Mathematics

If it follows the rules, it's correct.

DAVID HUME'S

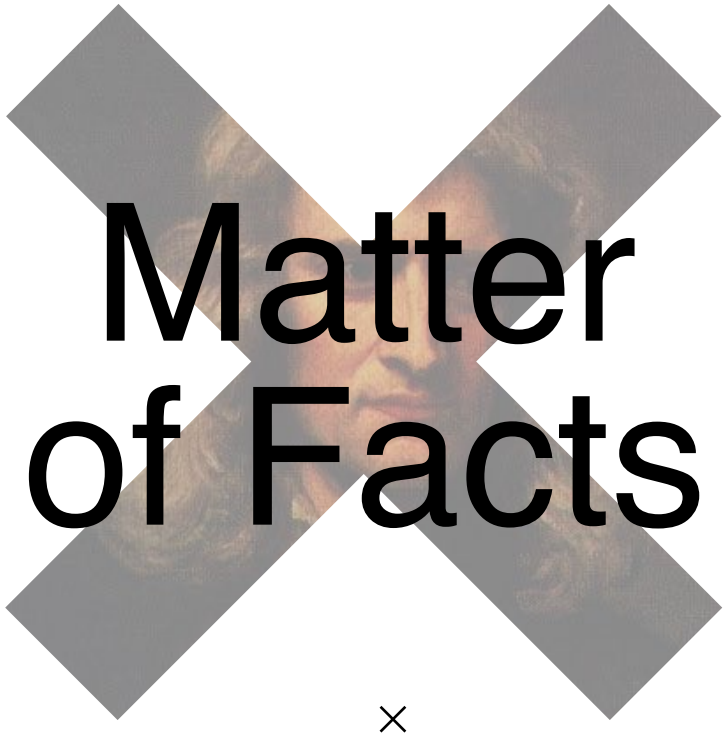


Relations of Ideas

×

MATHEMATICS

A PRIORI

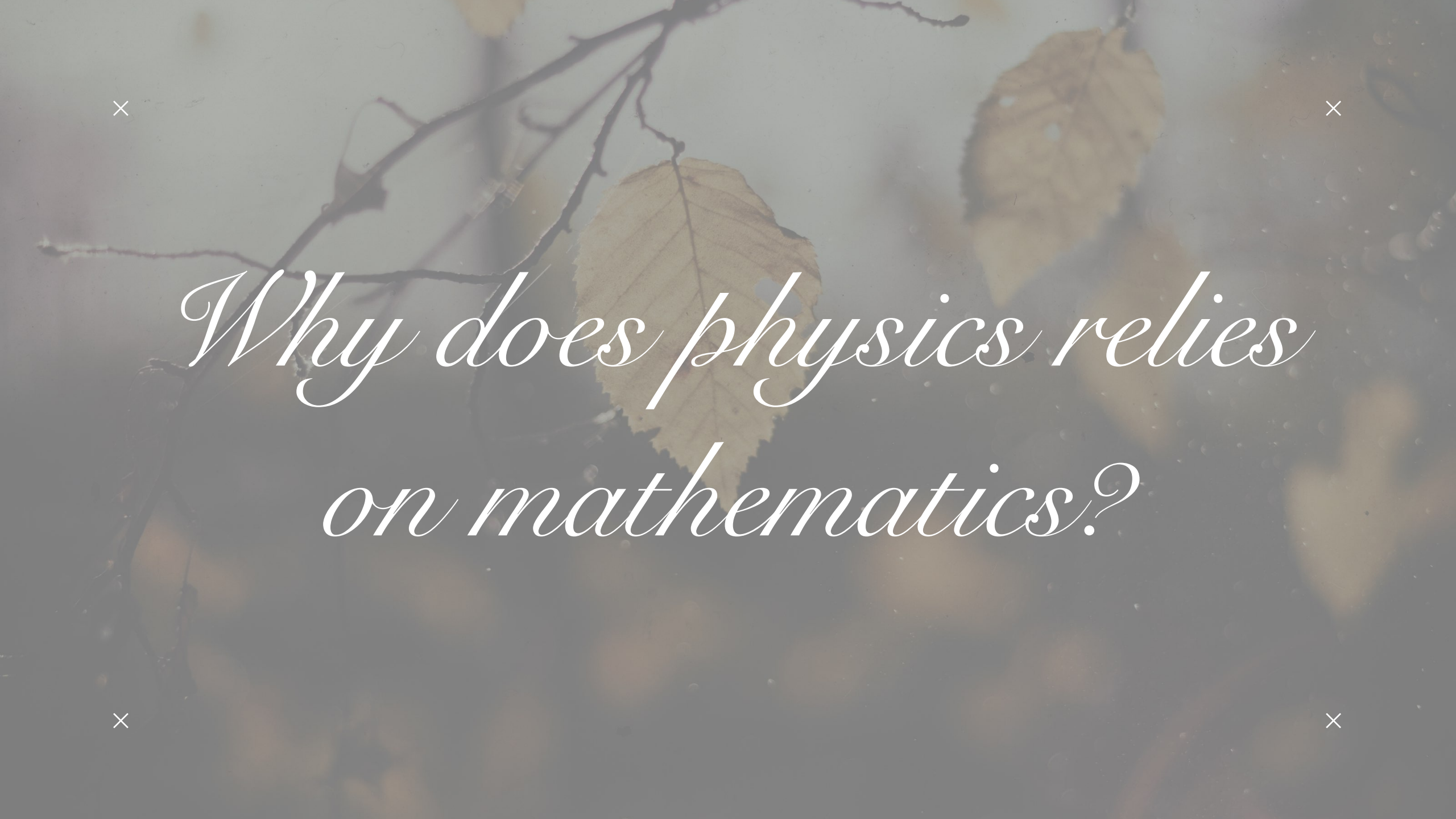


Matter of Facts

×

PHYSICS

A POSTERIORI


Why does physics relies
on mathematics?

Abstraction

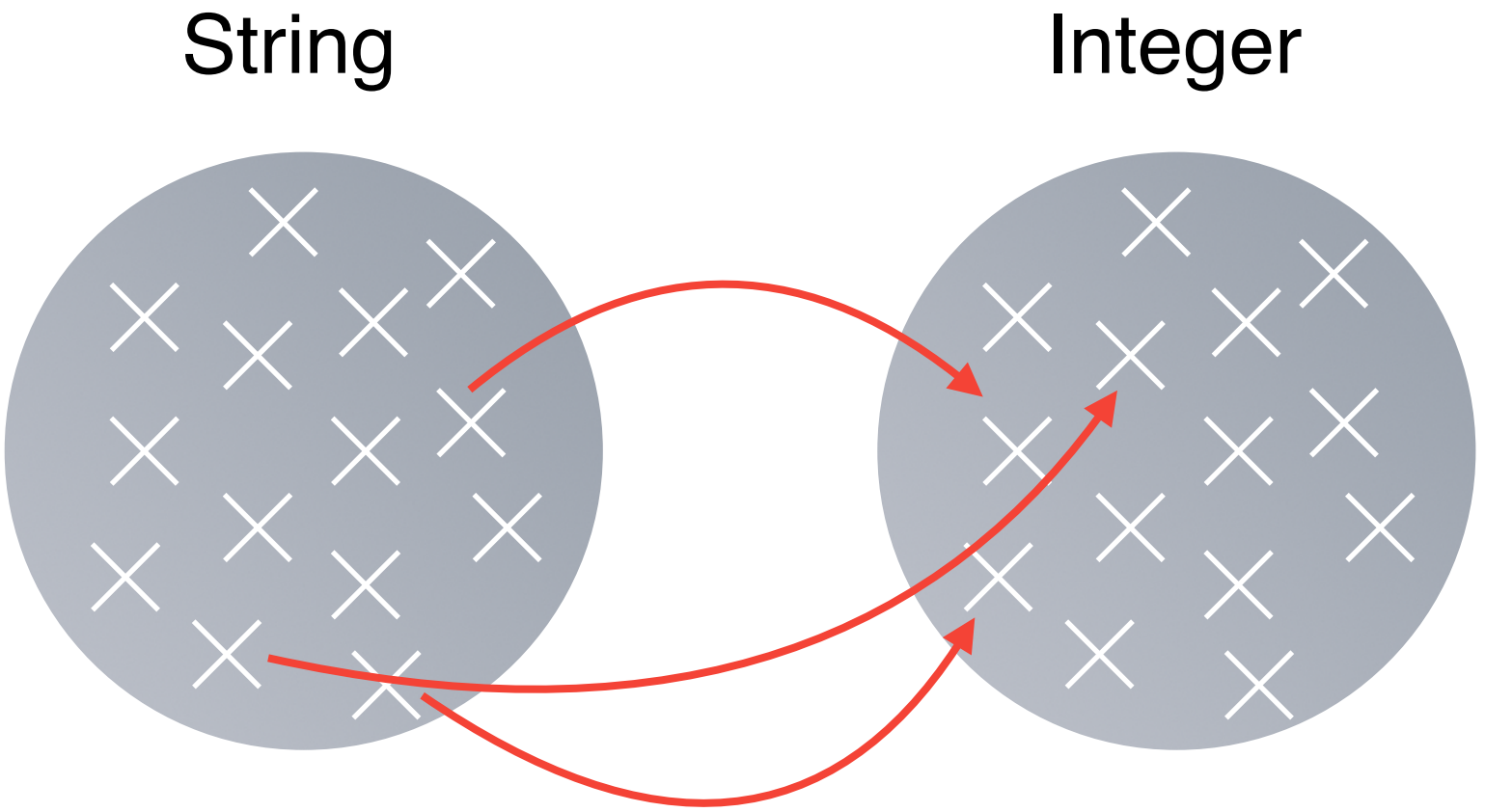
The ability of concentrating in the essential aspects of a certain context.

Remove all unnecessary detail.

Abstraction

The ability of concentrating in the essential aspects of a certain context.

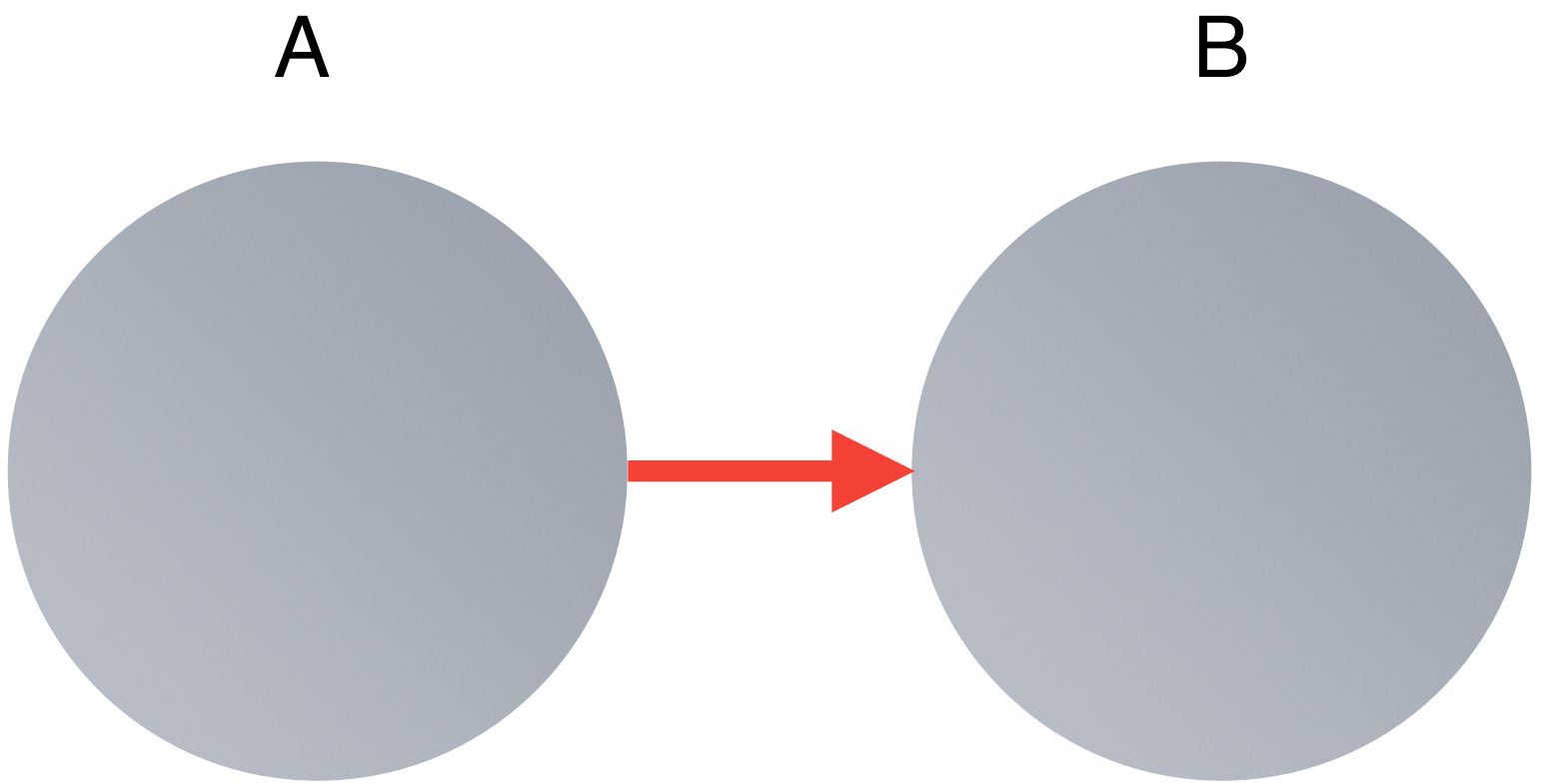
Remove all unnecessary detail.



Abstraction

The ability of concentrating in the essential aspects of a certain context.

Remove all unnecessary detail.



Abstraction

The ability of concentrating in the essential aspects of a certain context.

Remove all unnecessary detail.

Abstraction

The ability of concentrating in the essential aspects of a certain context.

Remove all unnecessary detail.

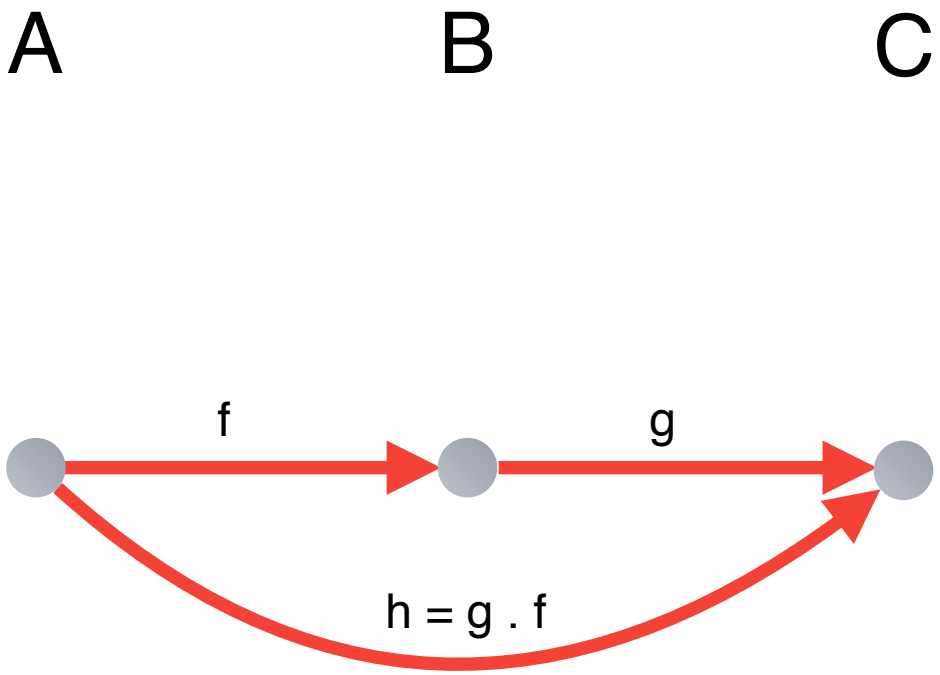
The end of road for abstraction.

Bartosz Milewski

Making claims and proving properties

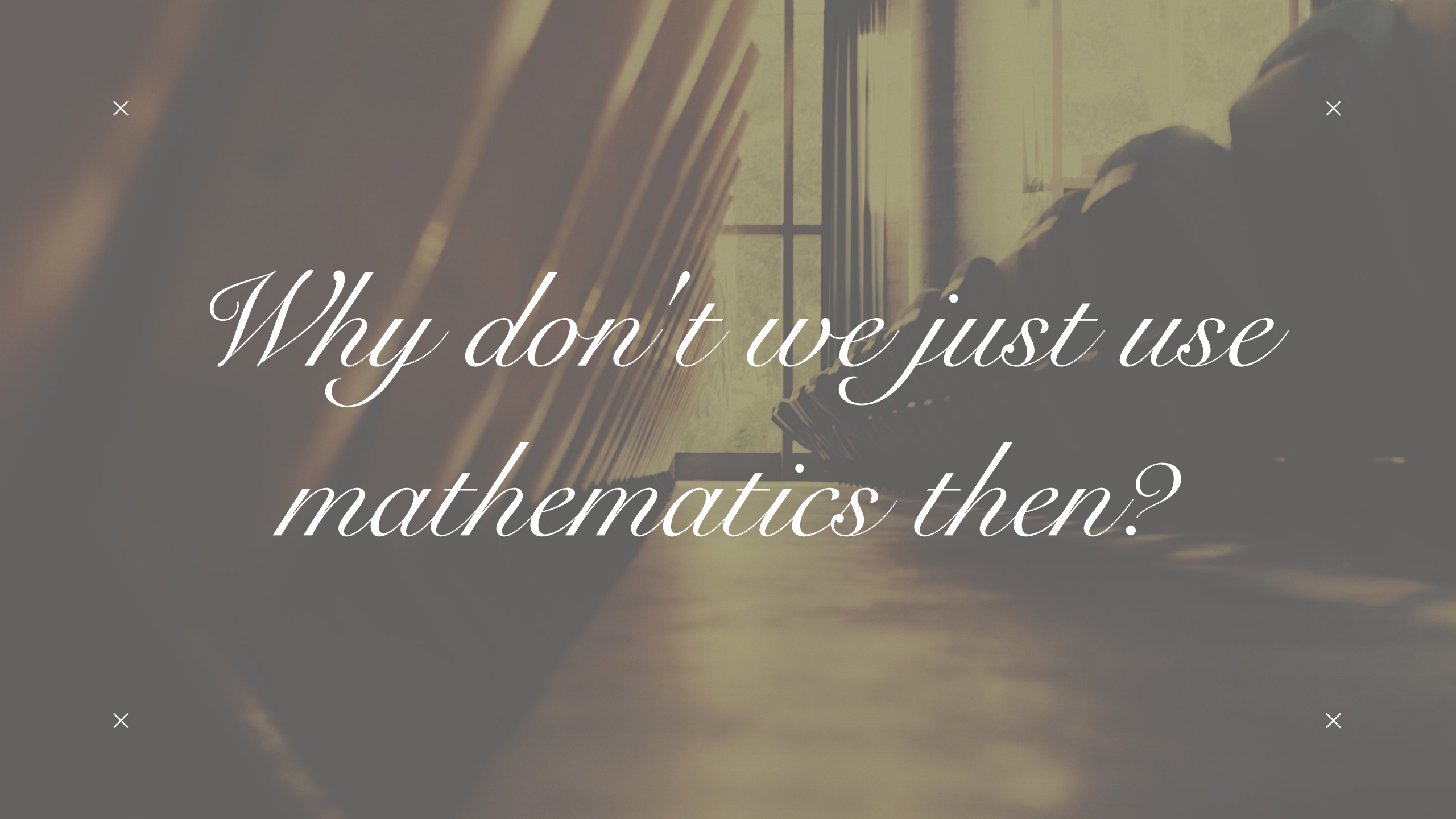
When we have a specific set of rules we can use to manipulate symbols we can reach truth.

This is the beauty of abstraction.



If controversies were to arise, there would be no more need of disputation between two philosophers than between two calculators. For it would suffice for them to take their pencils in their hands and to sit down at the abacus, and say to each other: "calculemus"

Gottfried Wilhelm von Leibniz


Why don't we just use
mathematics then?

Kurt Gödel

If a logical system is consistent, it cannot be complete. **There will be true statements which can't be proven.**

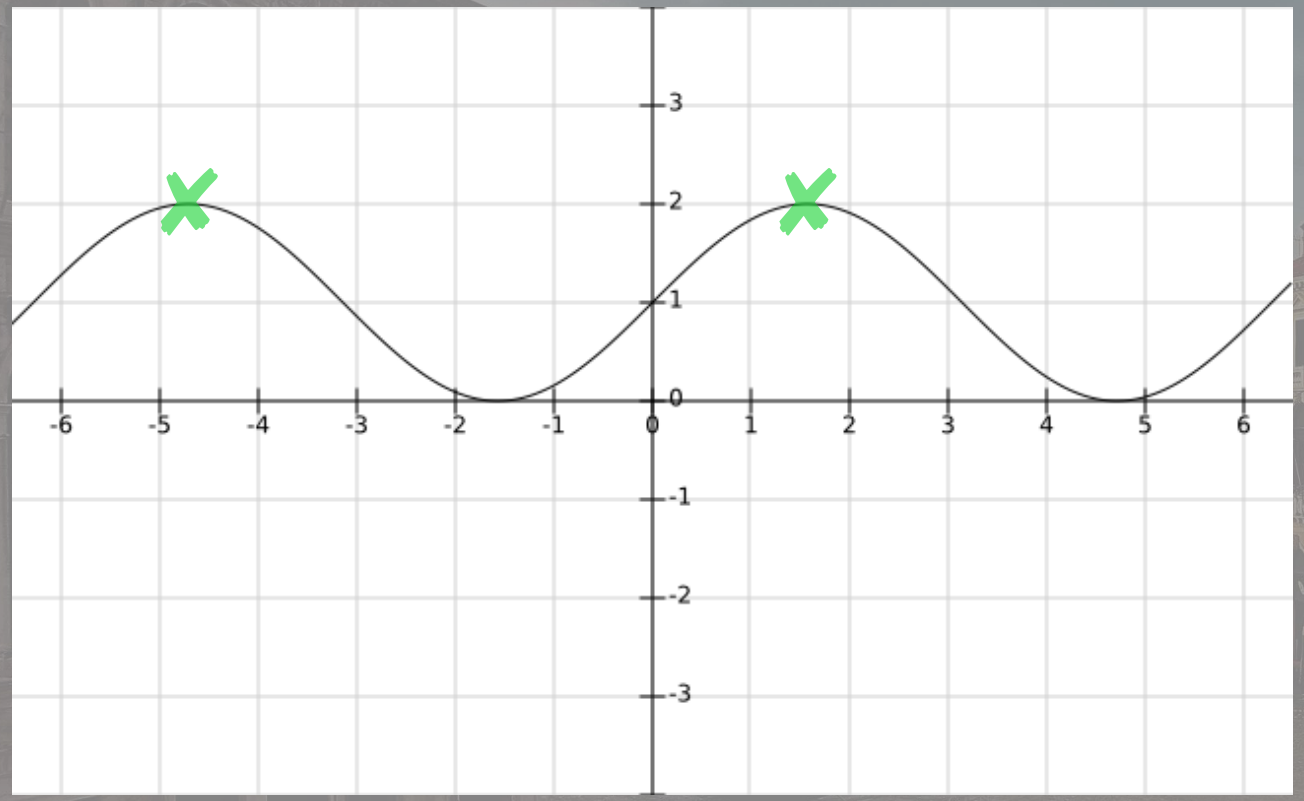
Kurt Gödel

If a logical system is consistent, it cannot be complete. **There will be true statements which can't be proven.**

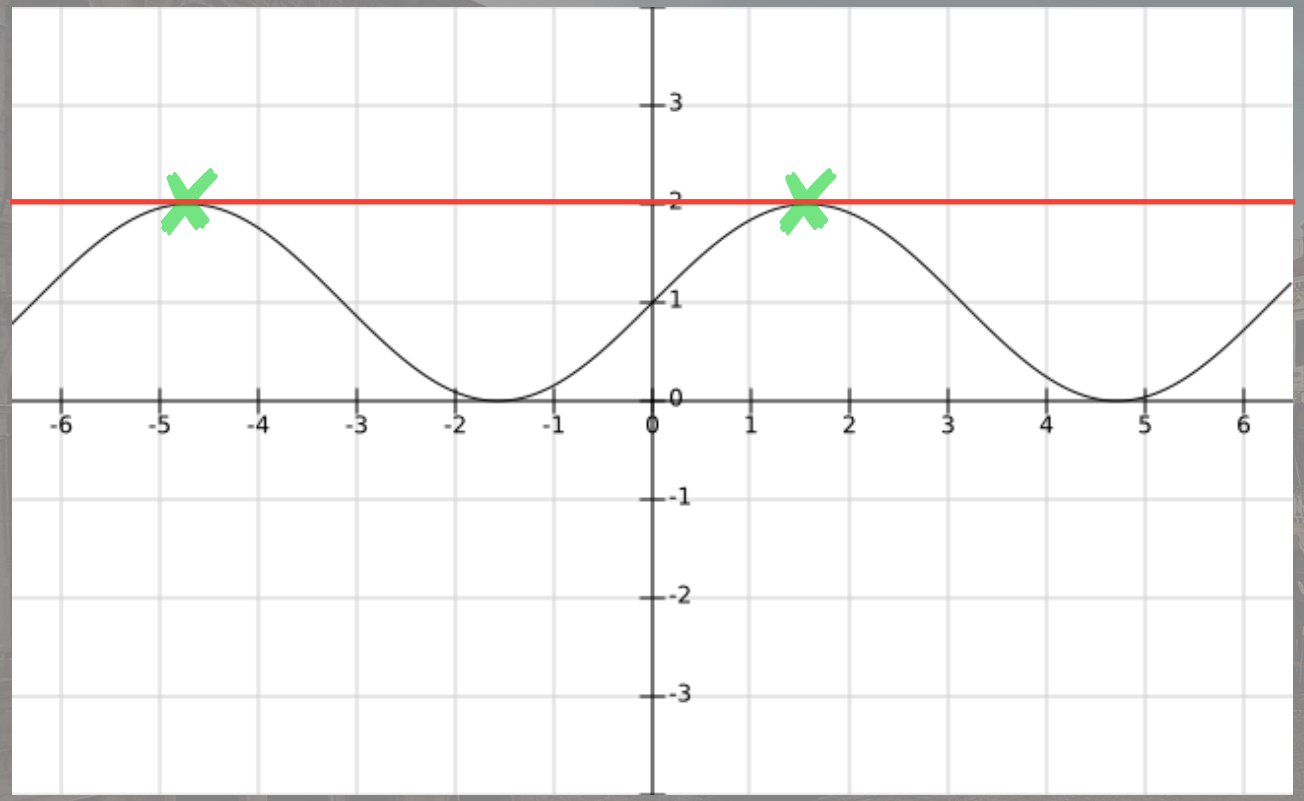
There is no way to show that any useful formal system is free of false statements

Kurt Gödel

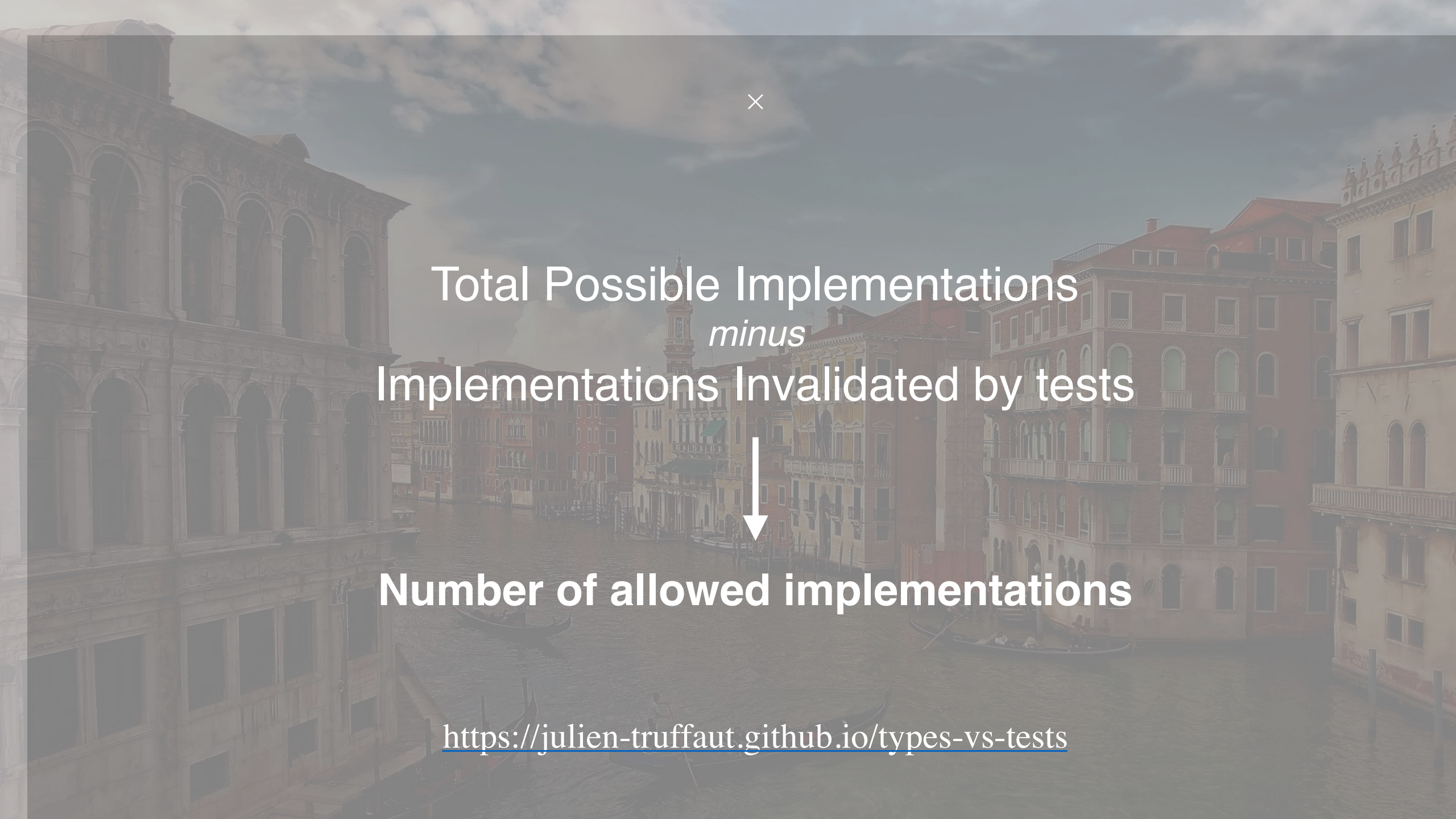
Getting a bit less philosophical



How can we constrain the implementation of a function in such a way that that the only possible implementation is the correct one?



How can we rule-out all incompatible implementations?


×
Total Possible Implementations
minus
Implementations Invalidated by tests
Number of allowed implementations
https://julien-truffaut.github.io/types-vs-tests

We can be sure that our function is correct when:

Number of allowed implementations = 1

https://julien-truffaut.github.io/types-vs-tests

We can be sure that our function is correct when:

Number of allowed implementations = 1

The one we want

https://julien-truffaut.github.io/types-vs-tests

Julien Truffaut's

Valid Implementation Count

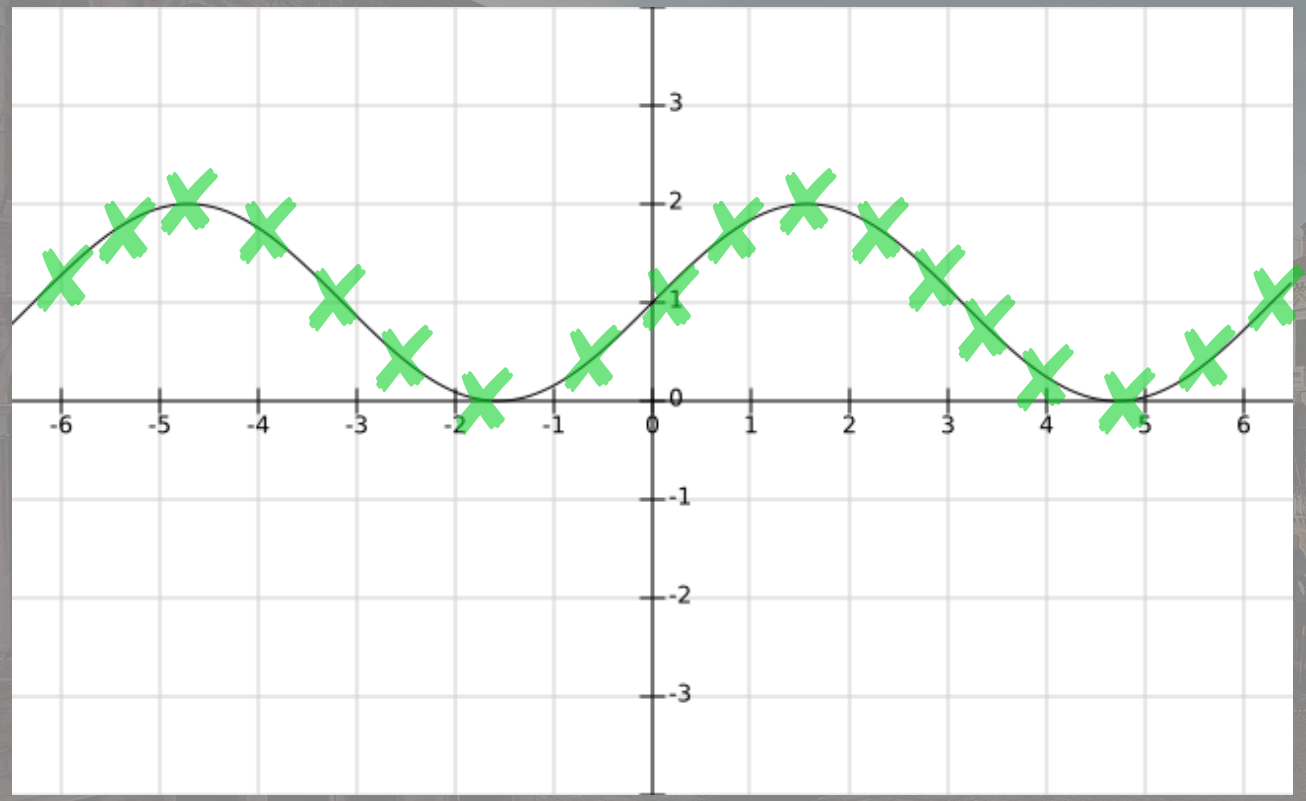
Smaller VICs mean more constrained functions.

https://julien-truffaut.github.io/types-vs-tests

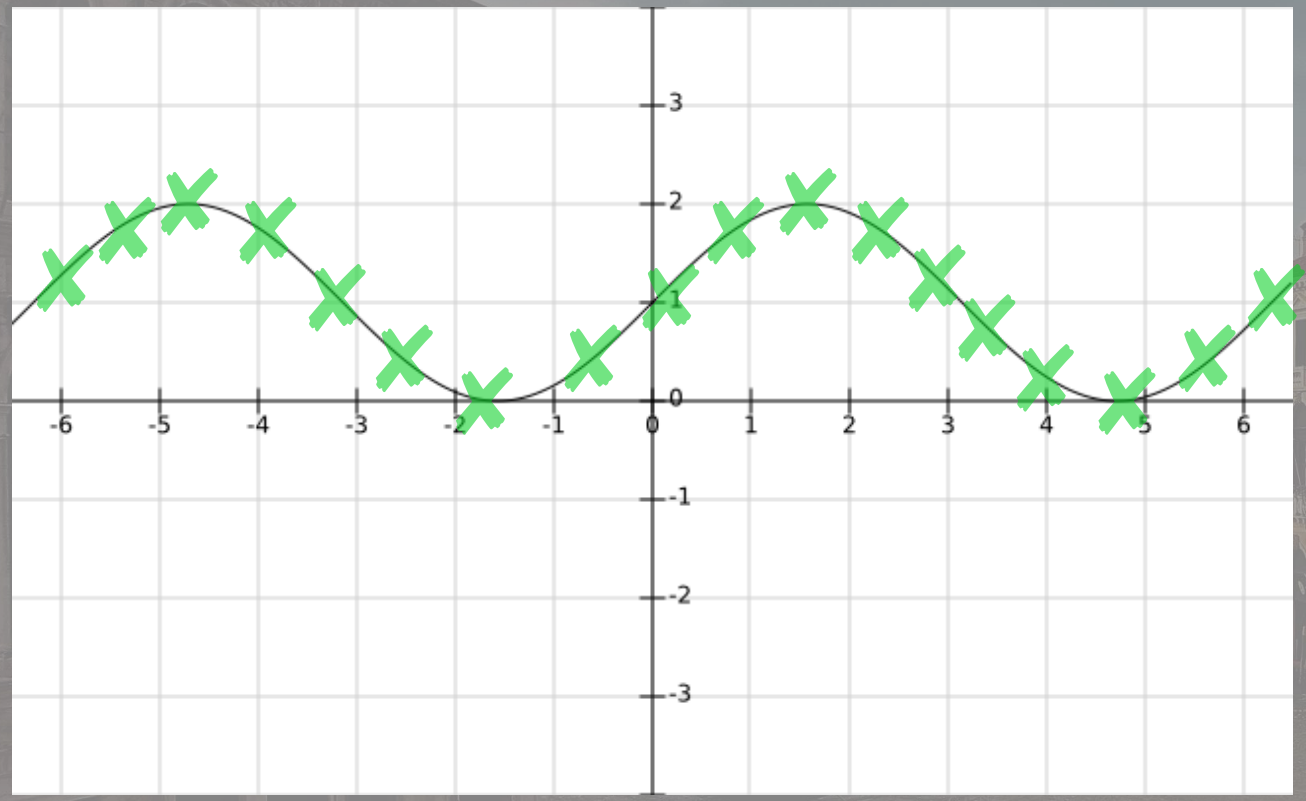
Valid Implementation Count = 1

Only one possible implementation: the correct one

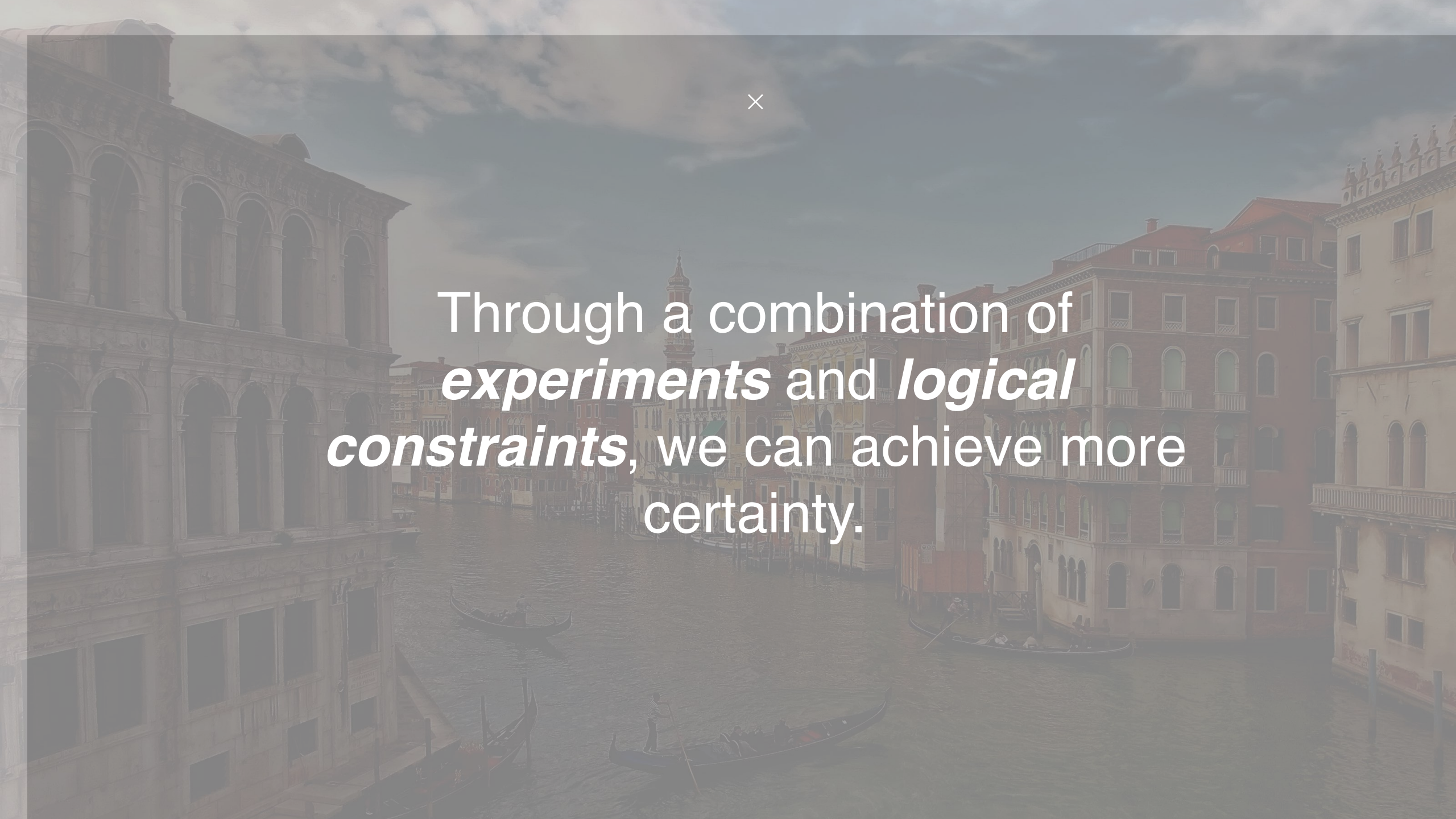
https://julien-truffaut.github.io/types-vs-tests



Cover every possible input and output

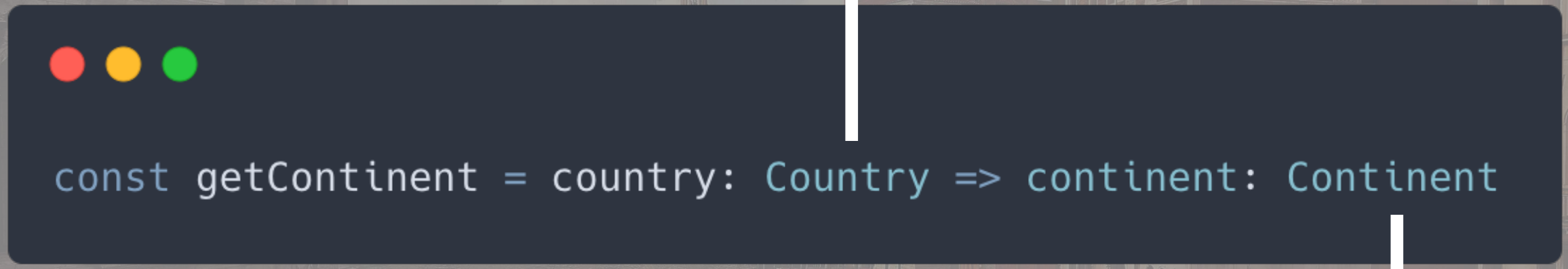


It's only feasible to cover every I/O pair by constraining them


Through a combination of
experiments and ***logical***
constraints, we can achieve more
certainty.


const getContinent = country: String => continent: String
How many possible values can we have for each of these?

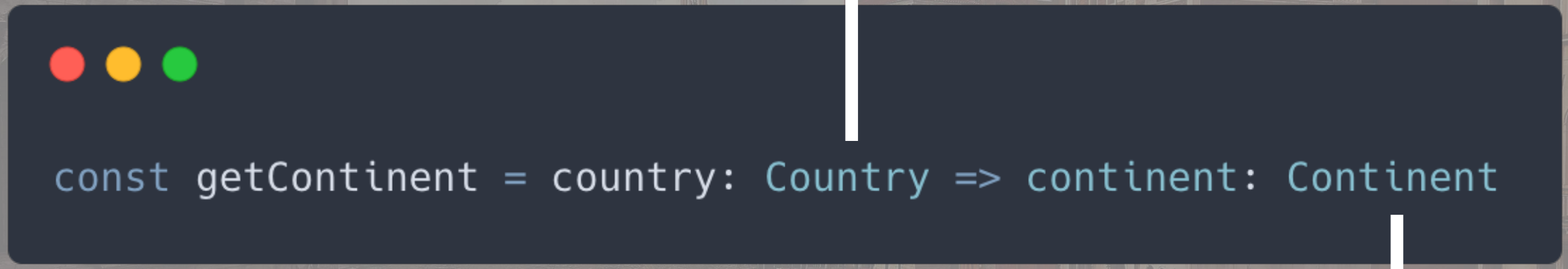
195



7

There are now 7^{195} valid implementations.
Just because we added types.

195



For every input tested, we reduce the number of possible implementations seven-fold.

7

195

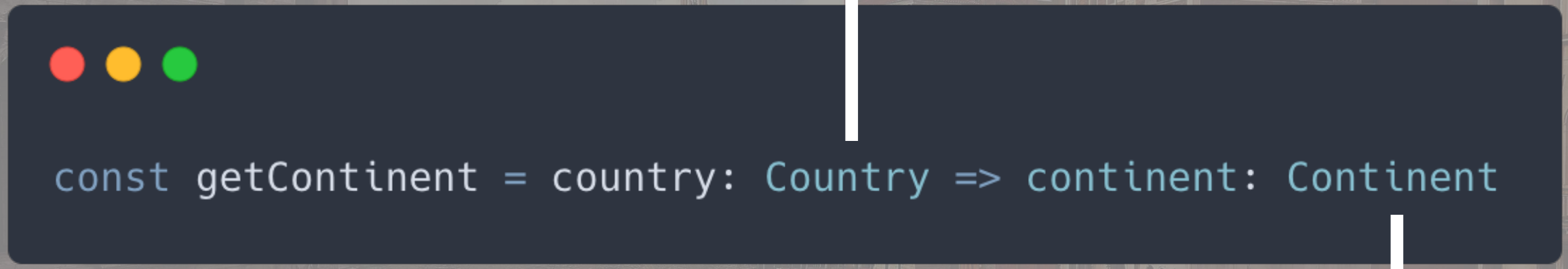
```
const getContinent = country: Country => continent: Continent
```

For every country tested, the possibilities of its result collapse into being only one.

$7^{195} / 7 = 7^{194}$

7

195



It's now feasible to test all countries and
collapse the possible implementations into one.

$7^{195} / 7 / 7 / 7 / 7 \ldots = 7^{0} = 1$

7

Type systems allow us to constrain reality in such a way that testing all possible inputs become possible.

Making impossible states impossible

"Making Impossible States Impossible" by Richard Feldman

Richard Feldman

ELM CONF

16 - Patrick Stapfer - Making Unreasonable States Impossible

Patrick Stapfer

REACT FINLAND

Leveraging correctness with types

FORMALLY ENFORCING CORRECTNESS

Making impossible states impossible

Blackjack

Making impossible states impossible

Blackjack

Google

pls teach me how to play blackjack

Todas | Imagens | Vídeos | Notícias | Shopping | Mais | Configurações | Ferramentas

Aproximadamente 569.000 resultados (0,60 segundos)

Clipe sugerido · 91 segundos

How To Play Blackjack - YouTube

https://www.youtube.com/watch?v=-9YGKFdP6sY

Acerca deste resultado | Comentários

Making impossible states impossible

```
type Player = {
    name: string
    email?: string
    countryCode?: string
    phone?: string
};

const playerOne: Player = { name: "Lucas" }
```

Can have a countryCode
without a phone and vice-versa

Only has a name!

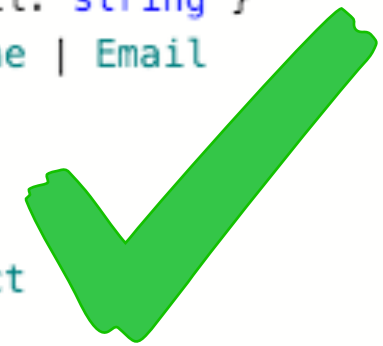
Making impossible states impossible

```
type Phone = { countryCode: string, phone: string }
type Player = {
    name: string
    email?: string
    phone?: Phone
};

const playerOne: Player = {
    name: "Lucas",
}
```

Still don't need a contact

Making impossible states impossible

```
type Phone = { countryCode: string, phone: string }
type Email = { email: string }
type Contact = Phone | Email

type Player = {
    name: string
    contact: Contact
};

const playerOne: Player = {
    name: "Lucas",
    contact: { email: "example@lucasfcosta.com"}
}
```

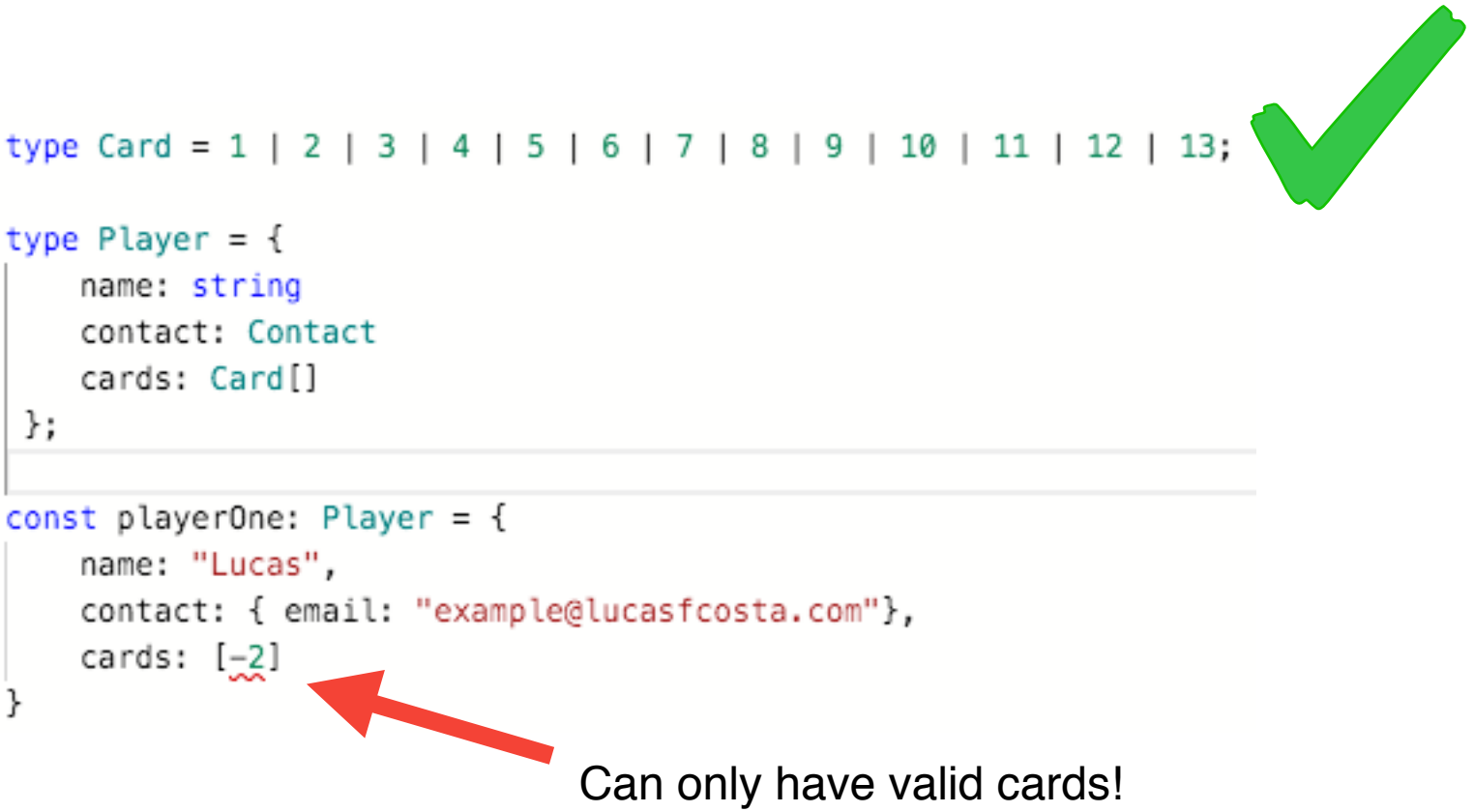


Making impossible states impossible

```
type Player = {
    name: string
    contact: Contact
    cards: number[]
};

const playerOne: Player = {
    name: "Lucas",
    contact: { email: "example@lucasfcosta.com"},
    cards: [-2]
}
```


We can have invalid cards!
Zero, negative numbers, huge numbers...

Making impossible states impossible



Making impossible states impossible

```
type Card = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13;

type Player = {
    name: string
    contact: Contact
    cards: Card[]
};

const playerOne: Player = {
    name: "Lucas",
    contact: { email: "example@lucasfcosta.com"},
    cards: [1]
}
```

Making impossible states impossible

```
type Card = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13;

type Player = {
    name: string
    contact: Contact
    cards: Card[]
};

const playerOne: Player = {
    name: "Lucas",
    contact: { email: "example@lucasfcosta.com"},
    cards: [1]
}
```

We can still have only one card!

Making impossible states impossible

```ts
type Card = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13;
type TwoOrMore<T> = { 0: T, 1: T } & Array<T>
type Hand = TwoOrMore<Card>

type Player = {
    name: string
    contact: Contact
    cards: Hand
};

const playerOne: Player = {
    name: "Lucas",
    contact: { email: "example@lucasfcosta.com"},
    cards: [1]
}
```

We must have two or more cards

Making impossible states impossible

```
type Game = {
    players: Player[]
    currentState: "win" | "tie" | "running"
    winner: Player
    tiePlayers: Player[],
    nextPlayer: Player
};
```

Can have zero players

lol wrong

Can have a 'win' state with a next player

Can have a running state with a winner

Can have a tie with no tiePlayers

Making impossible states impossible

States are now constrained

```
type Win =  { name: "win", winner: Player }
type Tie =  { name: "tie", tiePlayers: TwoOrMore<Player> }
type Unfinished = { name: "running", nextPlayer: Player }
type GameState = Win | Tie | Unfinished

type Game = {
    players: Player[]
    turn: Player
    gameState: GameState
};
```

Can have a next player which has busted or surrendered!

Making impossible states impossible

```
type PlayingState = "playing"
type NonPlayingState = "surrendered" | "busted" | "winner" | "loser" | "tie"
type PlayerState<T> = { state: T }
type PlayingPlayer = Player & PlayerState<PlayingState>
type NonPlayingPlayer = Player & PlayerState<NonPlayingState>
```

Making impossible states impossible

```
type Win =  { name: "win", winner: Player }
type Tie =  { name: "tie", tiePlayers: TwoOrMore<Player> }
type Unfinished = { name: "running", nextPlayer: PlayingPlayer }
type GameState = Win | Tie | Unfinished
```

Can only have valid players now :)

Contents [hide]

For a next time

From the most complex to the most simple problems

All languages are typed

But some will only tell you at runtime.

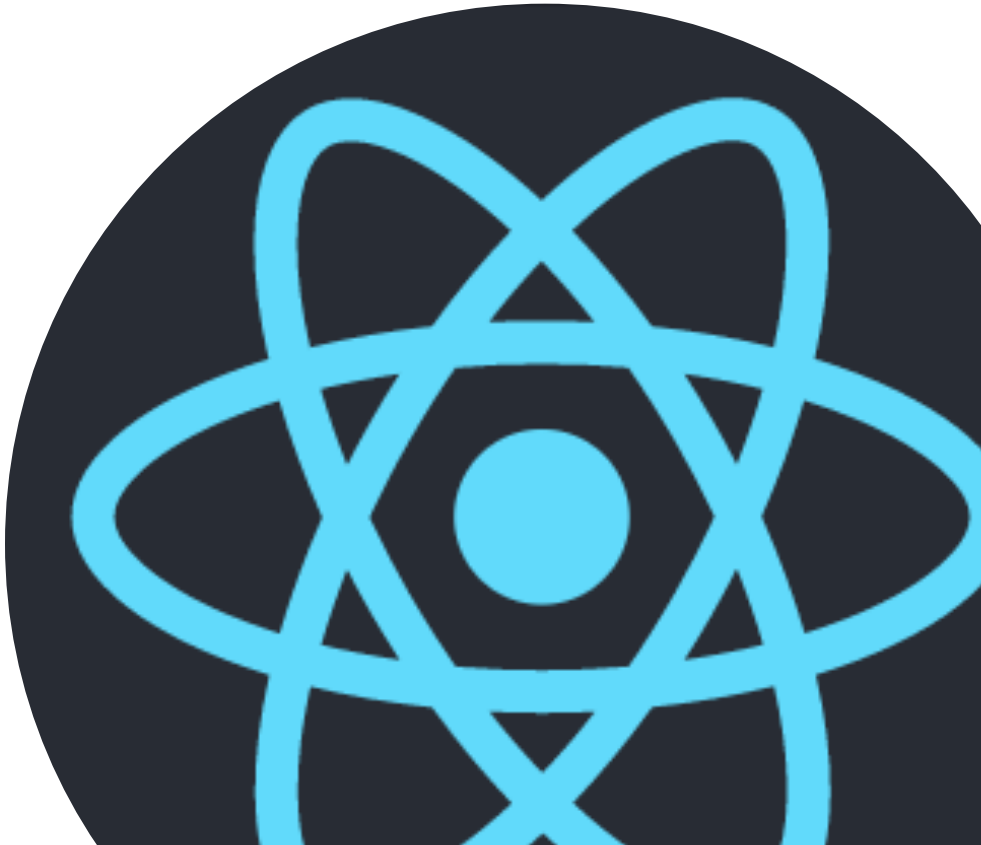
```
Uncaught TypeError: undefined is not a function
```

```
Uncaught TypeError: Cannot read property 'foo' of undefined
```

Prohibiting
invalid properties

Collapsing the number of possible props

Combinatorial Explosion



Exhaustive Checks

No unhandled actions.

```
type ADD = { type: 'ADD', payload: number };
type SUBTRACT = { type: 'SUBTRACT', payload: number };
type Events = ADD | SUBTRACT;

const unhandledAction = (value: never): never => { throw new Error(`Unhandled action`) }

function process(event: Events) {
  switch(event.type) {
    case 'ADD':
      break;
    case 'SUBTRACT':
      break;
    default:
      unhandledAction(event);
  }
}
```

Ensures unhandledAction is never called!

All actions are handled so this can't happen!

Exhaustive Checks

No unhandled actions.

```ts
type ADD = { type: 'ADD', payload: number };
type SUBTRACT = { type: 'SUBTRACT', payload: number };
type UNKNOWN = { type: 'UNKNOWN', payload: number };
type Events = ADD | SUBTRACT | UNKNOWN;

const unhandledAction = (value: never): never => { throw new Error(`Unhandled action`) }

function process(event: Events) {
  switch(event.type) {
    case 'ADD':
      break;
    case 'SUBTRACT':
      break;
    default:
      unhandledAction(event);
  }
}
```

We're not handling unknown!

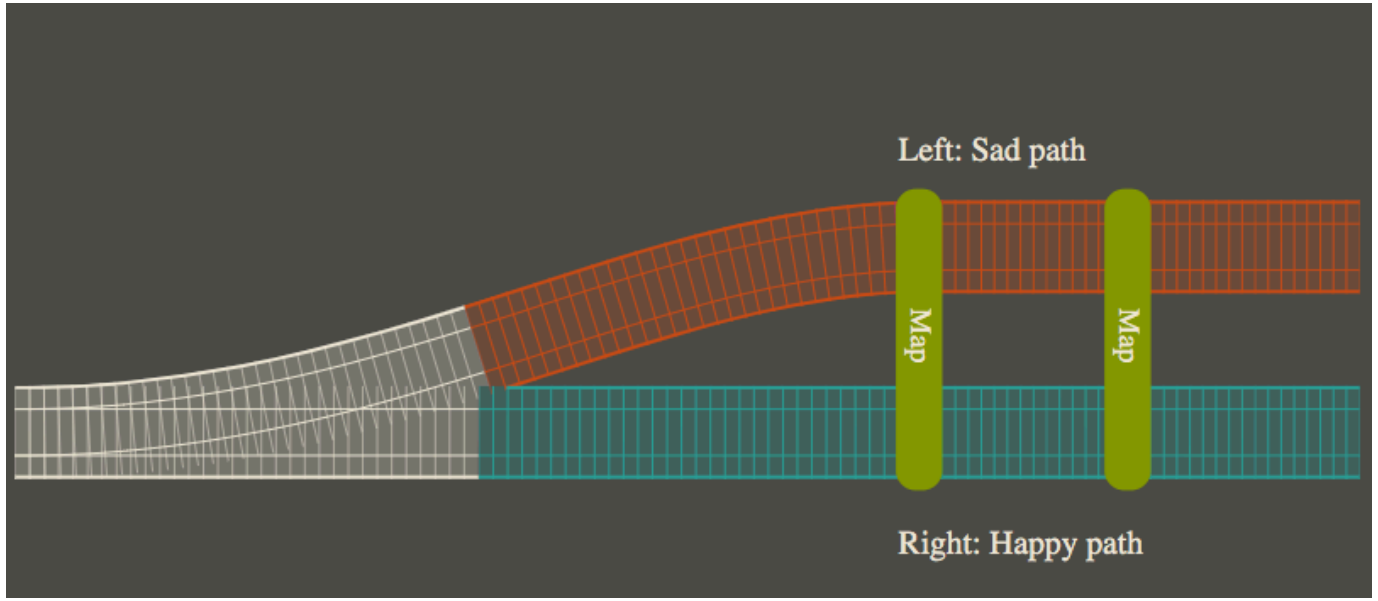
Exhaustive Checks

No unhandled actions.

```ts
type ADD = { type: 'ADD', payload: number };
type SUBTRACT = { type: 'SUBTRACT', payload: number };
type UNKNOWN = { type: 'UNKNOWN', payload: number };
type Events = ADD | SUBTRACT | UNKNOWN;

const unhandledAction = (value: never): never => { throw new Error(`Unhandled action`) }

function process(event: Events) {
  switch(event.type) {
    case 'ADD':
      break;
    case 'SUBTRACT':
      break;
    default:
      unhandledAction(event);
  }
}
```

We're not handling unknown!

Modeling Uncertainty



https://fsharpforfunandprofit.com/rop/ - Scott Wlaschin

Producing reliable software depends on choosing good abstractions and executing the correct experiments.

Leveraging
correctness
with ***tests***

AUTOMATED CRAP IS STILL CRAP

×

WRITING TESTS IS NOT ENOUGH

Coupling and cost management

What is the cost of having tests?

What value does having tests produce?

How brittle should my tests be?

Capitalism 101

Capitalism 101

What matters: less costs, more revenue

Capitalism 101

What matters: less costs, more revenue

What doesn't matter: code coverage, correctness, tests

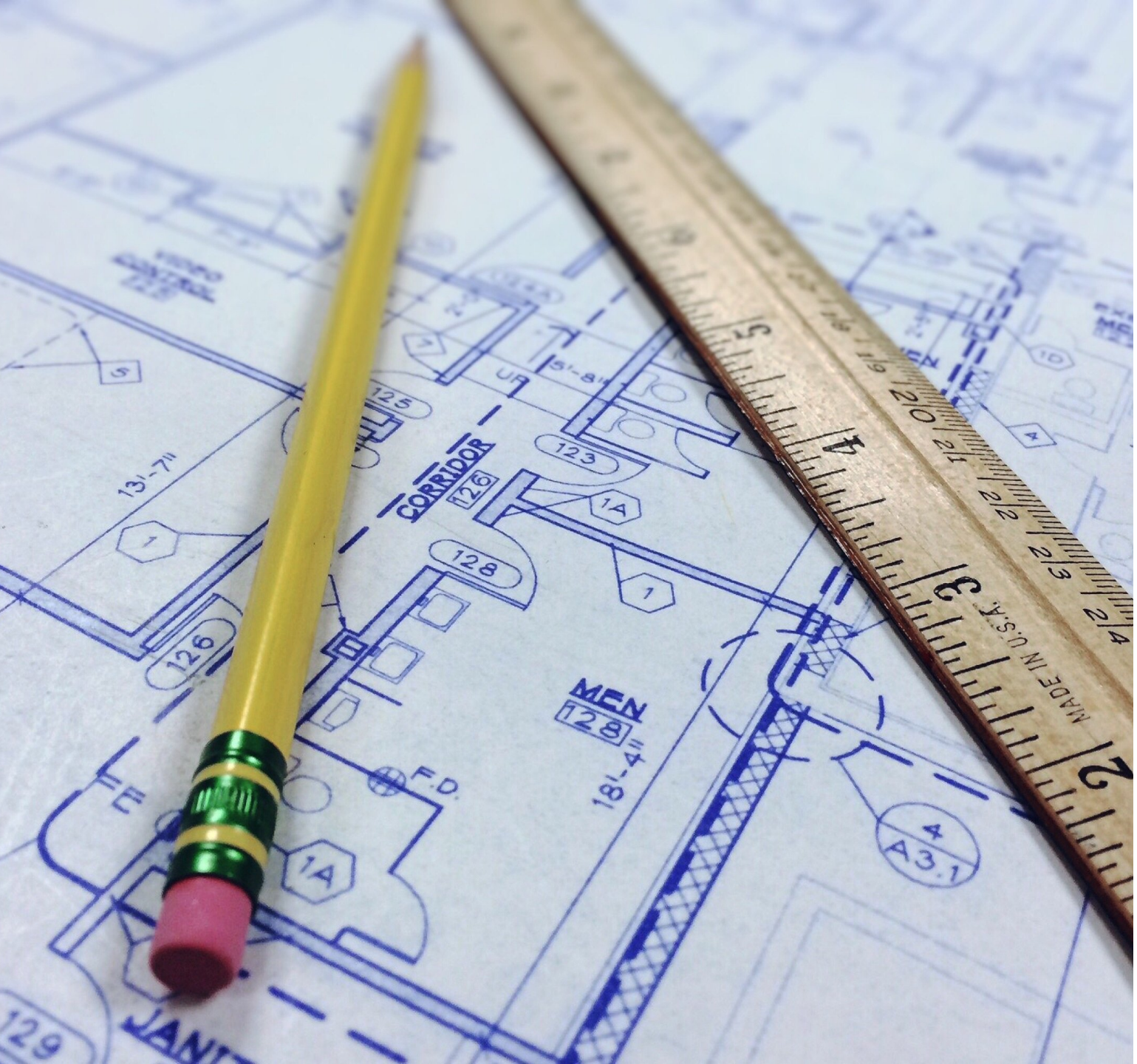
THE PRICE

Tests add upfront cost

But reduce long-term costs.
You get back in instalments.
Tests are subject to diminishing returns.

You pay for tests in maintenance

TESTS ARE
CODE TOO



THE PRICE

Avoid coupling.

The more tests you have to change when you do a change, the bigger your cost is.

THE PRICE

Tests shouldn't be too fragile nor too loose.

Think about when you would like them to break.

THE PRICE

Tests shouldn't be too fragile nor too loose.

Tests that never fail are useless. They don't produce any information.

THE PRICE

Tests shouldn't be too fragile nor too loose.

Tests that never fail are not scientific.

THE PRICE

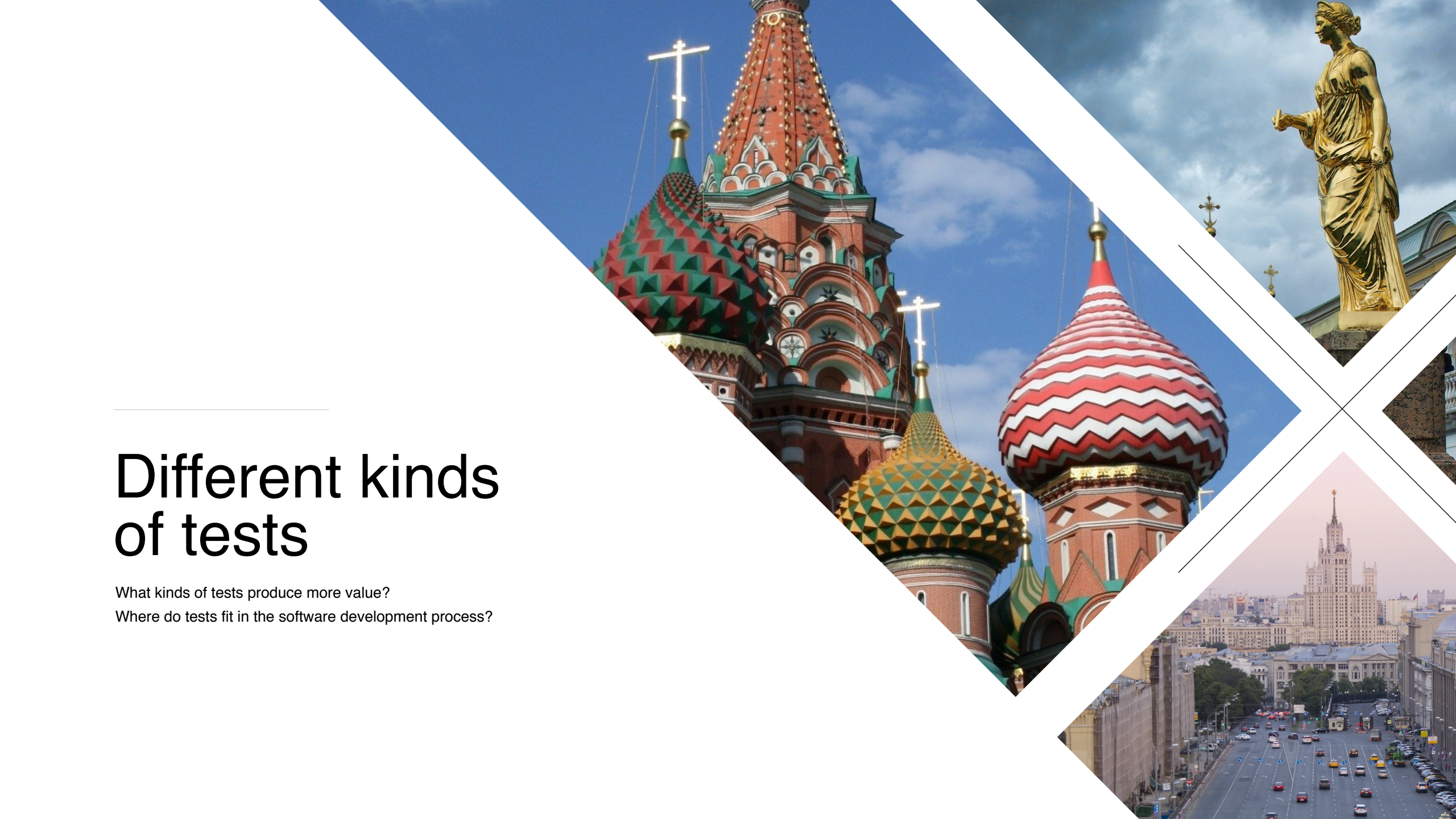
Tests shouldn't be too fragile nor too loose.

Tests that never fail are pseudo-science.

Different kinds of tests

What kinds of tests produce more value?

Where do tests fit in the software development process?




We need to talk about
Test Driven Development

Test Driven Development is *not* about tests

TDD is about taking small steps.

TDD is a fear reduction tool.

TDD exists to help orienting developers when they change code.

Test Driven Development is *not* about tests

TDD is about
TDD is a fear
TDD exists to
when they ch

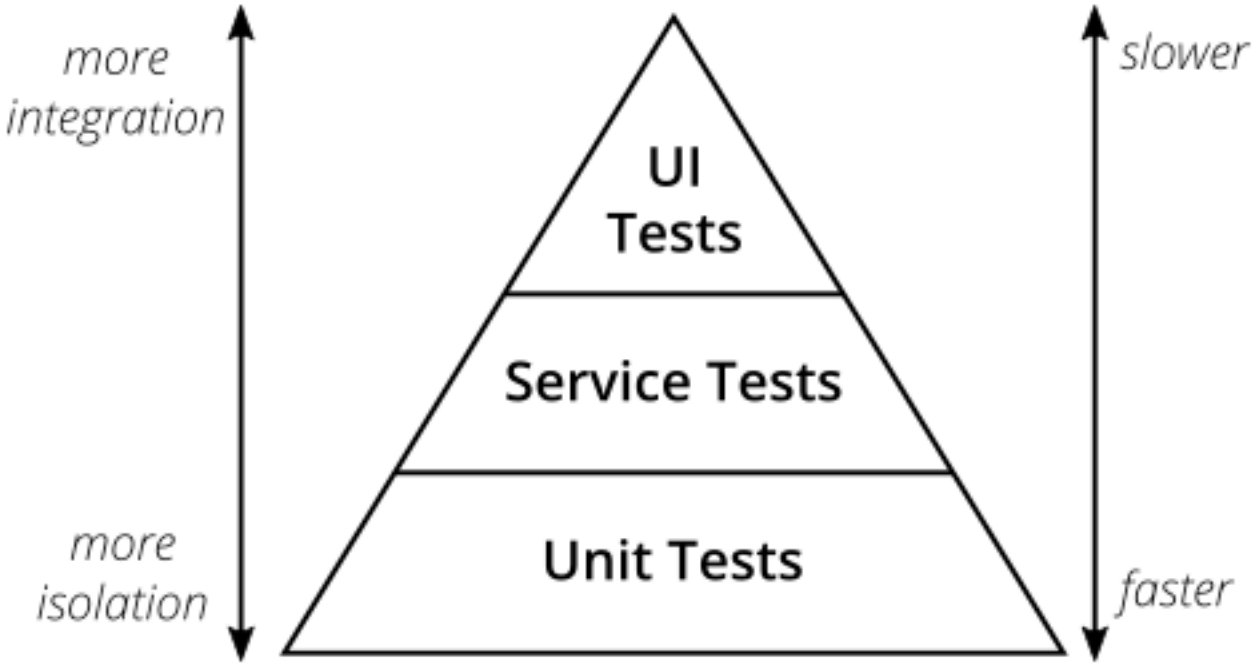


Figure 2: The Test Pyramid

From MartinFowler.com
The Practical Test Pyramid — Written by Ham Vocke

https://martinfowler.com/articles/practical-test-pyramid.html

Different types of tests generate different types of value.

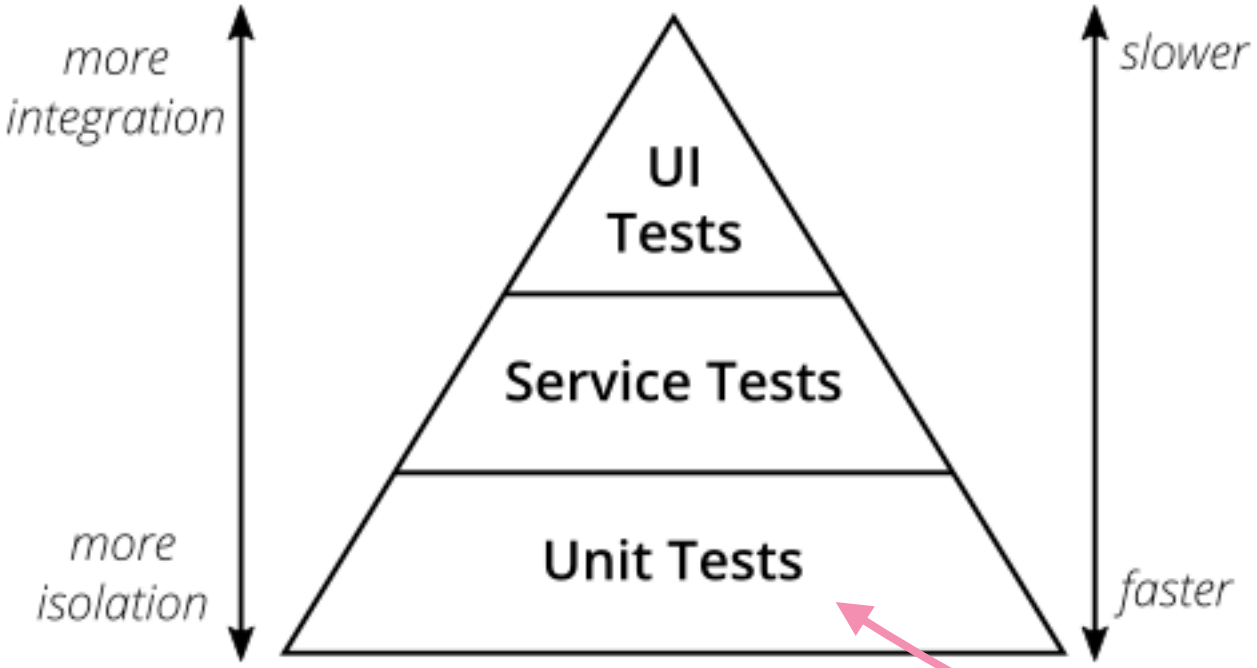


Figure 2: The Test Pyramid

From MartinFowler.com
The Practical Test Pyramid — Written by Ham Vocke

https://martinfowler.com/articles/practical-test-pyramid.html

Different types of tests generate different types of value.

Cheap, and fast
but produce relatively low value

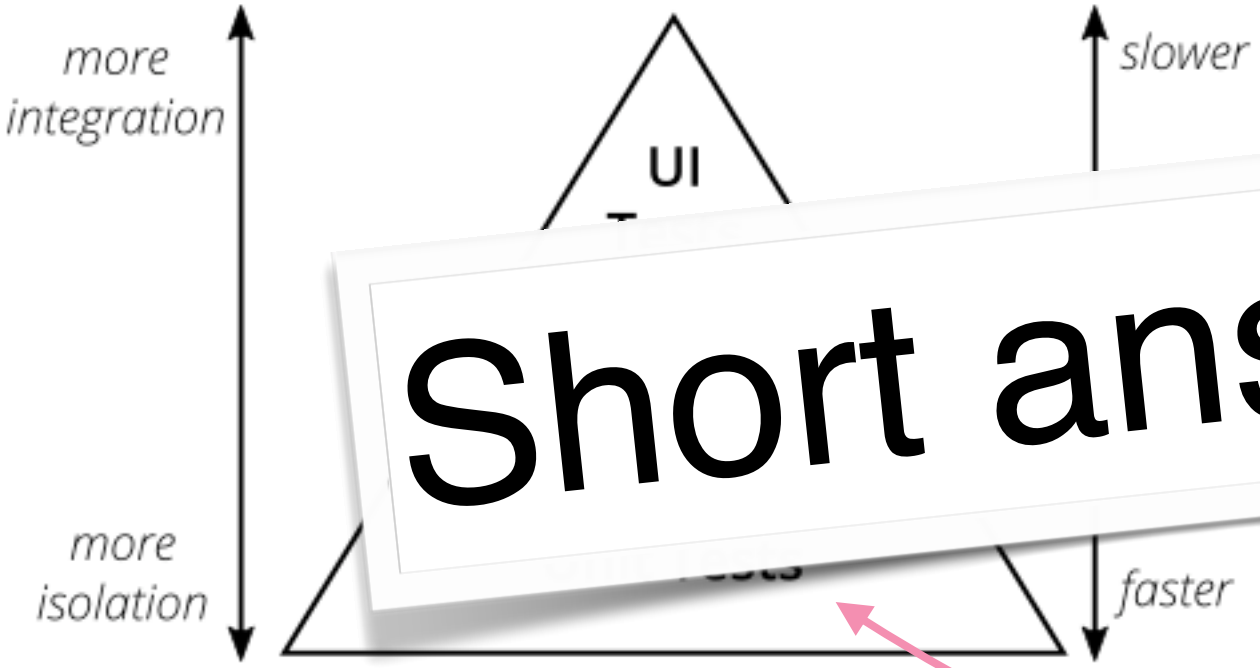


Figure 2: The Test Pyramid

Short answer:

Diff… …pes

different types
of value.

From MartinFowler.com
The Practical Test Pyramid — Written by Ham Vocke

https://martinfowler.com/articles/practical-test-pyramid.html

Cheap, and fast
but produce relatively low value

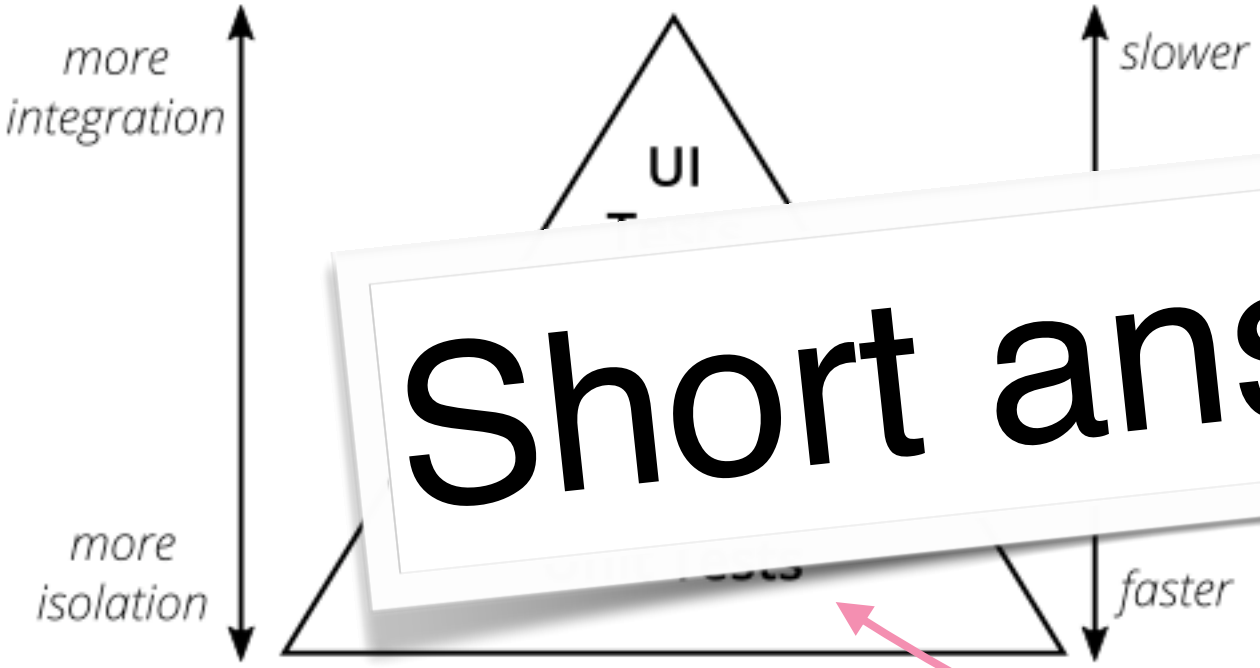


Figure 2: The Test Pyramid

Short answer: no

Different types of value.

From MartinFowler.com
The Practical Test Pyramid — Written by Ham Vocke

https://martinfowler.com/articles/practical-test-pyramid.html

Cheap, and fast
but produce relatively low value

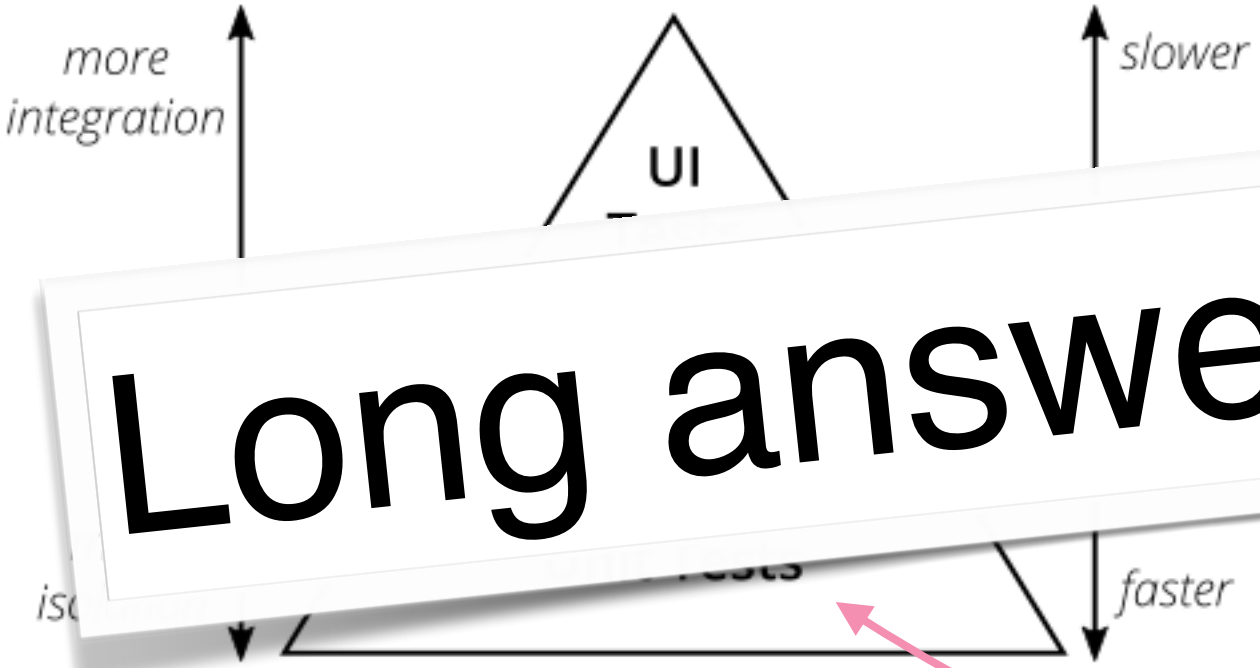


Figure 2: The Test Pyramid

From MartinFowler.com
The Practical Test Pyramid — Written by Ham Vocke

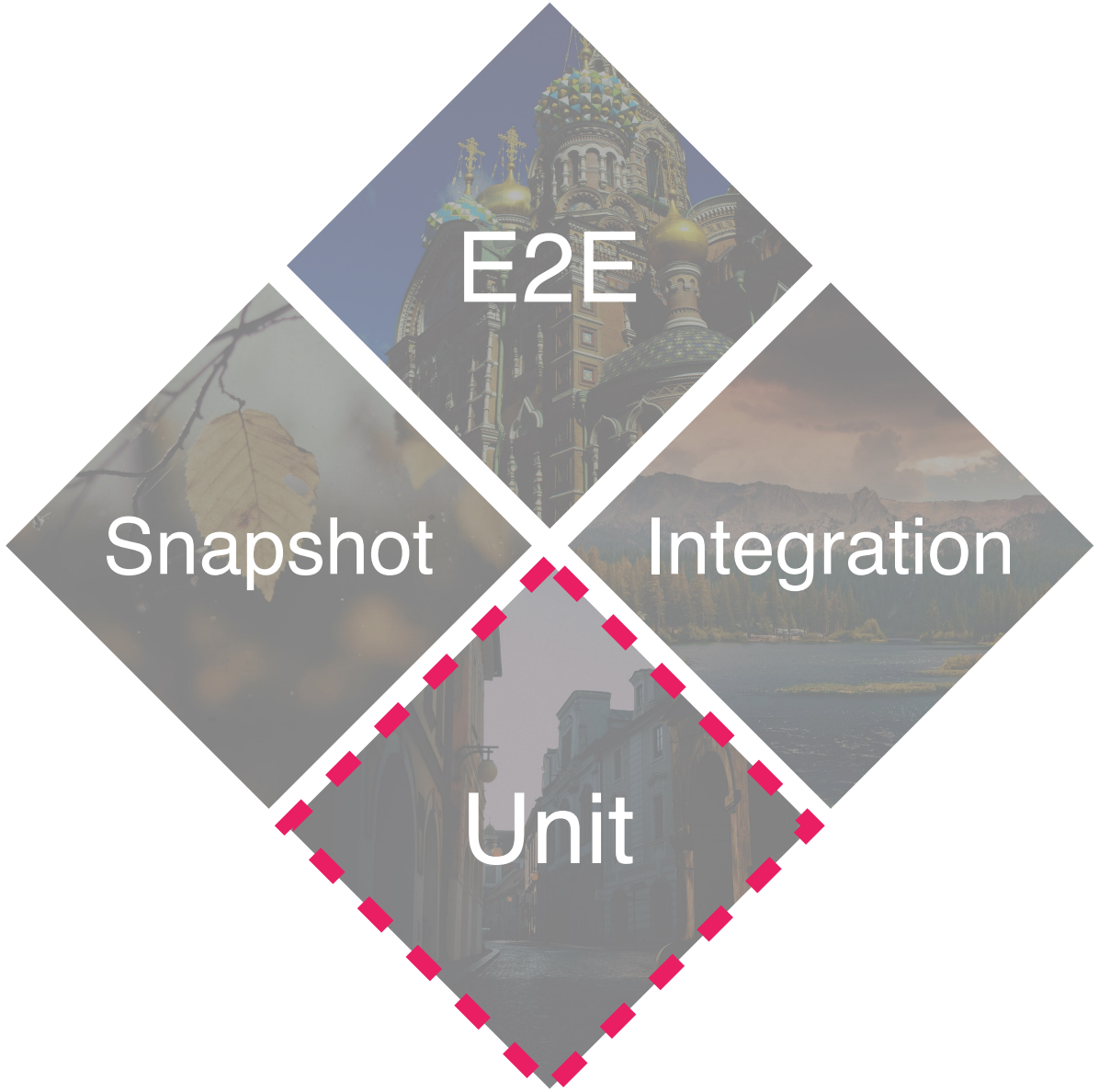
https://martinfowler.com/articles/practical-test-pyramid.html

Long answer: well…

different types
of value.

Cheap, and fast
but produce relatively low value

When to write what

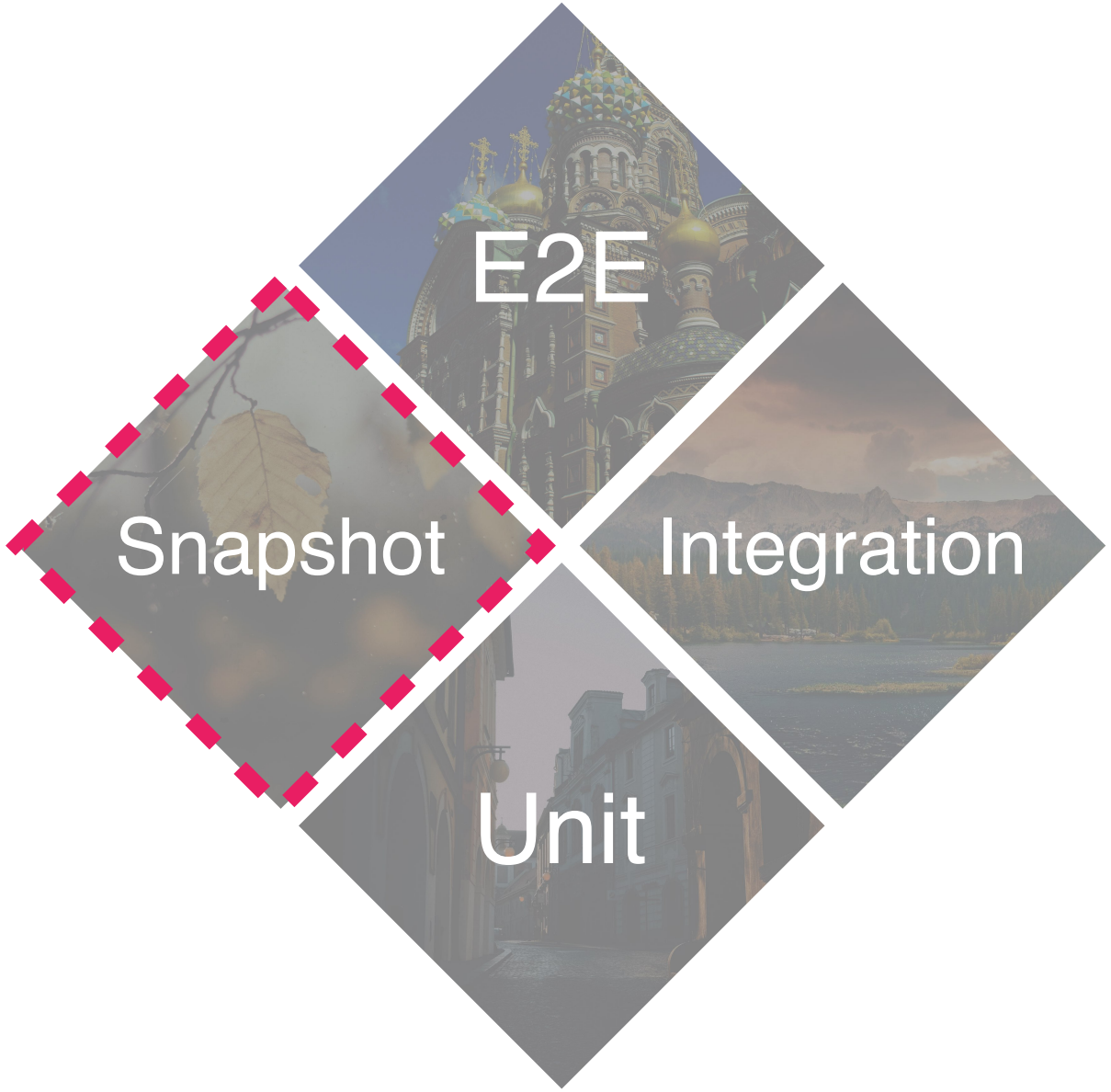


When should I write unit tests?

In parallel with writing code.

Unit tests guide development and
help refactoring code safely.

Not mainly for correctness.
As documentation for your future self.
As a contract, specification of the unit under test.



When should I write snapshot tests?

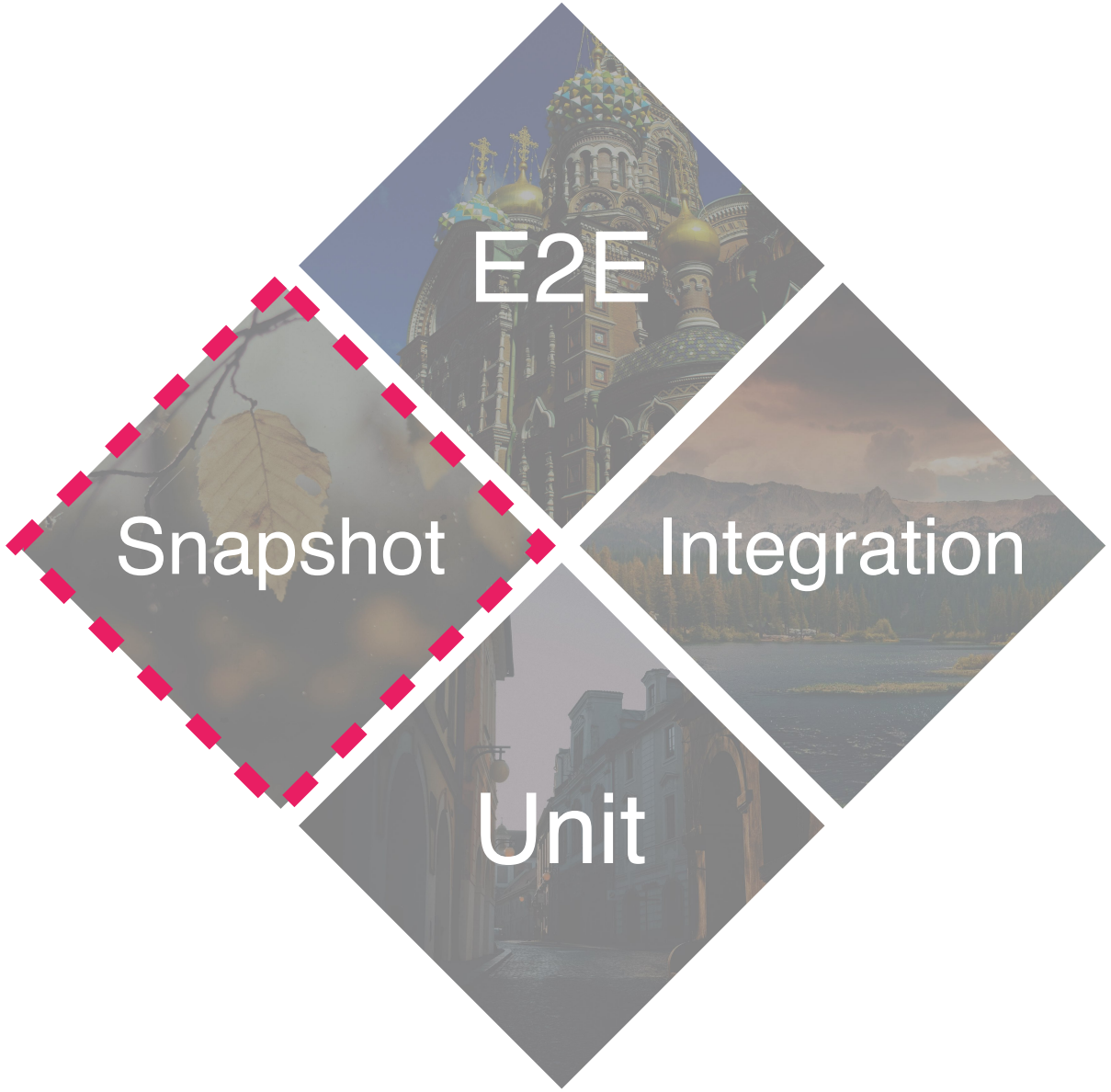
Not as guidance for iterating, but as an extra safeguard against failure.

When asserting on output is repetitive.

When output is too big and detailed to manage.

When should I write snapshot tests?

Jest extensively uses snapshots to test itself.



A few tips for snapshot tests

```
"snapshotSerializers": ["enzyme-to-json/serializer"]
```

jest-snapshot-serializer-raw

npm v1.1.0 | build unknown | coverage 100%

jest snapshot serializer for reducing escapes in the snapshot file

react-native-jest-serializer

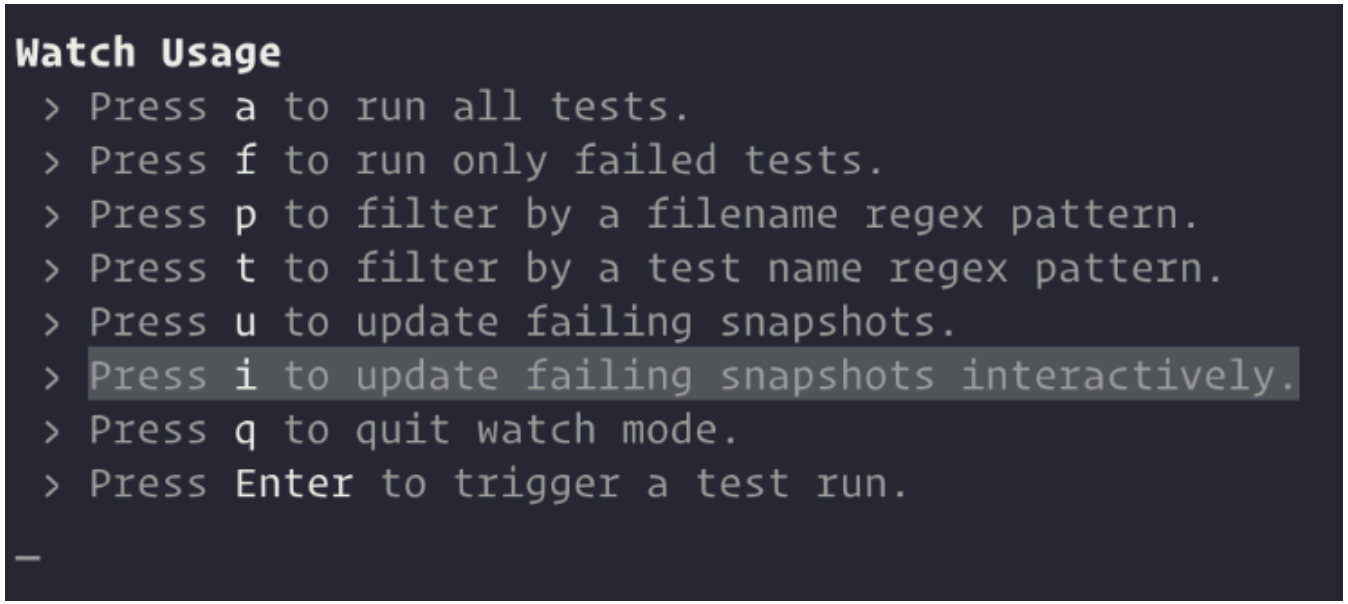
1.1.0 • Public • Published 2 months ago

Readme

react-native-jest-serializer

A few tips for snapshot tests

- Find relevant serialisers for your problem domain. Readable snapshots matter.



A few tips for snapshot tests

- Find relevant serialisers for your problem domain. Readable snapshots matter.

```
expect(person).toMatchSnapshot({
  name: expect.stringMatching(/(.?) (?)/g),
  pets: expect.arrayContaining(["Dog"]),
  createdAt: expect.any(Date)
});
```

`expect.extend(matchers)`

You can use `expect.extend` to add your own matchers to Jest.

jest-image-snapshot

2.11.0 • Public • Published 2 months ago

Readme

jest-image-snapshot

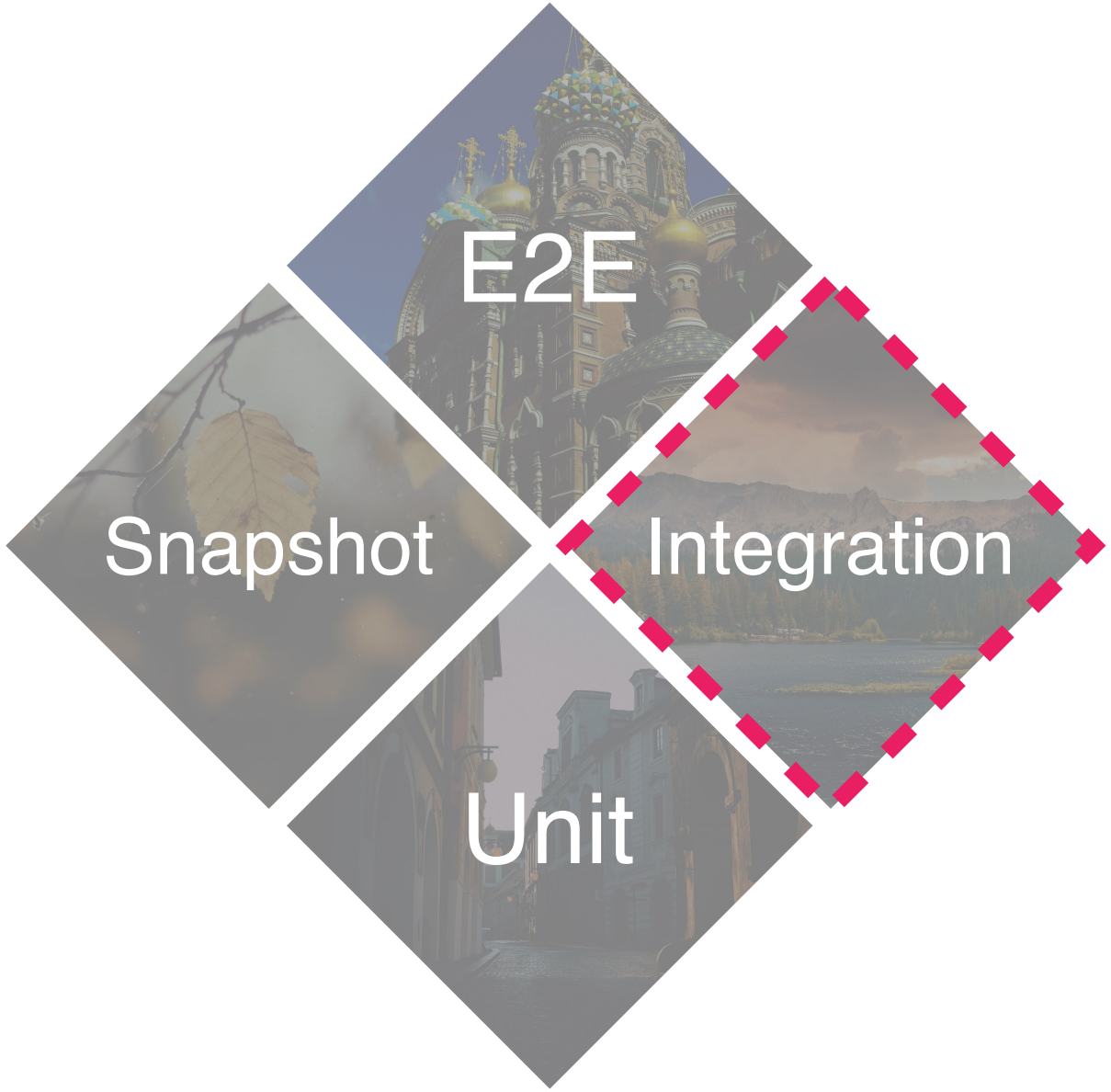
A few tips for snapshot tests

- Find relevant serialisers for your problem domain. Readable snapshots matter.
- Use custom asymmetric snapshot matchers to balance maintainability and rigorousness

```
it('increments count', () => {
  const bigComponentInstance = shallow(<MyBigComponent />);
  expect(bigComponentInstance.find(".bigChunkOfMarkup")).toMatchSnapshot();
  expect(bigComponentInstance.find(".result").text()).to.be.equal(0);

  bigComponentInstance.find(".incrementButton").simulate("click");
  expect(bigComponentInstance.find(".bigChunkOfMarkup")).toMatchSnapshot();
  expect(bigComponentInstance.find(".result").text()).to.be.equal(1)
});
```

A few tips for snapshot tests

- Find relevant serialisers for your problem domain. Readable snapshots matter.
- Use custom asymmetric snapshot matchers to balance maintainability and rigorousness
- Don't be afraid to have tests with partial snapshots.



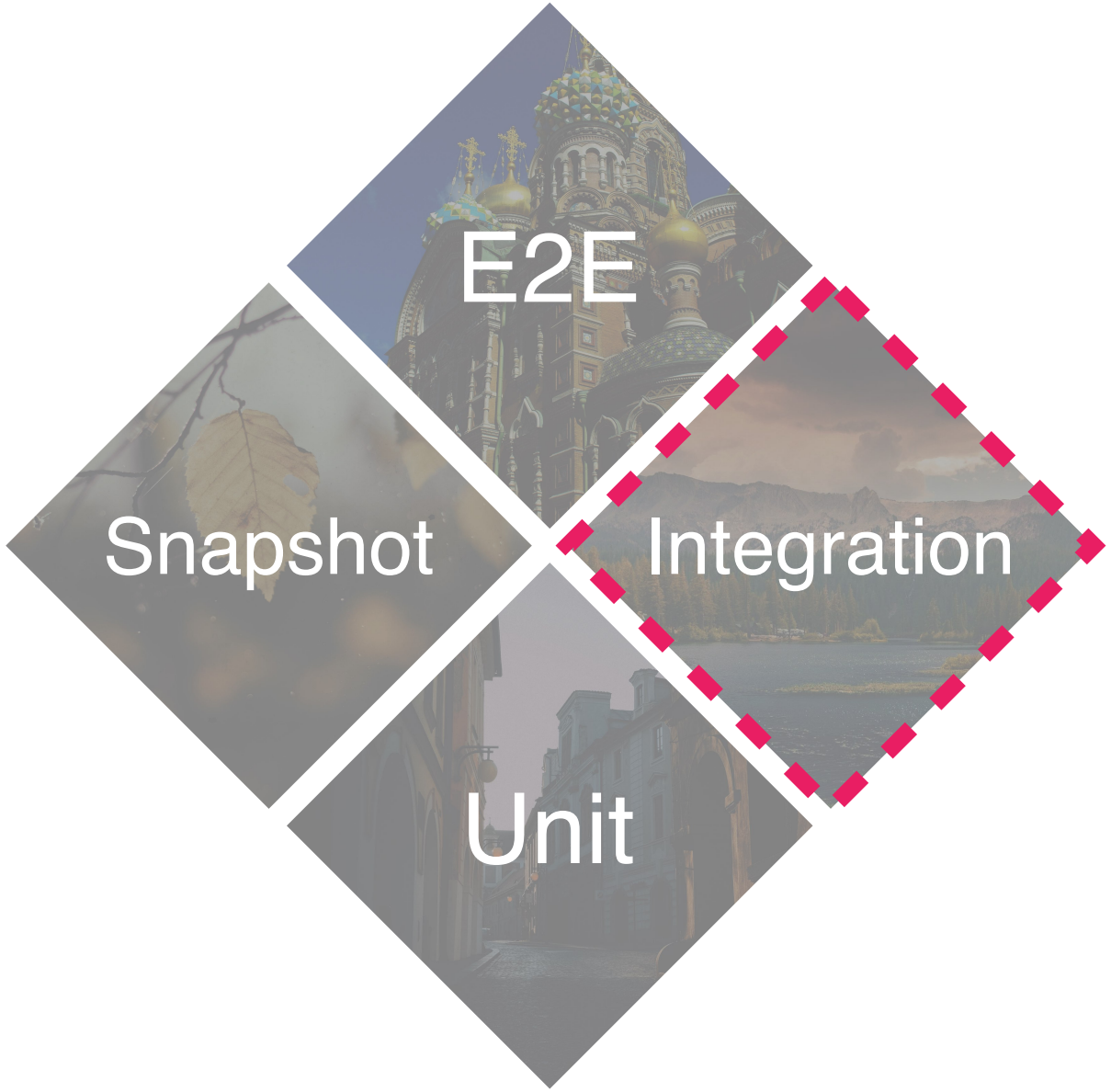
When should I write integration tests?

To test functional requirements.

To ensure correctness and prevent regressions.

To ensure you are using third party dependencies correctly.

When a certain behaviour is critical to your application.



Practices that I consider integration tests:

- Interacting with actual components (Enzyme/ react-testing-library)
- Sending actual HTTP requests
- Hitting a database and fetching data from it
- Asserting on I/O (i.e. interacting with the filesystem)
- Spinning separate processes

When should I write end-to-end tests?

When interaction with a real UI matters.

To avoid visual regressions.

To ensure multiple services work together from a user's perspective.

The most valuable kind of testing from a correctness perspective.

Can't emphasise how good this is:

cypress

- Amazing docs
- Easy access to your application's runtime environment
- Not flaky (but be careful with the global chain of events!)
- Extremely quick to run
- Extremely easy to setup

Avoiding false positives

How can I setup tests in such a way as to catch the most bugs?

How can I avoid getting false positives?

How do assertion libraries work?

Assertions that are loose by design

- .includes
- .isDefined
- .increases
- .decreases

The set of passing results is too broad

AVOIDING FALSE POSITIVES

Avoid loose assertions

Assertions which allow multiple different outputs.

Assertions that are loose by design

```
expect(result).to.be.a.number
```

AVOIDING FALSE POSITIVES

Avoid loose assertions

Assertions which allow multiple different outputs.

Assertions that are loose by design

```
expect(result).to.be.a.number
```

Can go from 5e-324 to 1.79769313486231570e+308

AVOIDING FALSE POSITIVES

Avoid loose assertions

Assertions which allow multiple different outputs.

Assertions that are loose by design

```
expect(result).to.be.a.number
```

AVOIDING FALSE POSITIVES

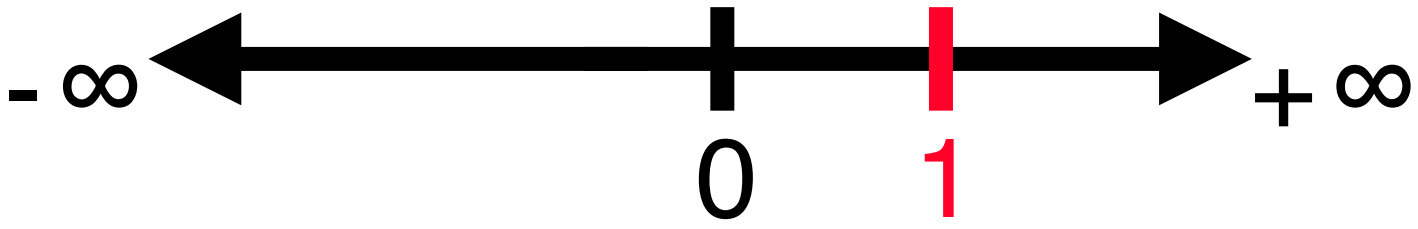
Avoid loose assertions

Assertions which allow multiple different outputs.

Negated assertions.

```
expect(result).to.not.be(1)
```

Negated assertions are the
loosest assertions one can make.



AVOIDING FALSE POSITIVES

Avoid loose assertions

Assertions which allow
multiple different outputs.

Loose assertions are essentially assertions with a semantic "or"

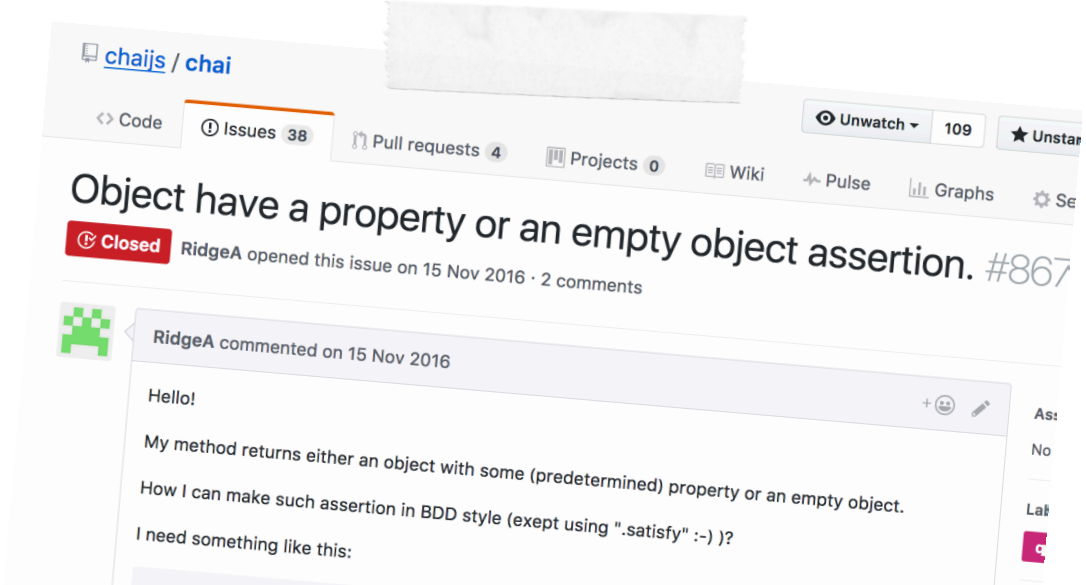
```
expect(result).to.be(1).or.to.be(2)
```

AVOIDING FALSE POSITIVES

Avoid loose assertions

Assertions which allow multiple different outputs.

This is why *.or* has never been included in Chai.



AVOIDING FALSE POSITIVES

Avoid loose assertions

Assertions which allow multiple different outputs.

```
expect(result).to.not.throw(TypeError, "example msg")
```

Is it an error if both don't match?
What if one matches and the other doesn't?

AVOIDING FALSE NEGATIVES

Assert on one subject at a time

Build inputs and expected outputs within your testing code.

```
const catFactory = (color) => (name) => ({ name, color, species: "cat" })
const blueCatFactory = catFactory("blue");

describe("catFactory", () => {
  it("can create blue cats", () => {
    const expected = blueCatFactory("Ludo");
    const actual = catFactory("blue")("Ludo");
    expect(expected).to.be.deep.equal(actual);
  });
});
```

circular assertion

Using application code to do tests means the correctness of the test depends on the correctness of the application itself.

AVOIDING FALSE POSITIVES

Avoid tautological tests

Don't test your code against itself.

Meaningful Feedback

What is the right size of a test?
How to debug in a scientific manner?
How do test runners work?

MEANINGFUL FEEDBACK

Choose the right assertions

Assertion libraries generate information for test runners to show you meaningful output.

Class: assert.AssertionError

- Extends: <errors.Error>

Indicates the failure of an assertion. All errors thrown by the `assert` module will be instances of the `AssertionError` class.

new assert.AssertionError(options)

Added in: v0.1.21

- `options` <Object>
  - `message` <string> If provided, the error message is set to this value.
  - `actual` <any> The `actual` property on the error instance.
  - `expected` <any> The `expected` property on the error instance.
  - `operator` <string> The `operator` property on the error instance.
  - `stackStartFn` <Function> If provided, the generated stack trace omits frames before this function.

A subclass of `Error` that indicates the failure of an assertion.

MEANINGFUL FEEDBACK

How test runners provide output

Assertions produce `AsertionError` instances.

jest-diff

Display differences clearly so people can review changes confidently.

The default export serializes JavaScript **values**, compares them line-by-line, and returns a string which includes comparison lines.

Two named exports compare **strings** character-by-character:

- `diffStringsUnified` returns a string.
- `diffStringsRaw` returns an array of `Diff` objects.

Three named exports compare **arrays of strings** line-by-line:

- `diffLinesUnified` and `diffLinesUnified2` return a string.
- `diffLinesRaw` returns an array of `Diff` objects.

```
- Expected
+ Received

  Array [
-   "delete",
    "common",
-   "changed from",
+   "changed to",
+   "insert",
  ]
```

MEANINGFUL FEEDBACK

Diffs are the runner's responsibility

Runners generate diffs based on the AssertionErrors thrown

```
const myObj = {};

function c() {
}

function b() {
    // Here we will store the current stack trace into myObj
    Error.captureStackTrace(myObj);
    c();
}

function a() {
    b();
}

// First we will call these functions
a();
```

MEANINGFUL FEEDBACK

Assertion libraries can help by generating meaningful errors

They can omit certain parts of the stack trace and provide meaningful information about the operators used.

```
// Now let's see what is the stack trace stored into myObj.stack
console.log(myObj.stack);

// This will print the following stack to the console:
//     at b (repl:3:7) <-- The B call is the last entry in the stack
//     at a (repl:2:1)
//     at repl:1:1 <-- Node internals below this line
//     at realRunInThisContextScript (vm.js:22:35)
//     at sigintHandlersWrap (vm.js:98:12)
//     at ContextifyScript.Script.runInThisContext (vm.js:
//     at REPLServer.defaultEval (repl.js:313:29)
//     at bound (domain.js:280:14)
//     at REPLServer.runBound [as eval] (domain.js:293:12)
//     at REPLServer.onLine (repl.js:513:10)
```

We captured the stack here

MEANINGFUL FEEDBACK

Assertion libraries can help by generating meaningful errors

They can omit certain parts of the stack trace and provide meaningful information about the operators used.

```
const myObj = {};

function d() {
    // Here we will store the current stack trace into myObj
    // This time we will hide all the frames after `b` and `b` itself
    Error.captureStackTrace(myObj, b);
}

function c() {
    d();
}

function b() {
    c();
}

function a() {
    b();
}

// First we will call these functions
a();
```

MEANINGFUL FEEDBACK

Assertion libraries can help by generating meaningful errors

They can omit certain parts of the stack trace and provide meaningful information about the operators used.

```
// Now let's see what is the stack trace stored into myObj.stack
console.log(myObj.stack);

// This will print the following stack to the console:
//     at a (repl:2:1) <-- We only get frames before `b` was called
//     at repl:1:1 <-- Node internals below this line
//     at realRunInThisContextScript (vm.js:22:35)
//     at sigintHandlersWrap (vm.js:98:12)
//     at ContextifyScript.Script.runInThisContext (vm.js:24:12)
//     at REPLServer.defaultEval (repl.js:313:29)
//     at bound (domain.js:280:14)
//     at REPLServer.runBound [as eval] (domain.js:293:12)
//     at REPLServer.onLine (repl.js:513:10)
//     at emitOne (events.js:101:20)
```

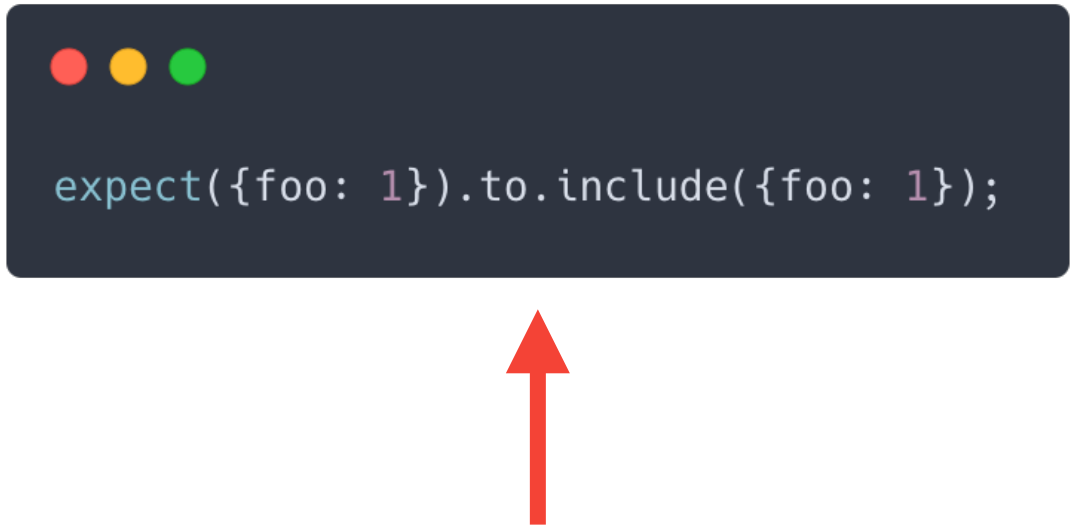
MEANINGFUL FEEDBACK

Assertion libraries can help by generating meaningful errors

They can omit certain parts of the stack trace and provide meaningful information about the operators used.

MEANINGFUL FEEDBACK

Assertion libraries can help by generating meaningful errors



They can omit certain parts of the stack trace and provide meaningful information about the operators used.

For each part of this assertion we keep resetting what is the start of the stack frame we are going to provide.

We only display the bottom stack frames, hiding our internal frames.

```
expect({ name: "HolyJS" })
```


```
new Assertion({ name: "HolyJS"})
```

MEANINGFUL FEEDBACK

Assertions behind the scenes

How Chai handles assertions.

MEANINGFUL FEEDBACK

Assertions behind the scenes

How Chai handles assertions.

Accessed properties trigger `getter` functions which always return the `assertion` object (`this`)

Each time a property is accessed, we reset the starting point of the stack.

MEANINGFUL FEEDBACK

Assertions behind the scenes

How Chai handles assertions.

```
expect({ name: "HolyJS" }).be.deep.equal({ name: "HolyJS" })
```

Accessing the property deep sets a flag called "`deep`" in the assertion object, which indicates to the `equal` assertion that it should perform a deep comparison

```
expect({ name: "HolyJS" }).be.deep.equal({ name: "HolyJS" })
```

The deep flag cannot be unset.

Do one assertion at a time!

MEANINGFUL FEEDBACK

Assertions behind the scenes

How Chai handles assertions.

MEANINGFUL FEEDBACK

More readable tests are easier to understand and debug

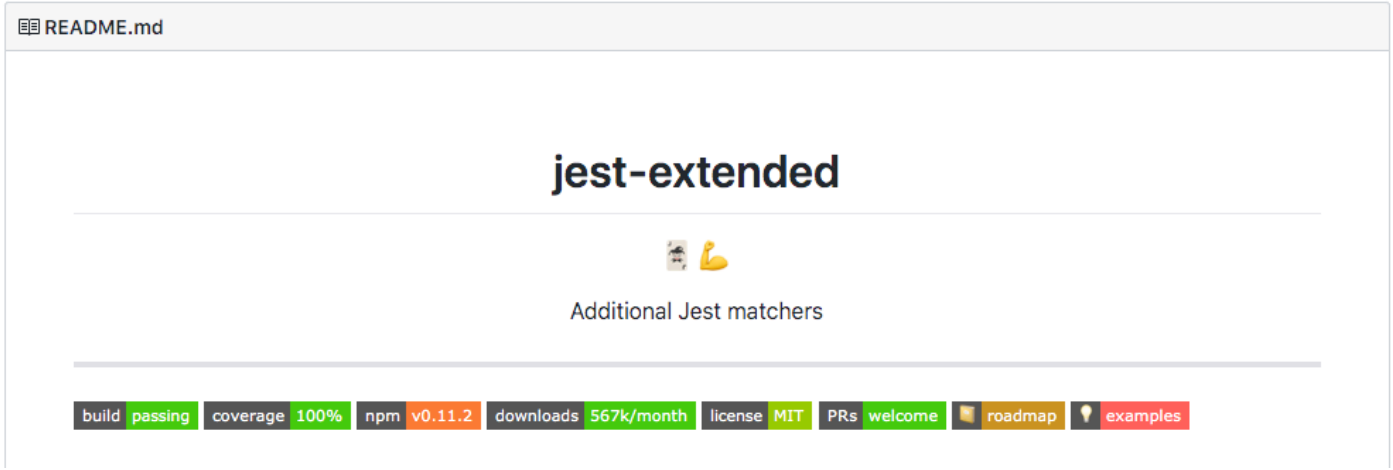
Use plugins or build your own.

```
jestExpect.extend({
  toRepeatString(actual, str, times) {
    const matches = actual.match(new RegExp(str, 'g') || [];

    const pass = matches.length > 0
    const message = pass
      ? () => `expected ${actual} to include ${str} ${times} times`
      : () => `expected ${actual} to not include ${str} ${times} times`;

    return {message, pass};
  }
});
```

MEANINGFUL FEEDBACK

More readable tests are easier to understand and debug

Use plugins or build your own.



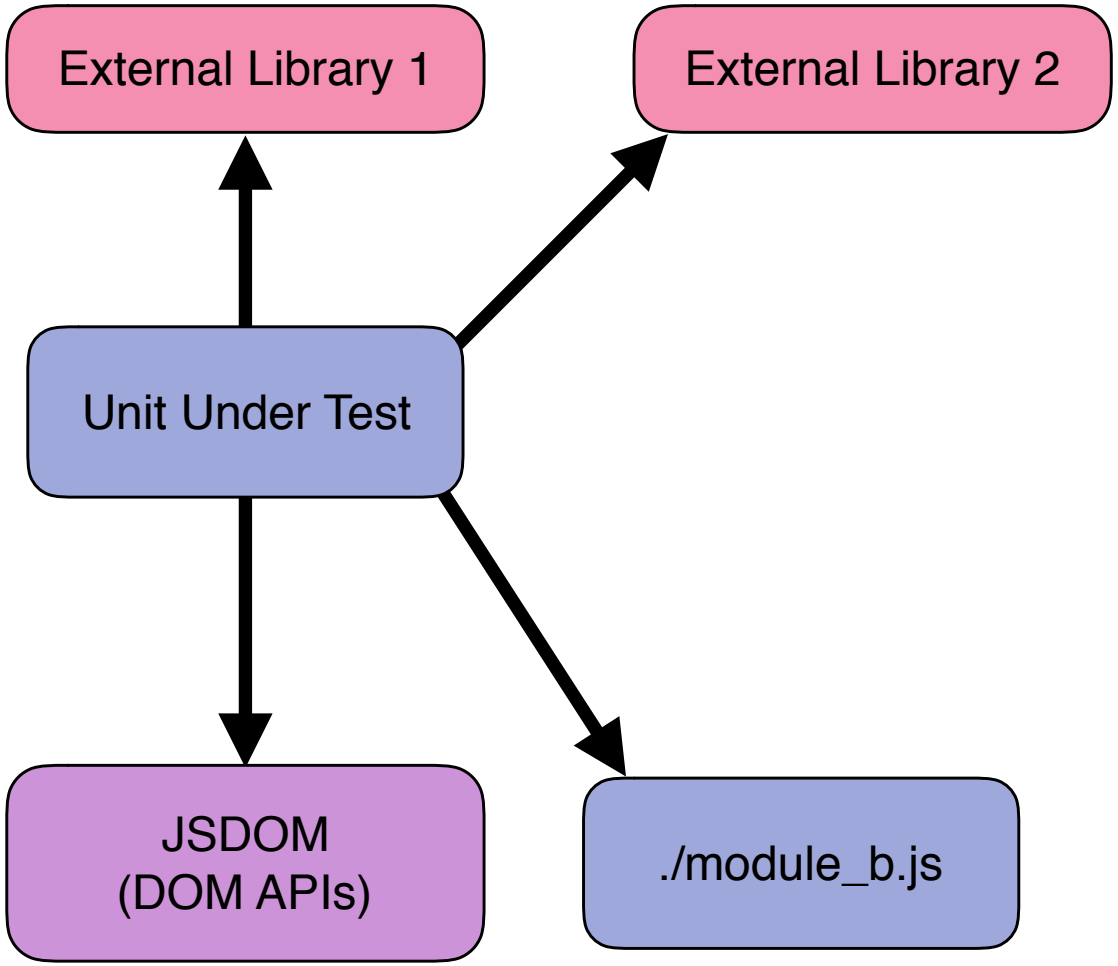
MEANINGFUL FEEDBACK

More readable tests are easier to understand and debug

Use plugins or build your own.

Isolating external dependencies

What parts of my application should I test and when?
How can I eliminate dependency on external libraries?



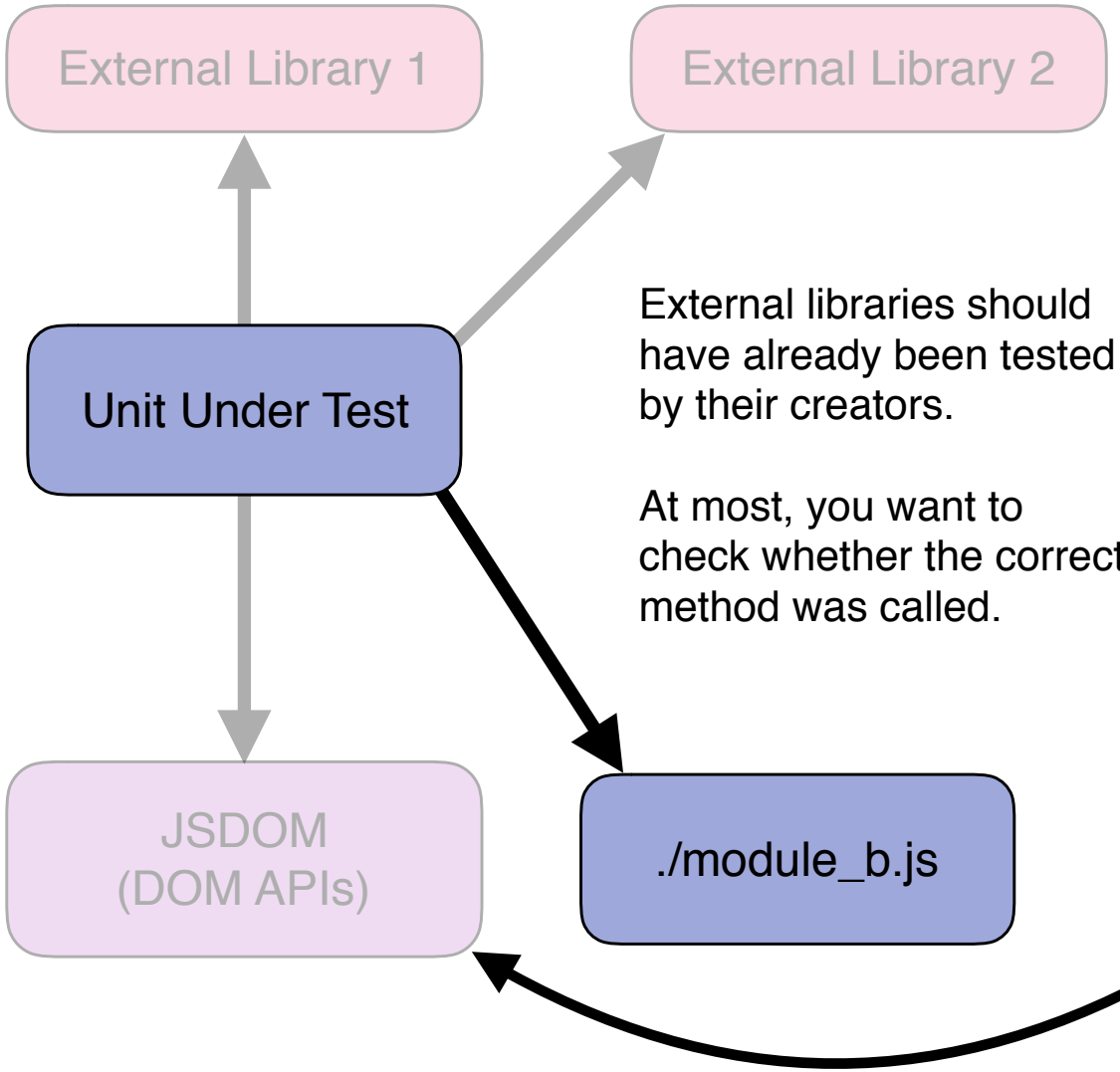
TEST ISOLATION

When should I mock?

Easy Answer: Mock what is not yours.

Hard Answer: It depends.

Test your code, not someone else's.



External libraries should have already been tested by their creators.

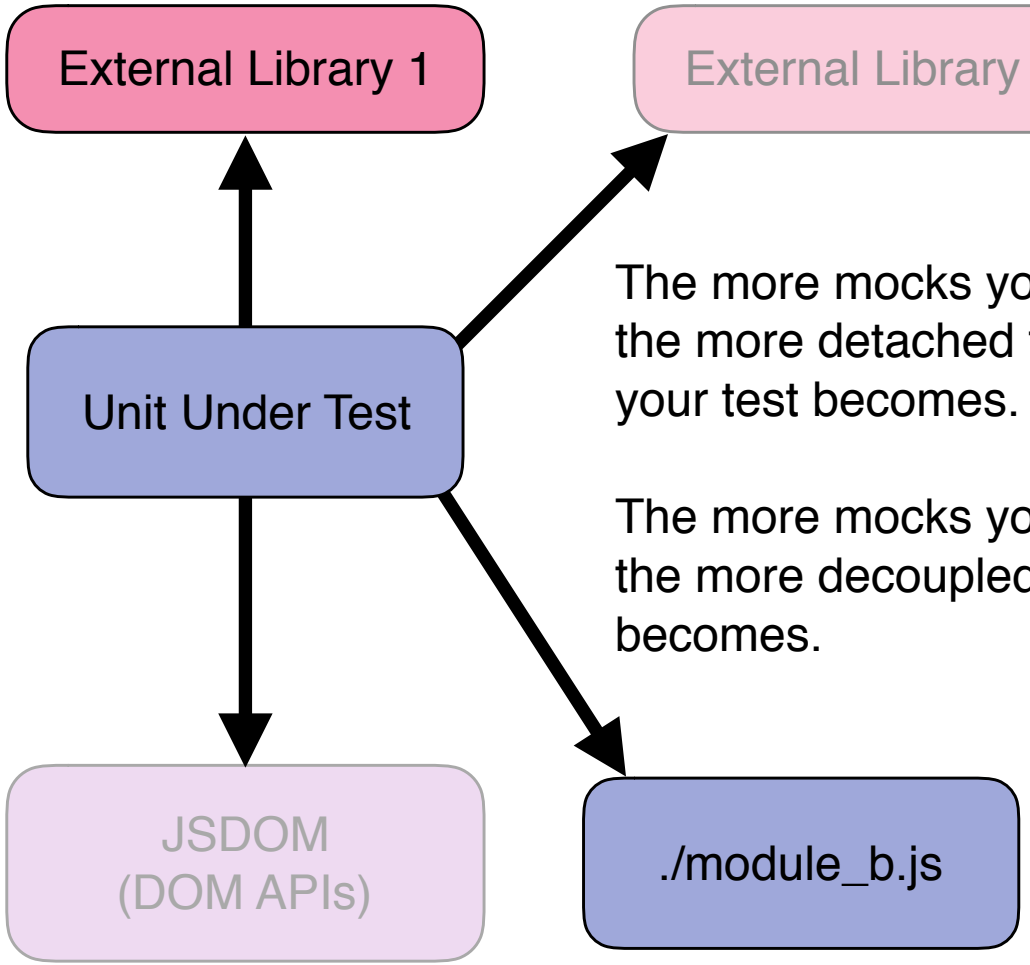
At most, you want to check whether the correct method was called.

Especially valid for DOM APIs!

TEST ISOLATION

When should I mock?

Easy Answer: Mock what is not yours.



The more mocks you have, the more detached from reality your test becomes.

The more mocks you have, the more decoupled your test becomes.

TEST ISOLATION

When should I mock?

Easy Answer: Mock what is not yours.

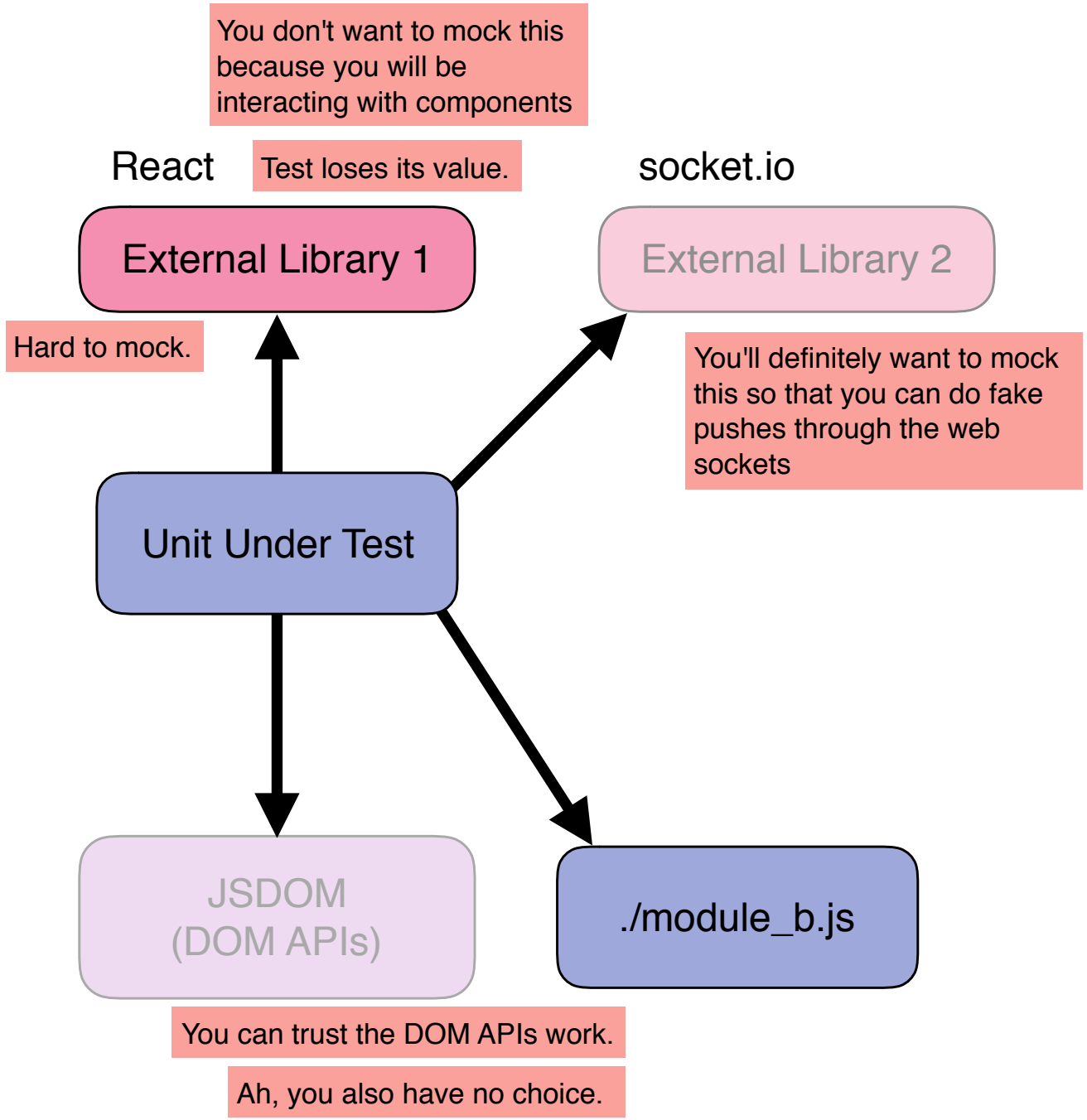
Hard Answer: It depends.

↓

Maintenance Cost vs. Isolation

How coupled to the dependency is the mock?

How critical is the code under test?



TEST ISOLATION

When should I mock?

Easy Answer: Mock what is not yours.

Hard Answer: It depends.

↓

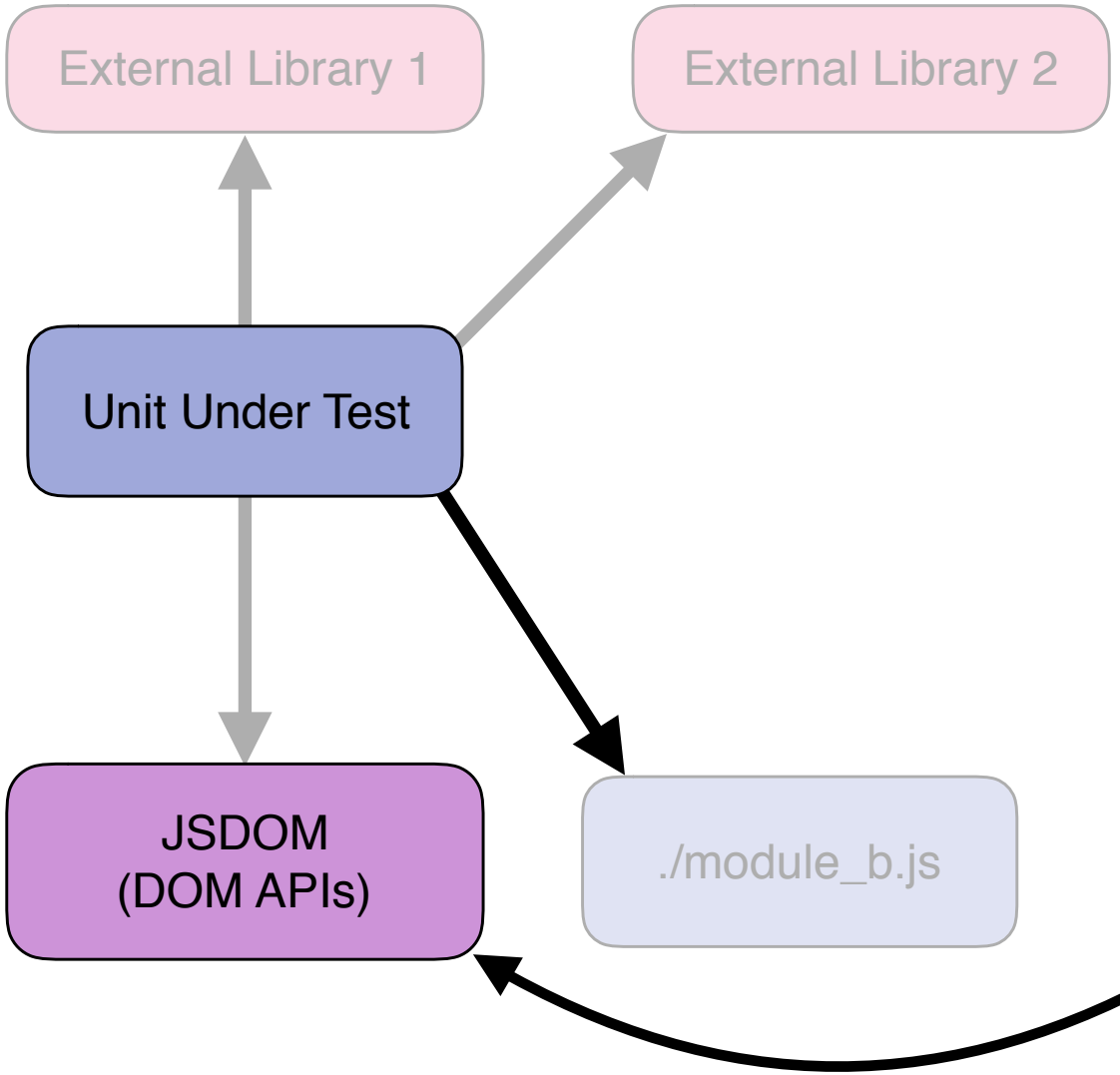
Maintenance Cost vs. Isolation

How coupled to the dependency is the mock?

How critical is the code under test?

TEST ISOLATION

What do you mean by trusting browser APIs?



Trust that they work,
but check that you are
using them

JSDOM
(DOM APIs)

jsdom

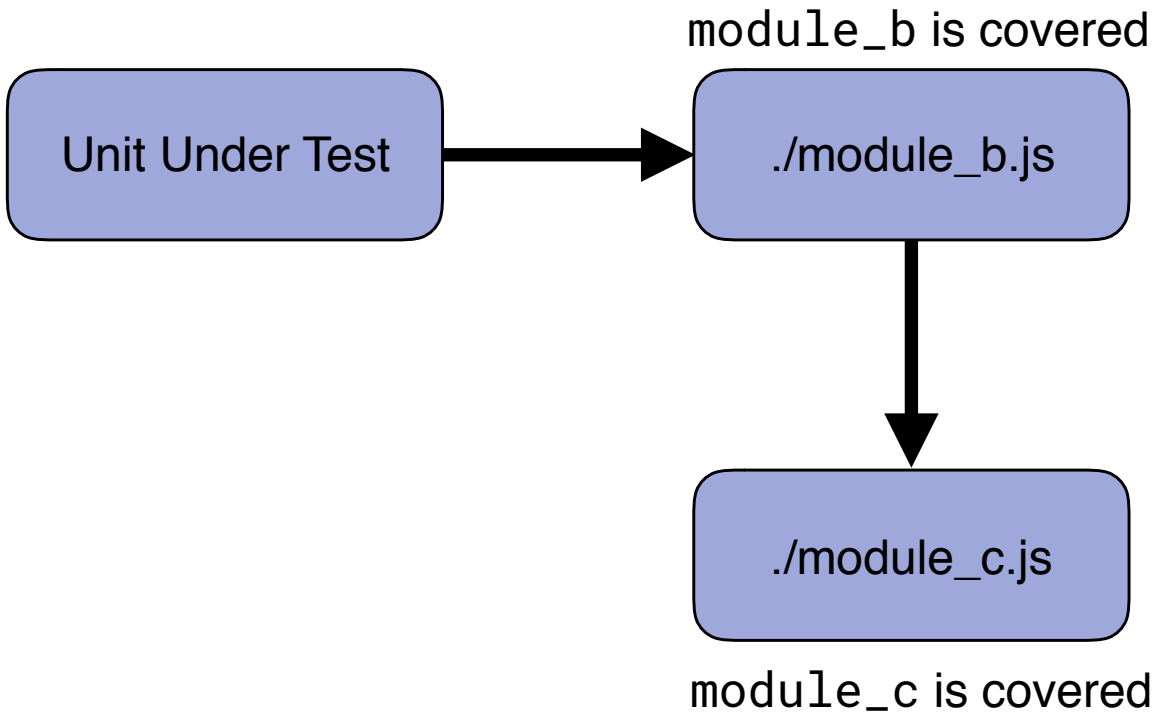
jsdom is a pure-JavaScript implementation of many web standards, notably the WHATWG DOM and HTML Standards, for use with Node.js. In general, the goal of the project is to emulate enough of a subset of a web browser to be useful for testing and scraping real-world web applications.

The latest versions of jsdom require Node.js v8 or newer. (Versions of jsdom below v12 still work with Node.js v6, but are unsupported.)

TEST ISOLATION

What do you mean by trusting browser APIs?

Trust that they work, but check that you are using them



TEST ISOLATION

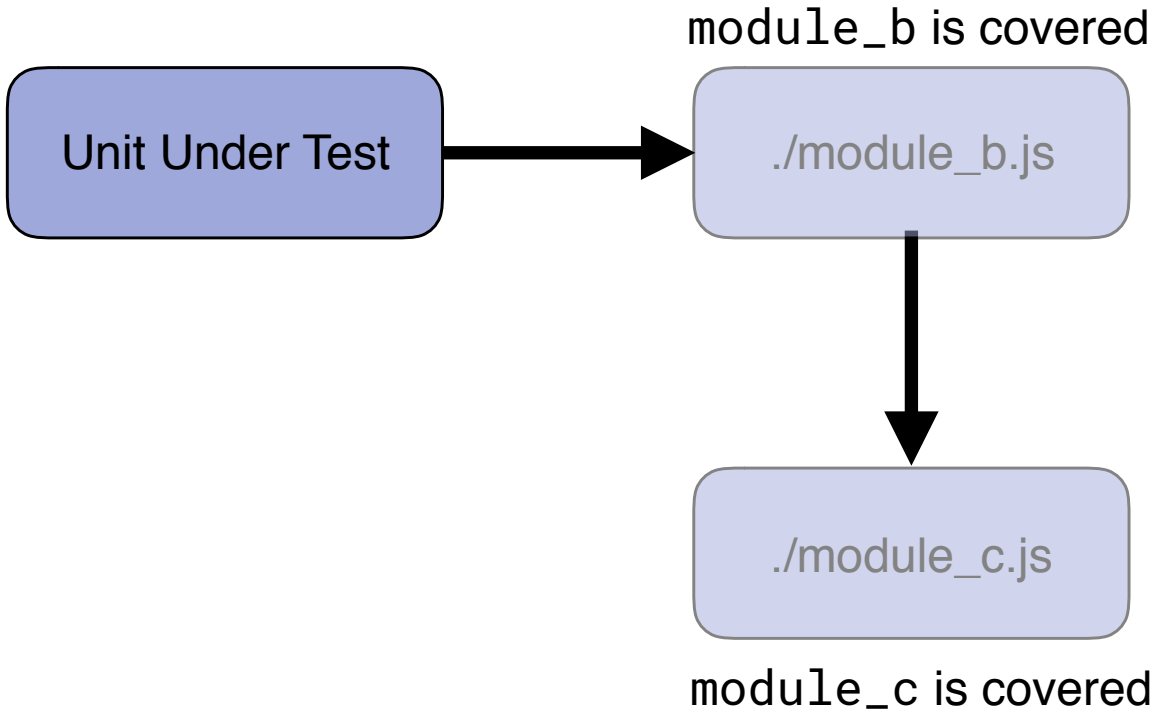
When should I mock?

Create transitive guarantees.

TEST ISOLATION

When should I mock?

mocking `module_b` helps you avoid redundant checks



Create transitive guarantees.

- Avoid brittleness.
- Avoid redundant checks.
- Avoid tests becoming too big.

If `module_c` is covered I don't need to check it in module_b's tests.

If UUT uses `module_b` and, transitively, `module_c`, I can mock `module_b`.

Sinon's Mocks, Stubs, and Spies	Jest's Mocks	mocking imports
		***proxyquire*** ***rewire*** ***rewiremock***
If you're not using jest	If you're already using jest	If you're not using jest
More custom behaviour	Easily mocking entire modules	Mocking on import level
Plugin integration	Simple and well documented API to assert on	Depends on being paired with Sinon or another mocking code
Sandboxes	Can automatically clear mocks	

TEST ISOLATION

How can I mock?

Mocking code in general.

TEST ISOLATION

How can I mock?

Mocking HTTP responses.

Mess with the requests yourself.

You have full control over what's happening.

Can get repetitive.

Reasonably annoying to set matchers for requests.

nock

Use a specific library.

Default behaviour is not always what's best or consistent.

Well defined API. No need for wrappers.

Can get a bit verbose if you need to mock uncommon features.

For your assertions:

chaijs/chai-http

visionmedia/supertest

Eliminating non-determinism

Why does determinism matters?

How to make non-deterministic tests deterministic?

DETERMINISM

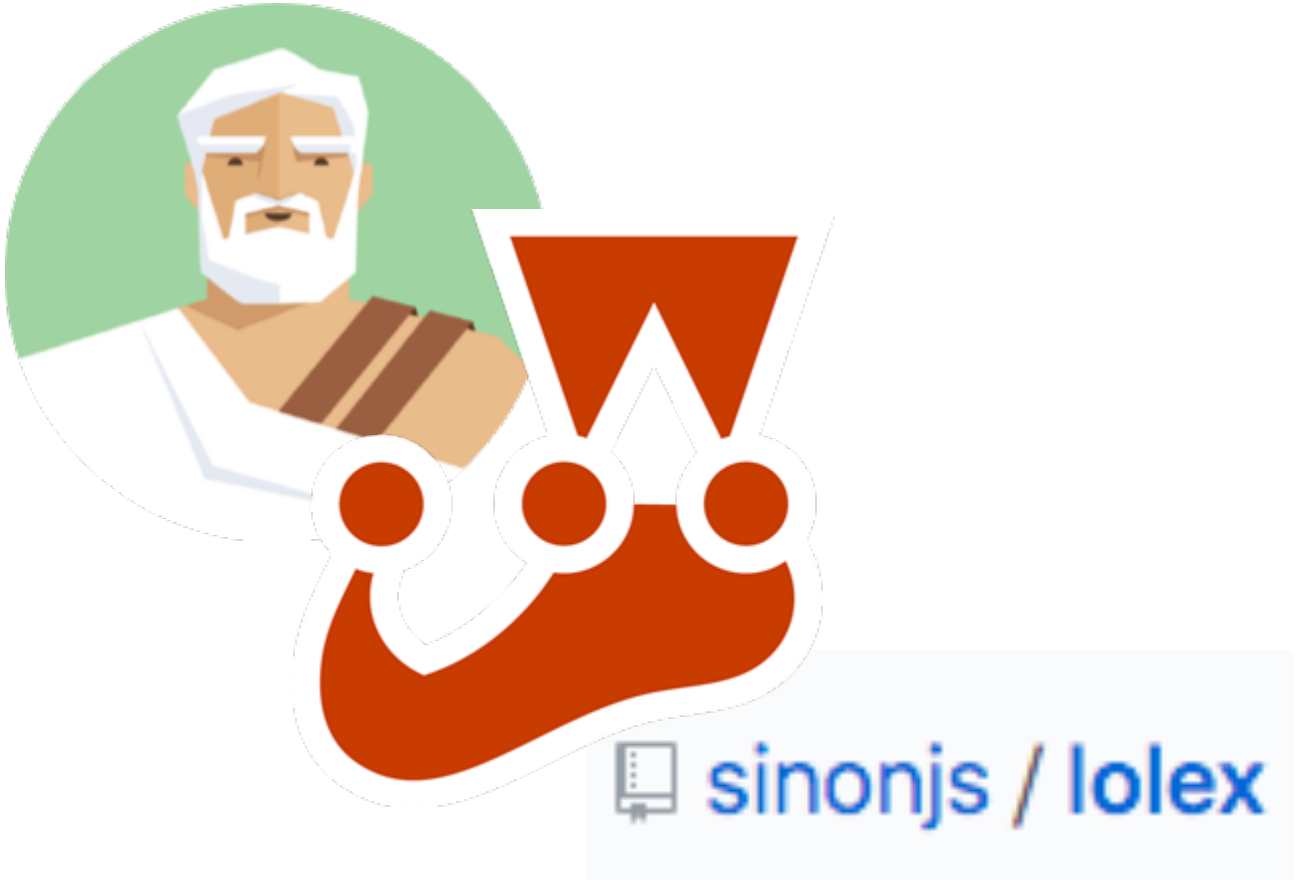
Why determinism matters?

Semantically speaking, flaky tests are the same as failing tests.

Flaky tests decrease the confidence in each build.

Is it a flaky test or is it flaky application code?

Approach 1: Mock-out non-deterministic pieces



DETERMINISM

How to solve non-determinism

Approach 2: Take variability into account

→ Use matchers

→ Use loose assertions willingly!
Allow broader sets of results.

This means you solve the problem on the testing side.

DETERMINISM

How to solve non-determinism

Approach 3: Make the code deterministic

Not always possible.

Ordering results within your tests.

Eliminating the usage of randomness.

Providing a deterministic state or seed.

DETERMINISM

How to solve non-determinism

Speeding-up test runs

Why does quick feedback matters?
How can I speed up test runs?
How are my tests scheduled?

QUICK FEEDBACK

Why does quick feedback matters?

If tests are slow, they won't get run frequently enough.

Quick feedback encourages you to take more gradual steps (proper TDD).

Quick feedback decreases frustration, creating a positive feedback loop.


2019
HOLYJS
MOSCOW
Good tests kill flawed theories;
we remain alive to guess again.
— POPPER, Karl

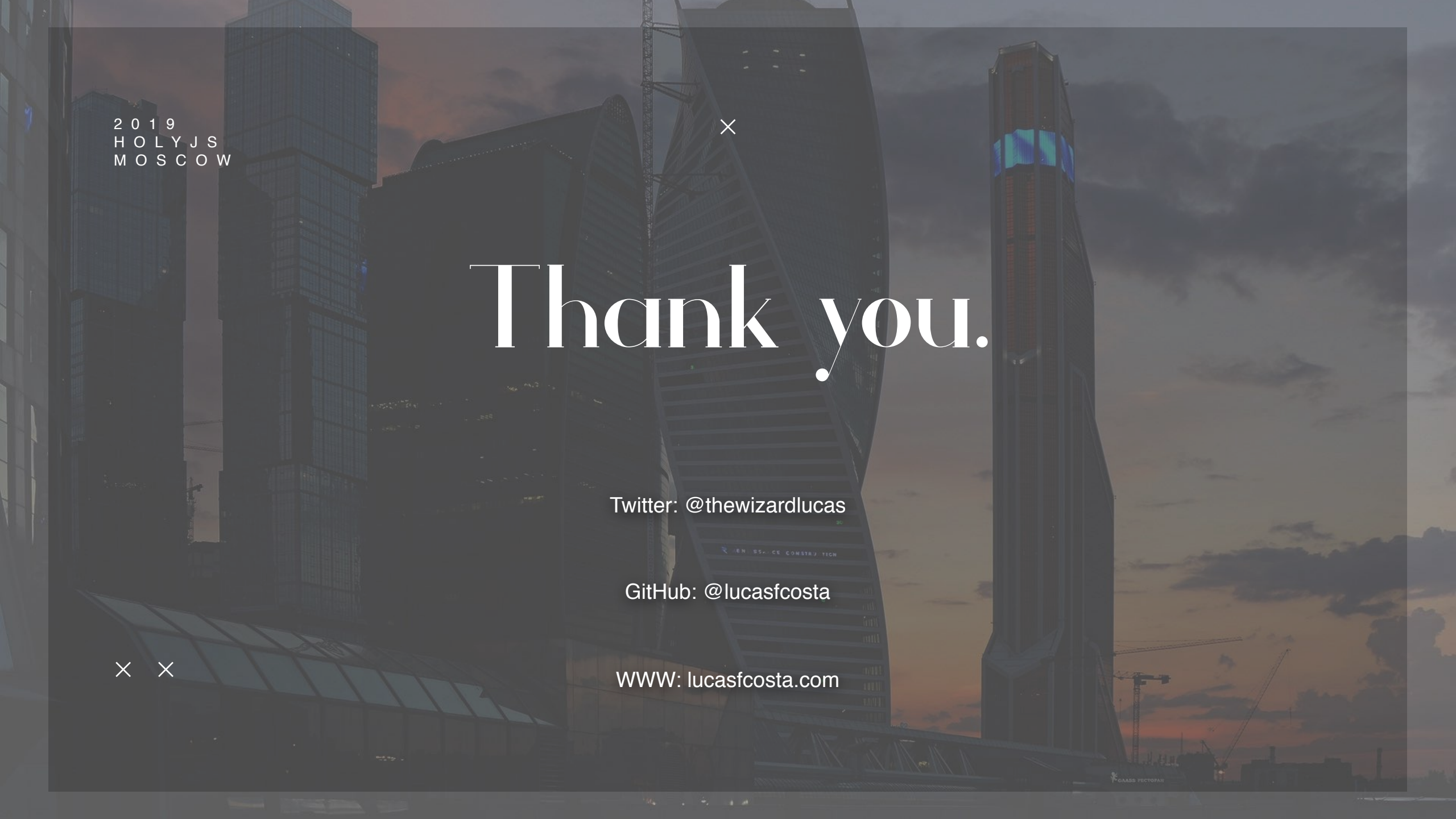
2019
HOLYJS
MOSCOW

Write code, read books

Twitter: @thewizardlucas

GitHub: @lucasfccosta

WWW: lucasfcosta.com



Thank you.

Twitter: @thewizardlucas

GitHub: @lucasfccosta

WWW: lucasfcosta.com