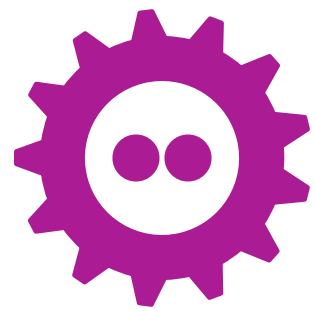
TESTING IN RUST

A PRIMER IN TESTING AND MOCKING

@donald_whyte



FOSDEM 2018

ABOUT ME

- Software Engineer @ **Engineers Gate**
- Real-time trading systems
- Scalable data infrastructure
- Python/C++/Rust developer

MOTIVATION

Rust focuses on memory safety.

While supporting advanced concurrency.

Does a great job at this.

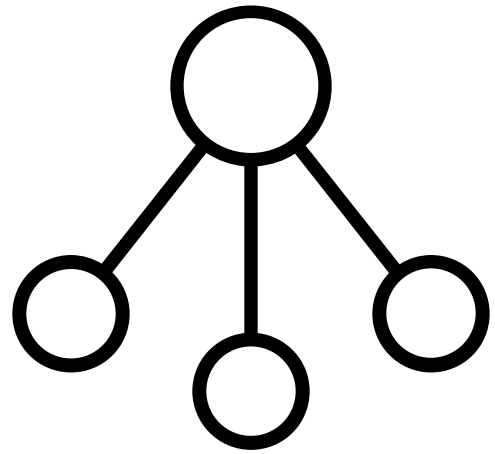
But even if our code is safe...

...we still need to make sure it's doing the **right** thing.

OUTLINE

- Rust unit tests
- Mocking in Rust using **`double`**
- Design considerations

1. UNIT TESTS



Create library: `cargo new`

```
cargo new some_lib
cd some_lib
```

Test fixture automatically generated:

```
> cat src/lib.rs
```

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        // test code in here
    }
}
```

Write unit tests for a module by defining a private **`tests`** module in its source file.

```
// production code
pub fn add_two(num: i32) -> i32 {
    num + 2
}

#[cfg(test)]
mod tests {
    // test code in here
}
```

Add isolated test functions to private **`tests`** module.

```
// ...prod code...

#[cfg(test)]
mod tests {
    use super::*;  // import production symbols from parent module

    #[test]
    fn ensure_two_is_added_to_negative() {
        assert_eq!(0, add_two(-2));
    }
    #[test]
    fn ensure_two_is_added_to_zero() {
        assert_eq!(2, add_two(0));
    }
    #[test]
    fn ensure_two_is_added_to_positive() {
        assert_eq!(3, add_two(1));
    }
}
```

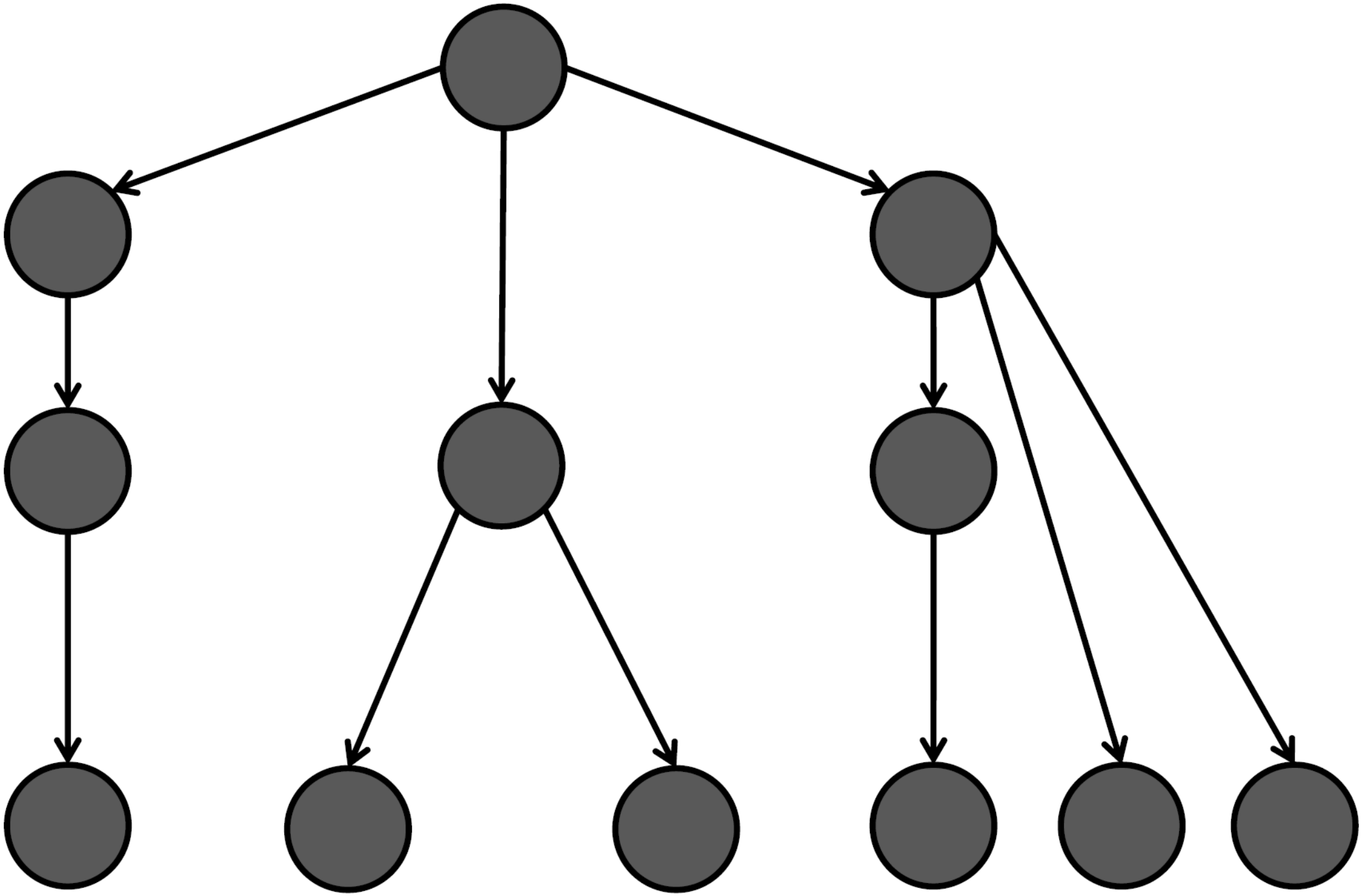
cargo test

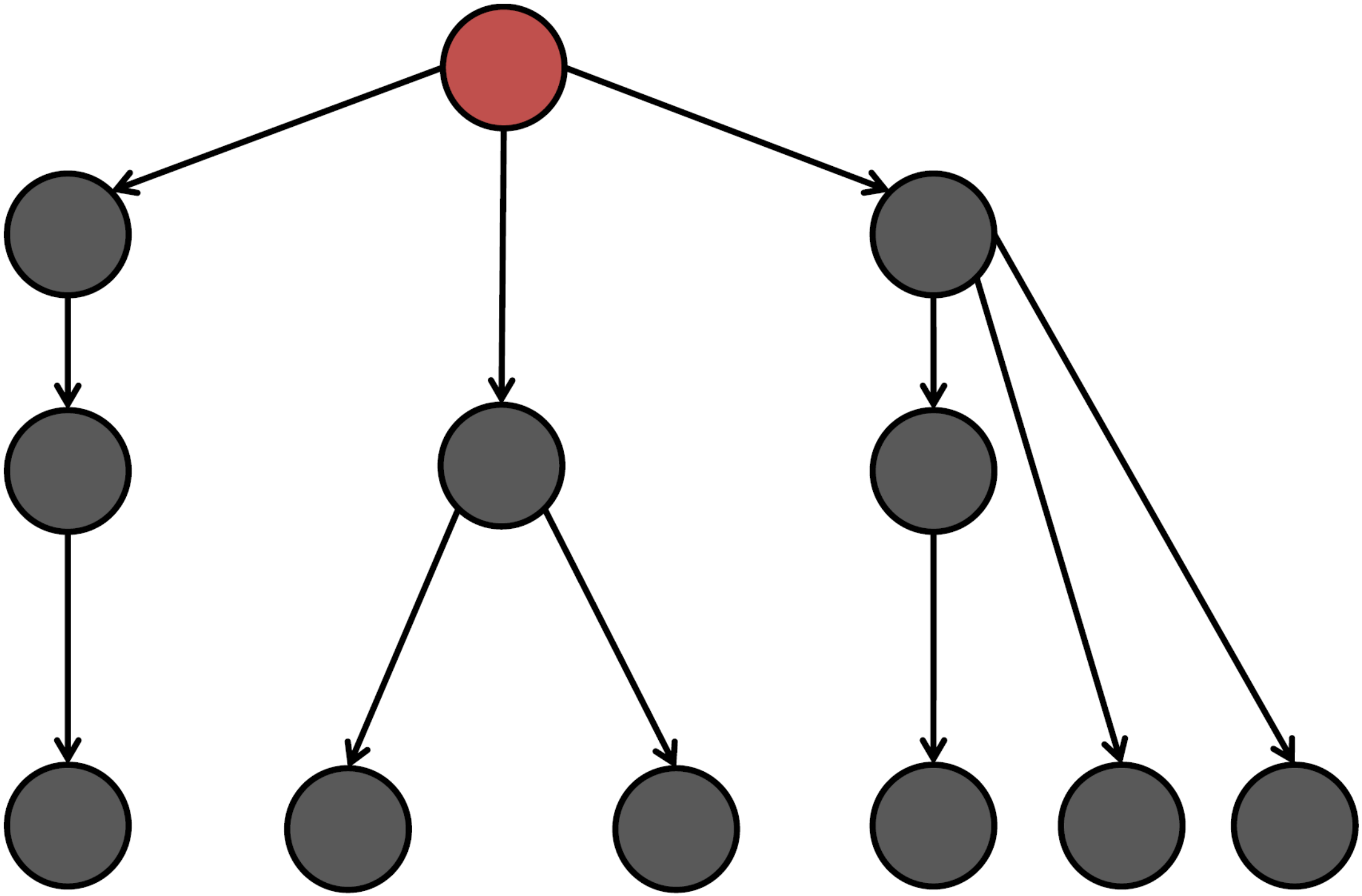
```
user:some_lib donaldwhyte$ cargo test
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/some_lib-4ea7f66796617175

running 3 tests
test tests::ensure_two_is_added_to_negative ... ok
test tests::ensure_two_is_added_to_positive ... ok
test tests::ensure_two_is_added_to_zero ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```

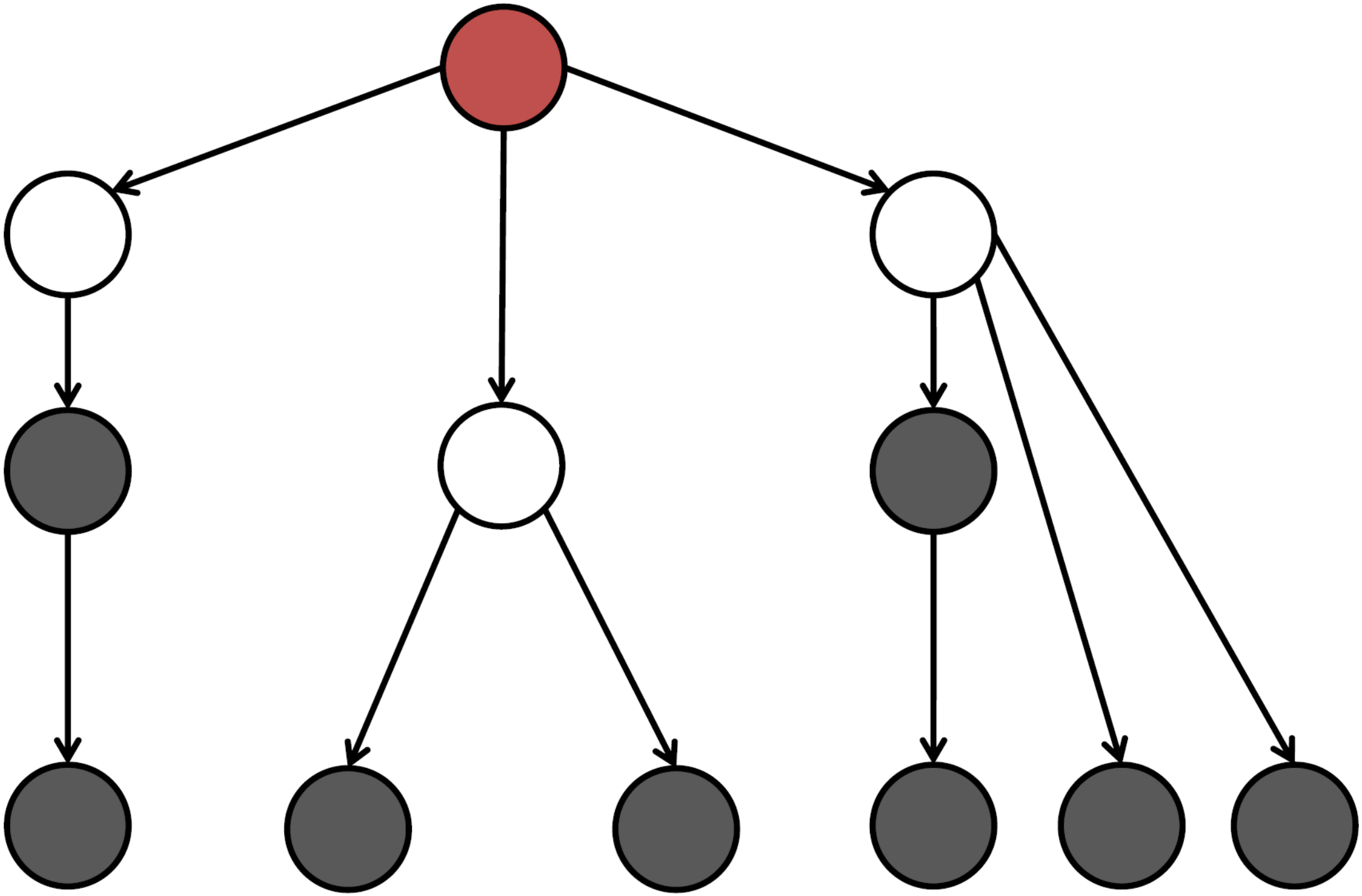
Rust has native support for:

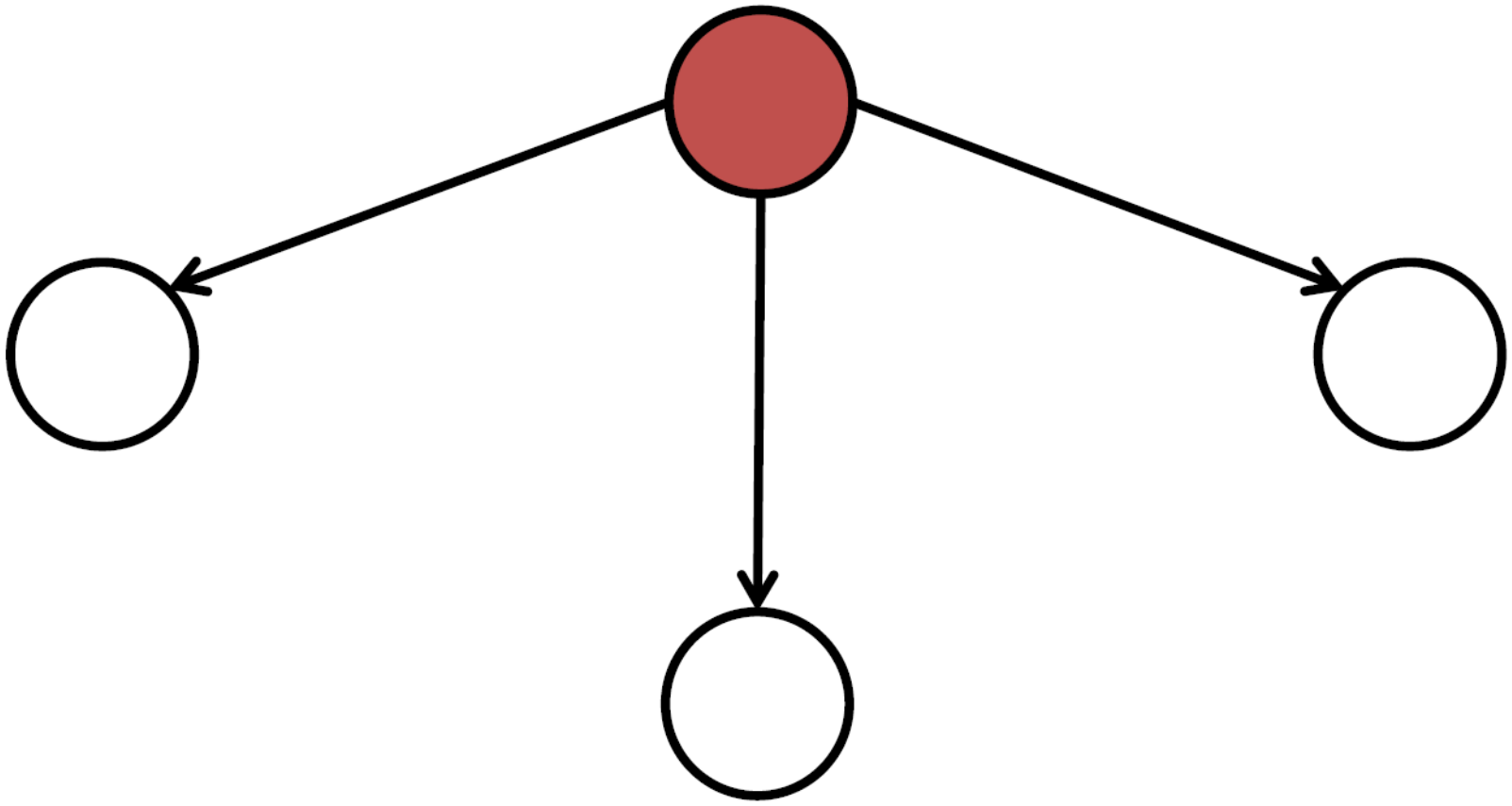
- documentation tests
- integration tests

2. WHAT IS MOCKING?









WHAT TO ELIMINATE

Anything non-deterministic that can't be reliably controlled within a unit test.

External data sources — files, databases

Network connections — services

External code dependencies — libraries

CAN ALSO ELIMINATE

Large internal dependencies for simpler tests.

SOLUTION: USE TEST DOUBLE

Term originates from a notion of a *"stunt double"* in films.

A **test double** is an object or function substituted for production code during testing.

Should behave in the same way as the production code.

Easier to control for testing purposes.

Many types of test double:

- Stub
- Spy
- Mock
- Fake

They're often all just referred to "mocks".

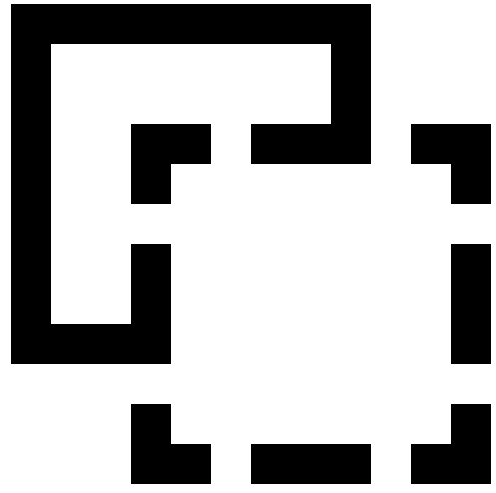
Spies are used in this talk.

SPIES PERFORM BEHAVIOUR VERIFICATION

Tests code by asserting its *interaction* with its *collaborators*.

3. TEST DOUBLES IN RUST

USING `DOUBLE`



`double` generates mock implementations for:

- **`trait`**s
- functions

Flexible configuration of a double's **behaviour**.

Simple and complex **assertions** on how mocks were used/called.

EXAMPLE

Predicting profit of a stock portfolio over time.

COLLABORATORS

```
pub trait ProfitModel {
    fn profit_at(&self, timestamp: u64) -> f64;
}
```

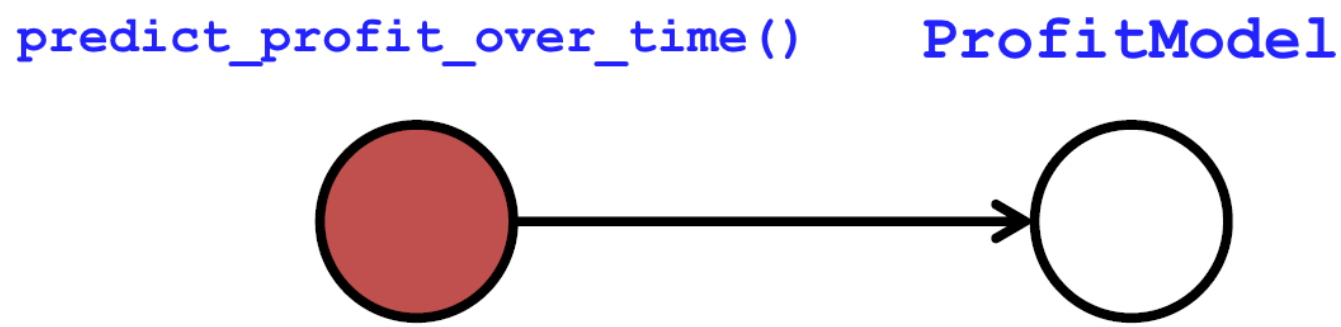
IMPLEMENTATION

```
pub fn predict_profit_over_time<M: ProfitModel>(
    model: &M,
    start: u64,
    end: u64) -> Vec<f64>
{
    (start..end + 1)
        .map(|t| model.profit_at(t))
        .collect()
}
```

We want to test **`predict_profit_over_time()`**.

Tests should be repeatable.

Not rely on an external environment.



One collaborator — **`ProfitModel`**.

PREDICTING PROFIT IS HARD

Real `ProfitModel` implementations use:

- external data sources (DBs, APIs, files)
- complex internal code dependencies (math models)

Let's mock `ProfitModel`.

`mock_trait!`

Generate mock **`struct`** that records interaction:

```
pub trait ProfitModel {
    fn profit_at(&self, timestamp: u64) -> f64;
}

mock_trait!(
    MockModel,
    profit_at(u64) -> f64);
```

mock_trait!

```
mock_trait!(
    NameOfMockStruct,
    method1_name(arg1_type, ...) -> return_type,
    method2_name(arg1_type, ...) -> return_type
    ...
    methodN_name(arg1_type, ...) -> return_type);
```

`mock_method!`

Generate implementations of all methods in mock **`struct`**.

```
mock_trait!(
    MockModel,
    profit_at(u64) -> f64);

impl ProfitModel for MockModel {
    mock_method!(profit_at(&self, timestamp: u64) -> f64);
}
```

mock_method!

```
impl TraitToMock for NameOfMockStruct {
  mock_method!(method1_name(&self, arg1_type, ...) -> return_type);
  mock_method!(method2_name(&self, arg1_type, ...) -> return_type);
  ...
  mock_method!(methodN_name(&self, arg1_type, ...) -> return_type);
}
```

Full code to generate a mock implementation of a **`trait`**:

```
mock_trait!(
    MockModel,
    profit_at(u64) -> f64);

impl ProfitModel for MockModel {
    mock_method!(profit_at(&self, timestamp: u64) -> f64);
}
```

USING GENERATED MOCKS IN TESTS

```
#[test]
fn test_profit_model_is_used_for_each_timestamp() {
  // GIVEN:
  let mock = MockModel::default();
  mock.profit_at.return_value(10);

  // WHEN:
  let profit_over_time = predict_profit_over_time(&mock, 0, 2);

  // THEN:
  assert_eq!(vec!(10, 10, 10), profit_over_time);
  assert_eq!(3, model.profit_at.num_calls());
}
```

GIVEN: SETTING MOCK BEHAVIOUR

DEFAULT RETURN VALUE

```rust
#[test]
fn no_return_value_specified() {
  // GIVEN:
  let mock = MockModel::default();

  // WHEN:
  let profit_over_time = predict_profit_over_time(&mock, 0, 2);

  // THEN:
  // default value of return type is used if no value is specified
  assert_eq!(vec!(0, 0, 0), profit_over_time);
}
```

ONE RETURN VALUE FOR ALL CALLS

```
#[test]
fn single_return_value() {
  // GIVEN:
  let mock = MockModel::default();
  mock.profit_at.return_value(10);

  // WHEN:
  let profit_over_time = predict_profit_over_time(&mock, 0, 2);

  // THEN:
  assert_eq!(vec!(10, 10, 10), profit_over_time);
}
```

SEQUENCE OF RETURN VALUES

```
#[test]
fn multiple_return_values() {
  // GIVEN:
  let mock = MockModel::default();
  mock.profit_at.return_values(1, 5, 10);

  // WHEN:
  let profit_over_time = predict_profit_over_time(&mock, 0, 2);

  // THEN:
  assert_eq!(vec!(1, 5, 10), profit_over_time);
}
```

RETURN VALUES FOR SPECIFIC ARGS

```
#[test]
fn return_value_for_specific_arguments() {
  // GIVEN:
  let mock = MockModel::default();
  mock.profit_at.return_value_for((1), 5);

  // WHEN:
  let profit_over_time = predict_profit_over_time(&mock, 0, 2);

  // THEN:
  assert_eq!(vec!(0, 5, 0), profit_over_time);
}
```

USE CLOSURE TO COMPUTE RETURN VALUE

```
#[test]
fn using_closure_to_compute_return_value() {
  // GIVEN:
  let mock = MockModel::default();
  mock.profit_at.use_closure(|t| t * 5 + 1);

  // WHEN:
  let profit_over_time = predict_profit_over_time(&mock, 0, 2);

  // THEN:
  assert_eq!(vec!(1, 6, 11), profit_over_time);
}
```

THEN: CODE USED MOCK AS EXPECTED

Verify mocks are called:

- the right number of times
- with the right arguments

ASSERT CALLS MADE

```rust
#[test]
fn asserting_mock_was_called() {
  // GIVEN:
  let mock = MockModel::default();

  // WHEN:
  let profit_over_time = predict_profit_over_time(&mock, 0, 2);

  // THEN:
  // Called at least once.
  assert!(mock.profit_at.called());
  // Called with argument 1 at least once.
  assert!(mock.profit_at.called_with((1)));
  // Called at least once with argument 1 and 0.
  assert!(mock.profit_at.has_calls((1), (0)));
}
```

TIGHTER CALL ASSERTIONS

```
#[test]
fn asserting_mock_was_called_with_precise_constraints() {
  // GIVEN:
  let mock = MockModel::default();

  // WHEN:
  let profit_over_time = predict_profit_over_time(&mock, 0, 2);

  // THEN:
  // Called exactly three times, with 1, 0 and 2.
  assert!(mock.profit_at.has_calls_exactly((1), (0), (2)));
  // Called exactly three times, with 0, 1 and 2 (in that order).
  assert!(mock.profit_at.has_calls_exactly_in_order(
      (0), (1), (2)
  ));
}
```

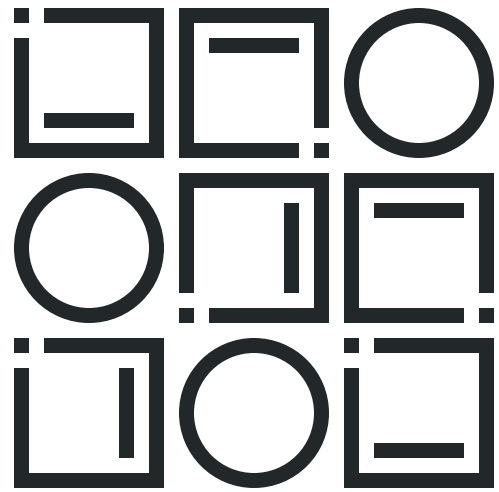
MOCKING FREE FUNCTIONS

Useful for testing code that takes function objects for runtime polymorphism.

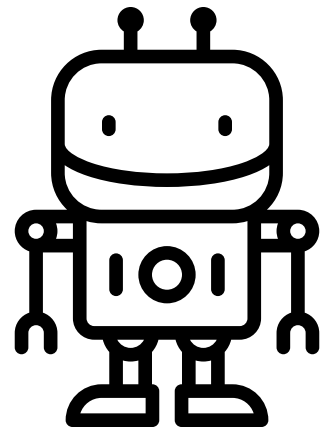
mock_func!

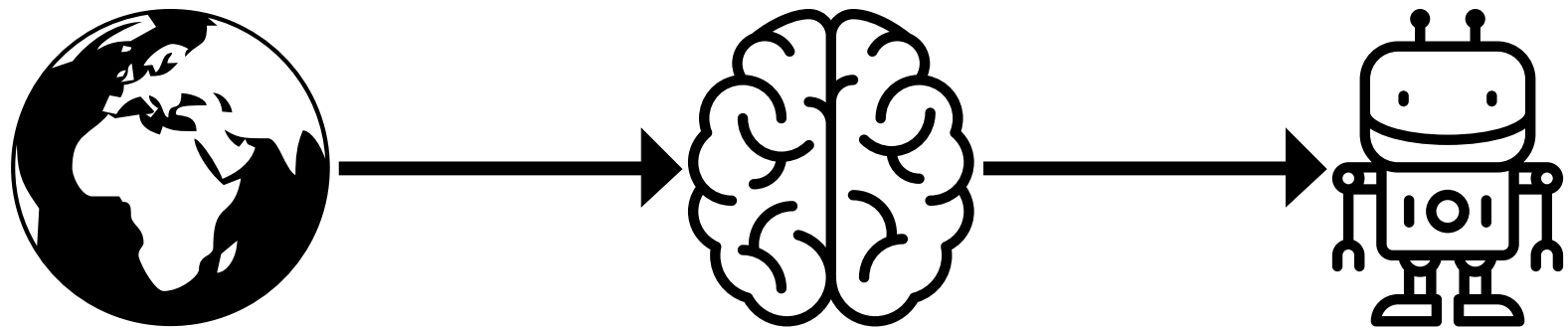
```
fn test_input_function_called_twice() {
    // GIVEN:
    mock_func!(mock,      // variable that stores mock object
               mock_fn,   // variable that stores closure
               i32,       // return value type
               i32);      // argument 1 type

    mock.return_value(10);

    // WHEN:
    code_that_calls_func_twice(&mock_fn);

    // THEN:
    assert_eq!(2, mock.num_calls());
    assert!(mock.called_with(42));
}
```

4. PATTERN MATCHING



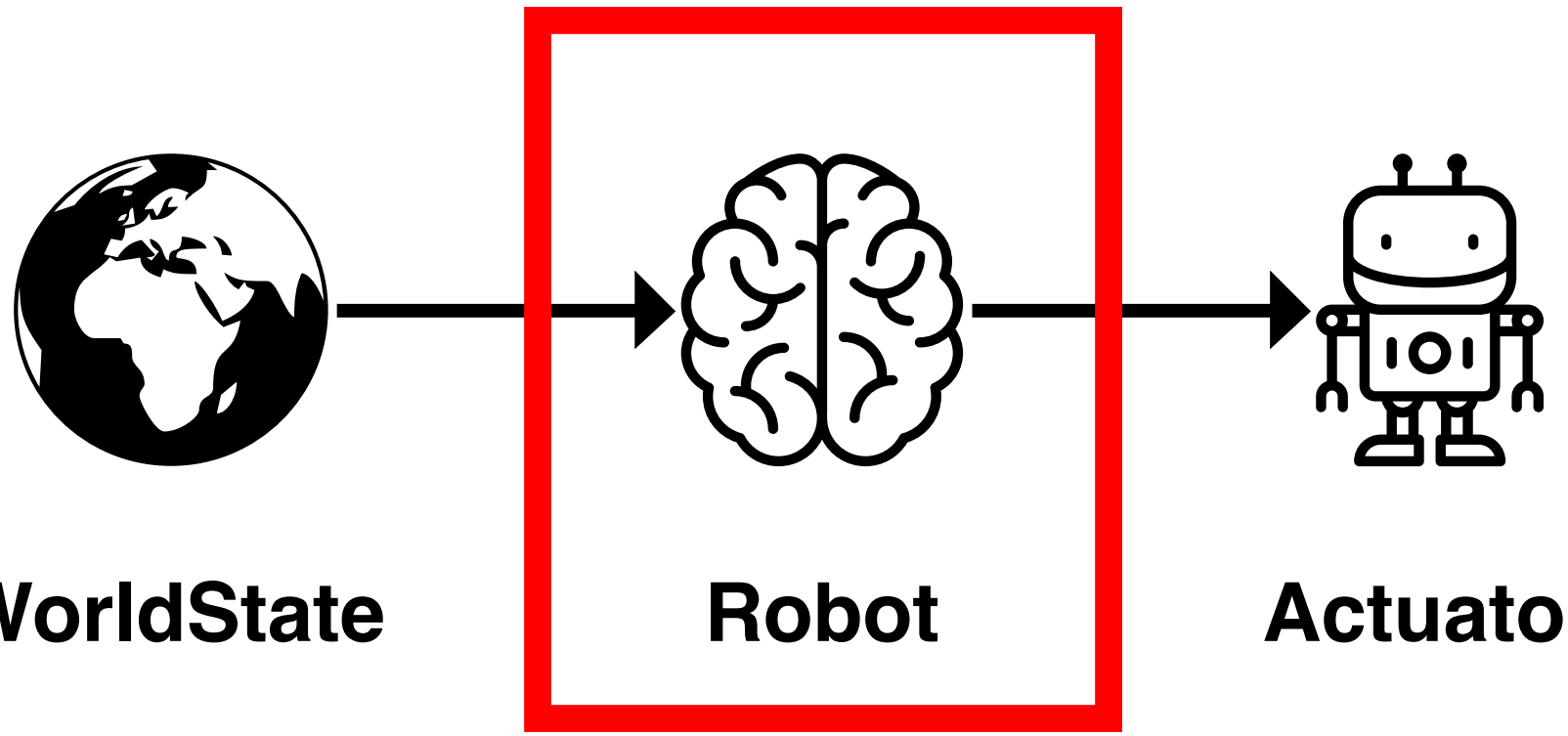
ROBOT DECISION MAKING





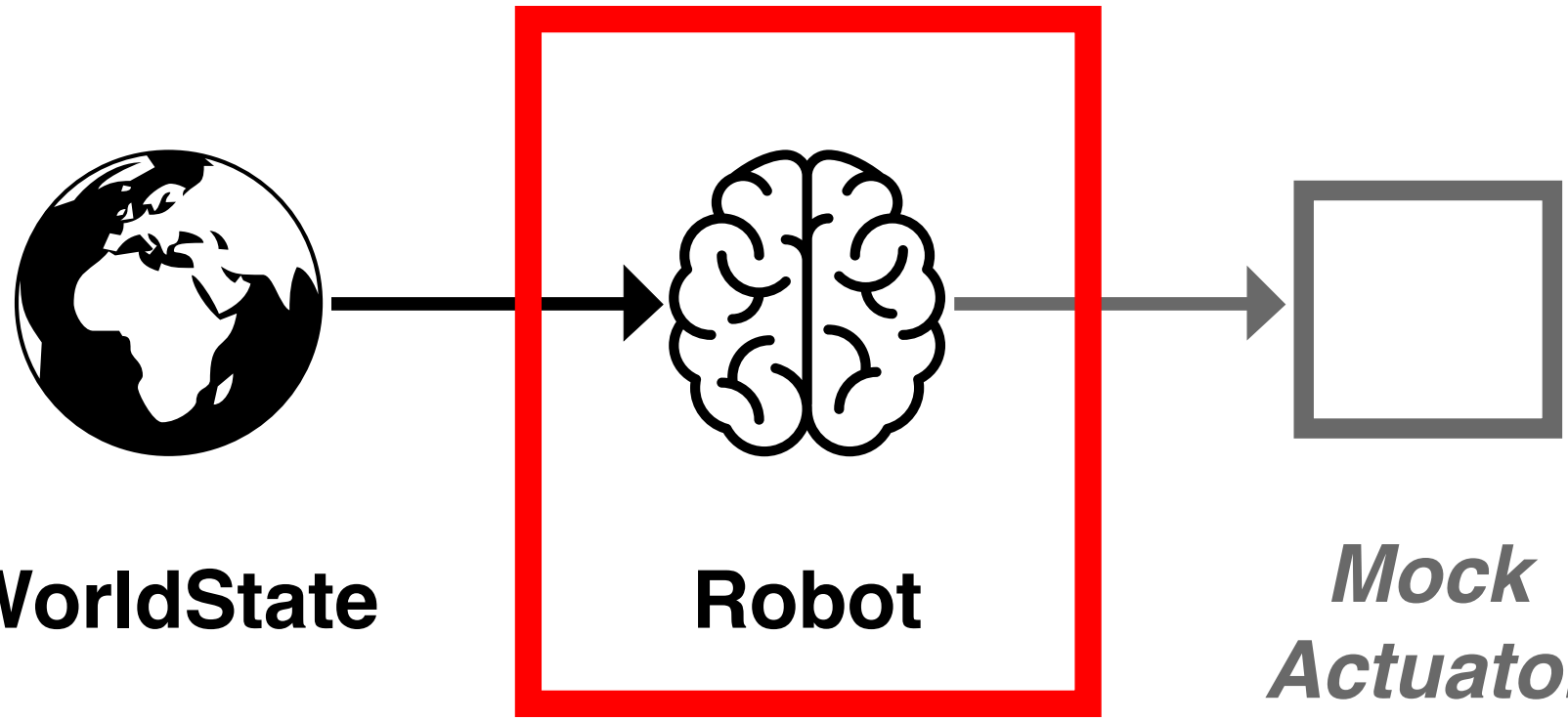
`WorldState`	Struct containing current world state
`Robot`	Processes state of the world and makes decisions on what do to next.
`Actuator`	Manipulates the world. Used by `Robot` to act on the decisions its made.

TEST THE ROBOT'S DECISIONS



WorldState **Robot** **Actuator**

TEST THE ROBOT'S DECISIONS



COLLABORATORS

```
pub trait Actuator {
    fn move_forward(&mut self, amount: i32);
    // ...
}
```

GENERATE MOCK COLLABORATORS

```
mock_trait!(
    MockActuator,
    move_forward(i32) -> ());

impl Actuator for MockActuator {
    mock_method!(move_forward(&mut self, amount: i32));
}
```

IMPLEMENTATION

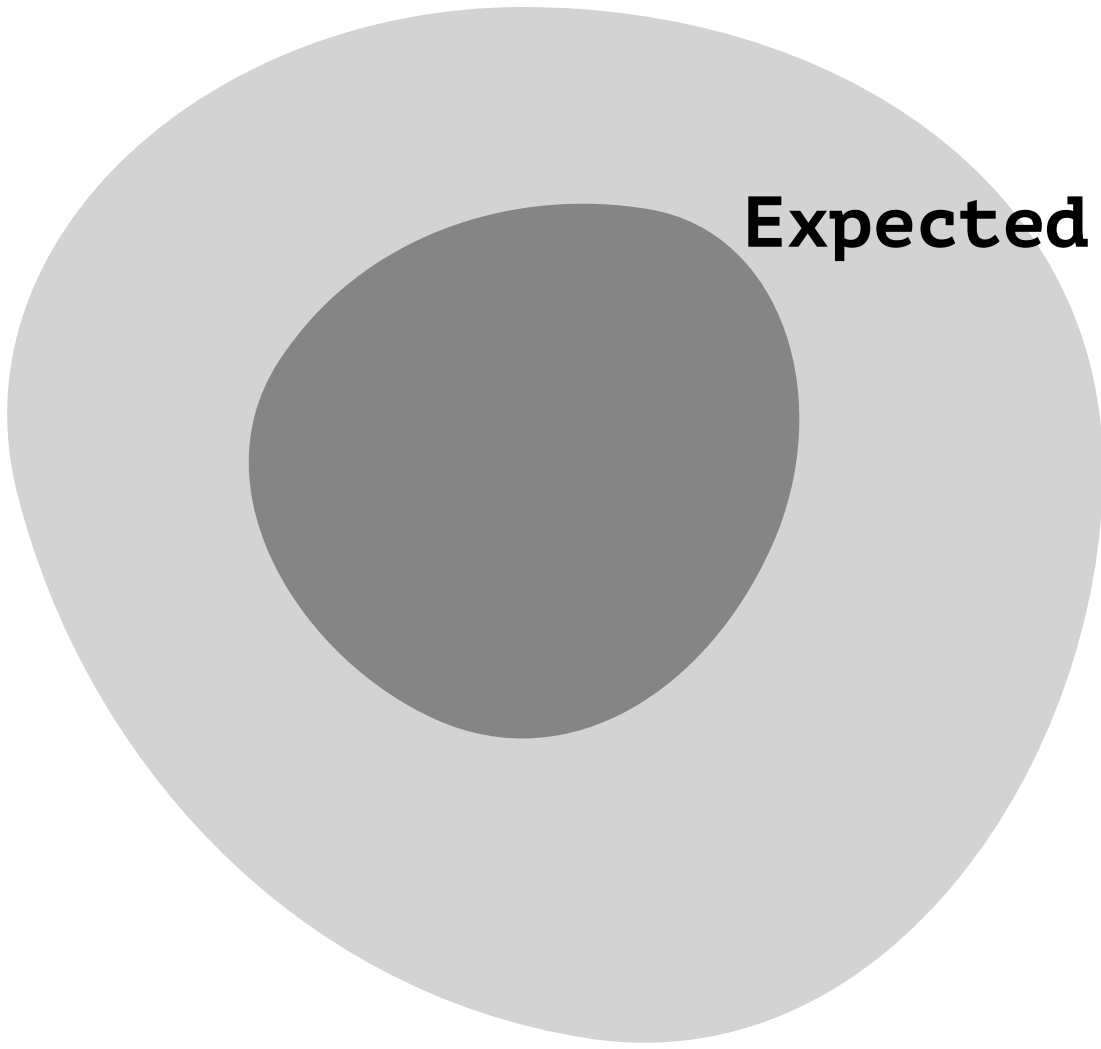
```
pub struct Robot<A> {
    actuator: &mut A
}

impl<A: Actuator> Robot {
    pub fn new(actuator: &mut A) -> Robot<A> {
        Robot { actuator: actuator }
    }

    pub fn take_action(&mut self, state: WorldState) {
        // Complex business logic that decides what actions
        // the robot should take.
        // This is what we want to test.
    }
  }
}
```

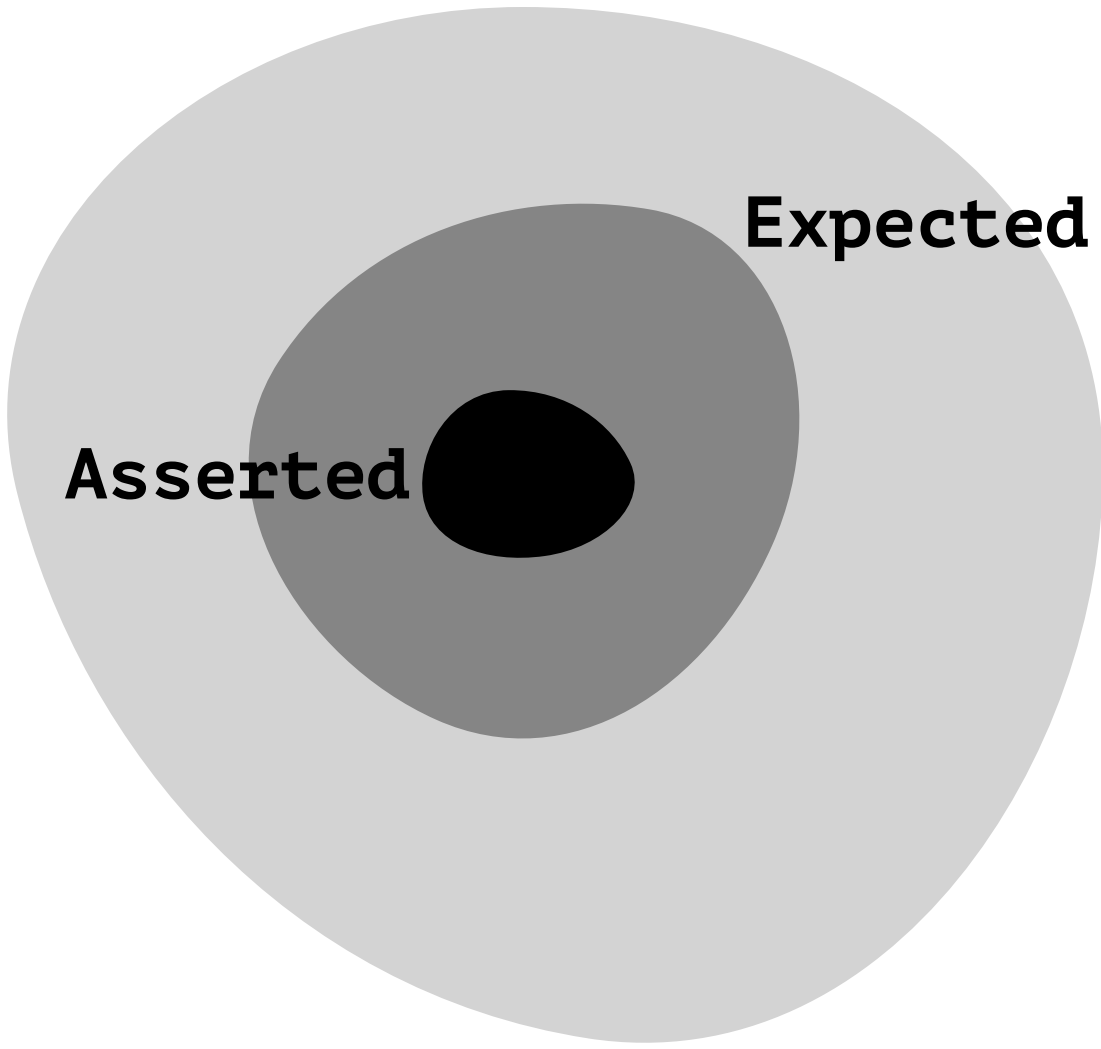
TESTING THE ROBOT

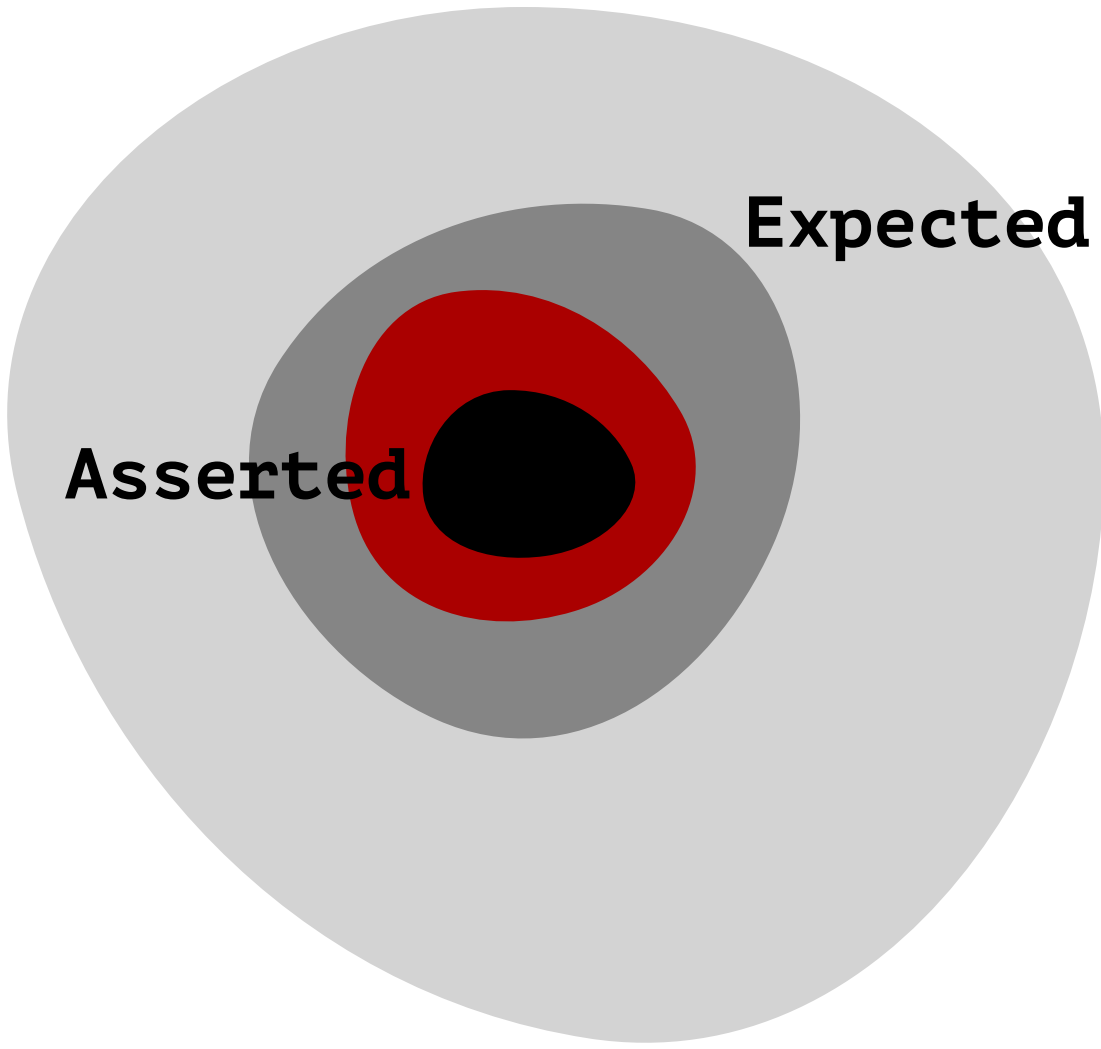
```
#[test]
fn test_the_robot() {
    // GIVEN:
    let input_state = WorldState { ... };
    let actuator = MockActuator::default();
    // WHEN:
    {
        let robot = Robot::new(&actuator);
        robot.take_action(input_state);
    }
    // THEN:
    assert!(actuator.move_forward.called_with(100));
}
```

Do we really care that the robot moved ***exactly*** 100 units?

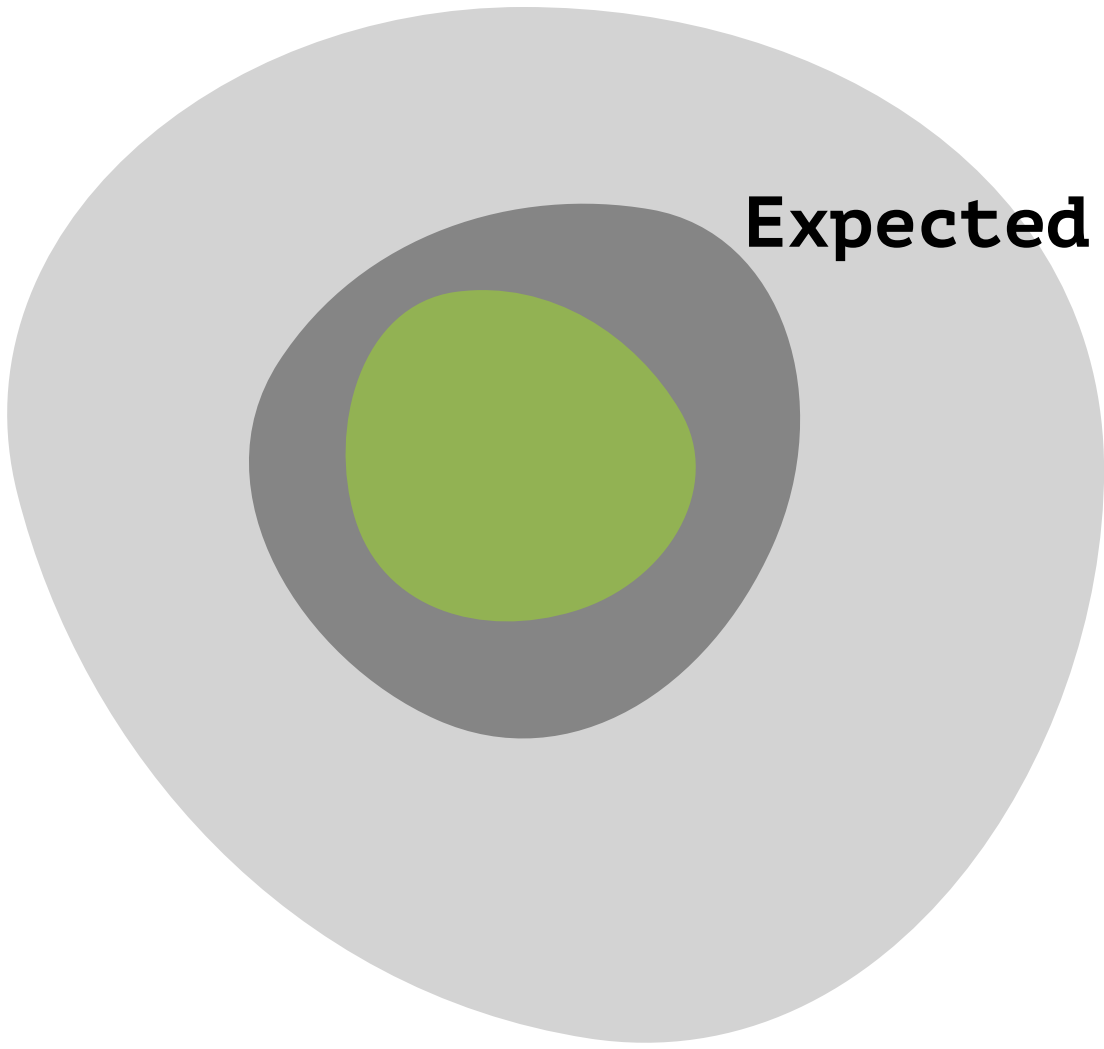
All Possible Behaviour


Expected
All Possible Behaviour


Expected
Asserted
All Possible Behaviour



Behaviour changes!


Expected + Asserted
All Possible Behaviour

Behaviour verification can **overfit** the implementation.

Lack of tooling makes this more likely.

PATTERN MATCHING TO THE RESCUE

Match argument values to patterns.

Not exact values.

Loosens test expectations, making them less brittle.

called_with_pattern()

```
#[test]
fn test_the_robot() {
    // GIVEN:
    let input_state = WorldState { ... };
    let actuator = MockActuator::default();
    // WHEN:
    {
        let robot = Robot::new(&actuator);
        robot.take_action(input_state);
    }
    // THEN:
    let is_greater_or_equal_to_100 = |arg: &i32| *arg >= 100;

    assert!(actuator.move_forward.called_with_pattern(
        is_greater_than_or_equal_to_100
    ));
}
```

Parametrised matcher functions:

```
/// Matcher that matches if `arg` is greater than or
/// equal to `base_val`.
pub fn ge<T: PartialEq + PartialOrd>(
    arg: &T,
    base_val: T) -> bool
{
    *arg >= base_val
}
```

Use `p!` to generate matcher closures on-the-fly.

```
use double::matcher::ge;

let is_greater_or_equal_to_100 = p!(ge, 100);
```

```
use double::matcher::*;

#[test]
fn test_the_robot() {
    // GIVEN:
    let input_state = WorldState { ... };
    let actuator = MockActuator::default();
    // WHEN:
    {
        let robot = Robot::new(&actuator);
        robot.take_action(input_state);
    }
    // THEN:
    assert!(actuator.move_forward.called_with_pattern(
        p!(ge, 100)
    ));
```

BUILT-IN MATCHERS

WILDCARD

`any()` argument can be any value of the correct type

COMPARISON MATCHERS

`eq(value)`	`argument == value`
`ne(value)`	`argument != value`
`lt(value)`	`argument < value`
`le(value)`	`argument <= value`
`gt(value)`	`argument > value`
`ge(value)`	`argument >= value`
`is_some(matcher)`	arg is `Option::Some`, whose contents matches `matcher`
`is_ok(matcher)`	arg is `Result::Ok`, whose contents matches `matcher`
`is_err(matcher)`	arg is `Result::er`, whose contents matches `matcher`

FLOATING-POINT MATCHERS

`f32_eq(value)`	argument is a value approximately equal to the **`f32 value`**, treating two NaNs as unequal.
`f64_eq(value)`	argument is a value approximately equal to the **`f64 value`**, treating two NaNs as unequal.
`nan_sensitive_f32_eq(value)`	argument is a value approximately equal to the **`f32 value`**, treating two NaNs as equal.
`nan_sensitive_f64_eq(value)`	argument is a value approximately equal to the **`f64 value`**, treating two NaNs as equal.

STRING MATCHERS

`has_substr(string)`	argument contains **`string`** as a sub-string.
`starts_with(prefix)`	argument starts with string **`prefix`**.
`ends_with(suffix)`	argument ends with string **`suffix`**.
`eq_nocase(string)`	argument is equal to **`string`**, ignoring case.
`ne_nocase(value)`	argument is not equal to **`string`**, ignoring case.

CONTAINER MATCHERS

`is_empty`	argument implements `IntoIterator` and contains no elements.
`has_length(size_matcher)`	argument implements `IntoIterator` whose element count matches `size_matcher`.
`contains(elem_matcher)`	argument implements `IntoIterator` and contains at least one element that matches `elem_matcher`.
`each(elem_matcher)`	argument implements `IntoIterator` and all of its elements match `elem_matcher`.
`unordered_elements_are(elements)`	argument implements `IntoIterator` that contains the same elements as the vector `elements` (ignoring order).
`when_sorted(elements)`	argument implements `IntoIterator` that, when its elements are sorted, matches the vector `elements`.

COMPOSITE MATCHERS

Assert that a single arg should match many patterns.

```
// Assert robot moved between 100 and 200 units.
assert!(robot.move_forward.called_with_pattern(
    p!(all_of, vec!(
        p!(ge, 100),
        p!(le, 200)
    ))
));
```

COMPOSITE MATCHERS

Assert all elements of a collection match a pattern:

```
let mock = MockNumberRecorder::default();

mock.record_numbers(vec!(42, 100, -49395, 502));

// Check all elements in passed in vector are non-zero.
assert!(mock.record_numbers.called_with_pattern(
    p!(each, p!(ne, 0))
));
```

CUSTOM MATCHERS

Define new matchers if the built-in ones aren't enough.

```
fn custom_matcher<T>(arg: &T, params...) -> bool {
    // matching code here
}
```

5. DESIGN CONSIDERATIONS

2 design goals in **`double`**.

1. RUST STABLE FIRST

2. NO CHANGES TO PRODUCTION CODE REQUIRED

Allows `traits` from the standard library or external crates to be mocked.

CHALLENGING

Meeting these goals is difficult, because Rust:

- is a compiled/statically typed language
- runs a borrow checker

Most mocking libraries require nightly.

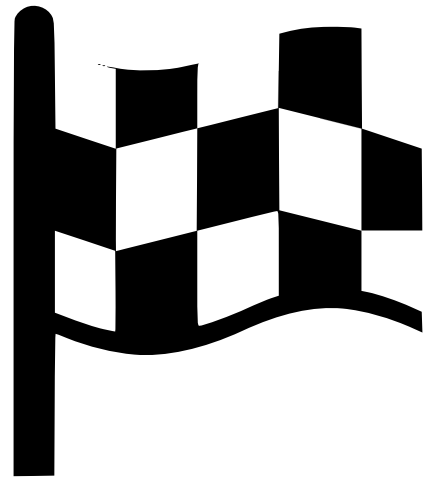
Most (all?) mocking libraries require prod code changes.

THE COST

`double` achieves the two goals at a cost.

Longer mock definitions.

FIN



Mocking is used to isolate unit tests from exernal resources or complex dependencies.

Achieved in Rust by replacing `trait`s and functions.

Behaviour verification can overfit implementation.

Pattern matching **expands asserted behaviour space** to reduce overfitting.

`double` is a crate for generating **`trait`**/function mocks.

Wide array of behaviour setups and call assertions.

First-class pattern matching support.

Requires no changes to production code.

ALTERNATIVE MOCKING LIBRARIES

- mockers
- mock_derive
- galvanic-mock
- mocktopus

LINKS

- these slides:
  - http://donsoft.io/mocking-in-rust-using-double
- double repository:
  - https://github.com/DonaldWhyte/double
- double documentation:
  - https://docs.rs/double/0.2.2/double/
- example code from this talk:
  - https://github.com/DonaldWhyte/mocking-in-rust-using-double/tree/master/code

GET IN TOUCH

don@donsoft.io
@donald_whyte
https://github.com/DonaldWhyte

APPENDIX

IMAGE CREDITS

- Gregor Cresnar
- Zurb
- Freepik
- Dave Gandy
- Online Web Fonts