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:

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

```rust
#[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.

```rust
// ...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.
Tests code by asserting its *interaction* with its *collaborators*. 
3. TEST DOUBLES IN RUST

USING DOUBLE
**double** generates mock implementations for:

- **traits**
- **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

```rust
pub trait ProfitModel {
    fn profit_at(&self, timestamp: u64) -> f64;
}
```
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.
Generate mock `struct` that records interaction:

```rust
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.

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

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

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

impl ProfitModel for MockModel {
    mock_method!(profit_at(&self, timestamp: u64) -> f64);
}
```
# Using generated mocks in tests

```rust
#[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);
}
```
# [test]

```rust
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);
}
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);
}
# [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
#[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)));
}
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.
```rust
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
<table>
<thead>
<tr>
<th>WorldState</th>
<th>Struct containing current world state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robot</td>
<td>Processes state of the world and makes decisions on what do to next.</td>
</tr>
<tr>
<td>Actuator</td>
<td>Manipulates the world. Used by Robot to act on the decisions its made.</td>
</tr>
</tbody>
</table>
TEST THE ROBOT'S DECISIONS
TEST THE ROBOT'S DECISIONS

WorldState → Robot → Mock Actuator
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));
}
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.
    }
}
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
All Possible Behaviour

Asserted

Expected
All Possible Behaviour

Expected

Asserted

Behaviour changes!
All Possible Behaviour

Expected + Asserted
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()
Parametrised matcher functions:

```rust
/// 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.

```rust
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

<table>
<thead>
<tr>
<th>Matcher</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq(value)</td>
<td>argument == value</td>
</tr>
<tr>
<td>ne(value)</td>
<td>argument != value</td>
</tr>
<tr>
<td>lt(value)</td>
<td>argument &lt; value</td>
</tr>
<tr>
<td>le(value)</td>
<td>argument &lt;= value</td>
</tr>
<tr>
<td>gt(value)</td>
<td>argument &gt; value</td>
</tr>
<tr>
<td>ge(value)</td>
<td>argument &gt;= value</td>
</tr>
<tr>
<td>is_some(matcher)</td>
<td>arg is Option::Some, whose contents matches matcher</td>
</tr>
<tr>
<td>is_ok(matcher)</td>
<td>arg is Result::Ok, whose contents matches matcher</td>
</tr>
<tr>
<td>is_err(matcher)</td>
<td>arg is Result::Err, whose contents matches matcher</td>
</tr>
</tbody>
</table>
## FLOATING-POINT MATCHERS

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f32_eq(value)</code></td>
<td>Argument is a value approximately equal to the <strong>f32 value</strong>, treating two NaNs as unequal.</td>
</tr>
<tr>
<td><code>f64_eq(value)</code></td>
<td>Argument is a value approximately equal to the <strong>f64 value</strong>, treating two NaNs as unequal.</td>
</tr>
<tr>
<td><code>nan_sensitive_f32_eq(value)</code></td>
<td>Argument is a value approximately equal to the <strong>f32 value</strong>, treating two NaNs as equal.</td>
</tr>
<tr>
<td><code>nan_sensitive_f64_eq(value)</code></td>
<td>Argument is a value approximately equal to the <strong>f64 value</strong>, treating two NaNs as equal.</td>
</tr>
</tbody>
</table>
## STRING MATCHERS

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>has_substr(string)</code></td>
<td>argument contains <code>string</code> as a sub-string.</td>
</tr>
<tr>
<td><code>starts_with(prefix)</code></td>
<td>argument starts with string <code>prefix</code>.</td>
</tr>
<tr>
<td><code>ends_with(suffix)</code></td>
<td>argument ends with string <code>suffix</code>.</td>
</tr>
<tr>
<td><code>eq_nocase(string)</code></td>
<td>argument is equal to <code>string</code>, ignoring case.</td>
</tr>
<tr>
<td><code>ne_nocase(value)</code></td>
<td>argument is not equal to <code>string</code>, ignoring case.</td>
</tr>
</tbody>
</table>
## CONTAINER MATCHERS

<table>
<thead>
<tr>
<th>Matcher</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_empty</code></td>
<td>argument implements <code>IntoIterator</code> and contains no elements.</td>
</tr>
<tr>
<td><code>has_length(size_matcher)</code></td>
<td>argument implements <code>IntoIterator</code> whose element count matches <code>size_matcher</code>.</td>
</tr>
<tr>
<td><code>contains(elem_matcher)</code></td>
<td>argument implements <code>IntoIterator</code> and contains at least one element that matches <code>elem_matcher</code>.</td>
</tr>
<tr>
<td><code>each(elem_matcher)</code></td>
<td>argument implements <code>IntoIterator</code> and all of its elements match <code>elem_matcher</code>.</td>
</tr>
<tr>
<td><code>unordered_elements_are(elements)</code></td>
<td>argument implements <code>IntoIterator</code> that contains the same elements as the vector <code>elements</code> (ignoring order).</td>
</tr>
<tr>
<td><code>when_sorted(elements)</code></td>
<td>argument implements <code>IntoIterator</code> that, when its elements are sorted, matches the vector <code>elements</code>.</td>
</tr>
</tbody>
</table>
COMPOSITE MATCHERS

Assert that a single arg should match many patterns.

```plaintext
// 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:

```rust
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.

```rust
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 external resources or complex dependencies.

Achieved in Rust by replacing traits 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:
GET IN TOUCH

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

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