Introduction to Software Testing

Kevin Burleigh

The Testing Mindset

Why do we test?

Because we want **confidence** that our code **works** as expected.

How much confidence?

- cost vs benefit vs risk

Works in which ways?

- functional behaviors:
  - Do these inputs give the correct outputs?
- non-functional behaviors:
  - Does the system use too much memory?
  - Is the system responsive enough?

When do we test?

When our code changes

- new requirements
- refactoring
- pretty much *any* development activity

When our code's dependencies change

- classes
- functions
- data
- types
- libraries
- language
- compiler
- toolset
- operating system
- external services

When required by some external organization or regulation

- FDA, FAA, etc.
- (not the focus of this talk)

How do we test?

- Manual testing
  - `print` statements
  - debugger
  - using the application
- Automated test suites
  (the focus of this talk)

What do we test?

The whole system?

"acceptance" testing
often very time-consuming
difficult to automate
very brittle
combinatorial explosion

A large chunk?

"integration" testing
difficult to set up
brittleness still an issue

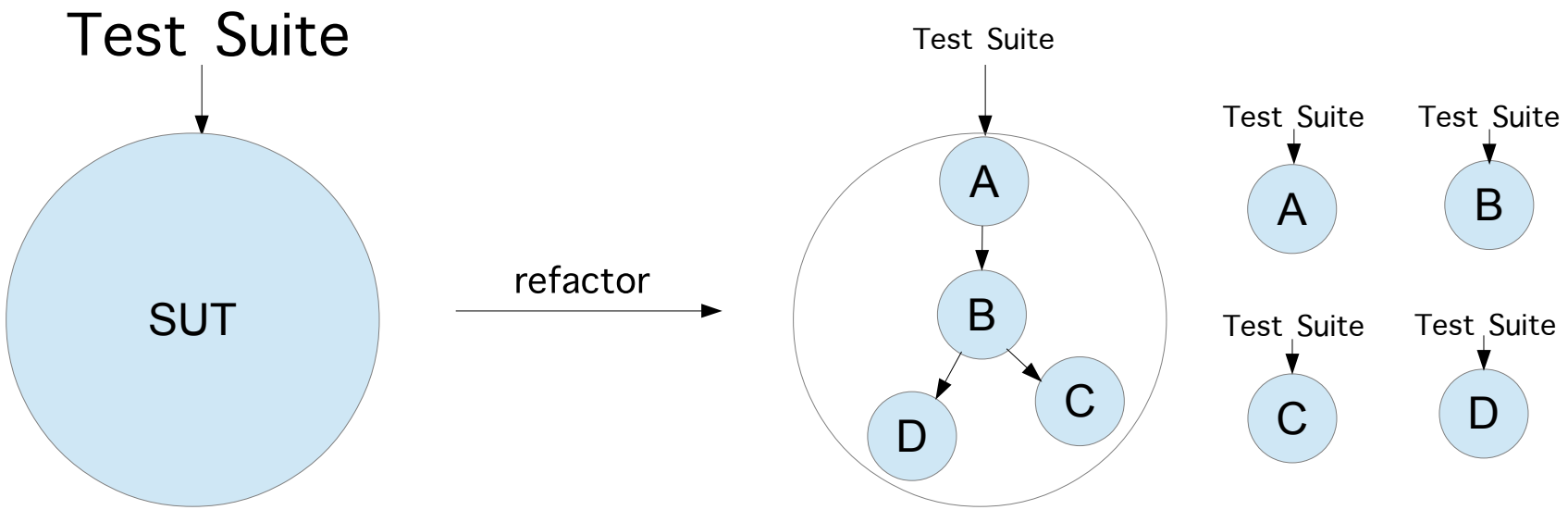
A tiny piece?

"unit" testing
very fast
easy to automate
tests can pass, but system still crashes

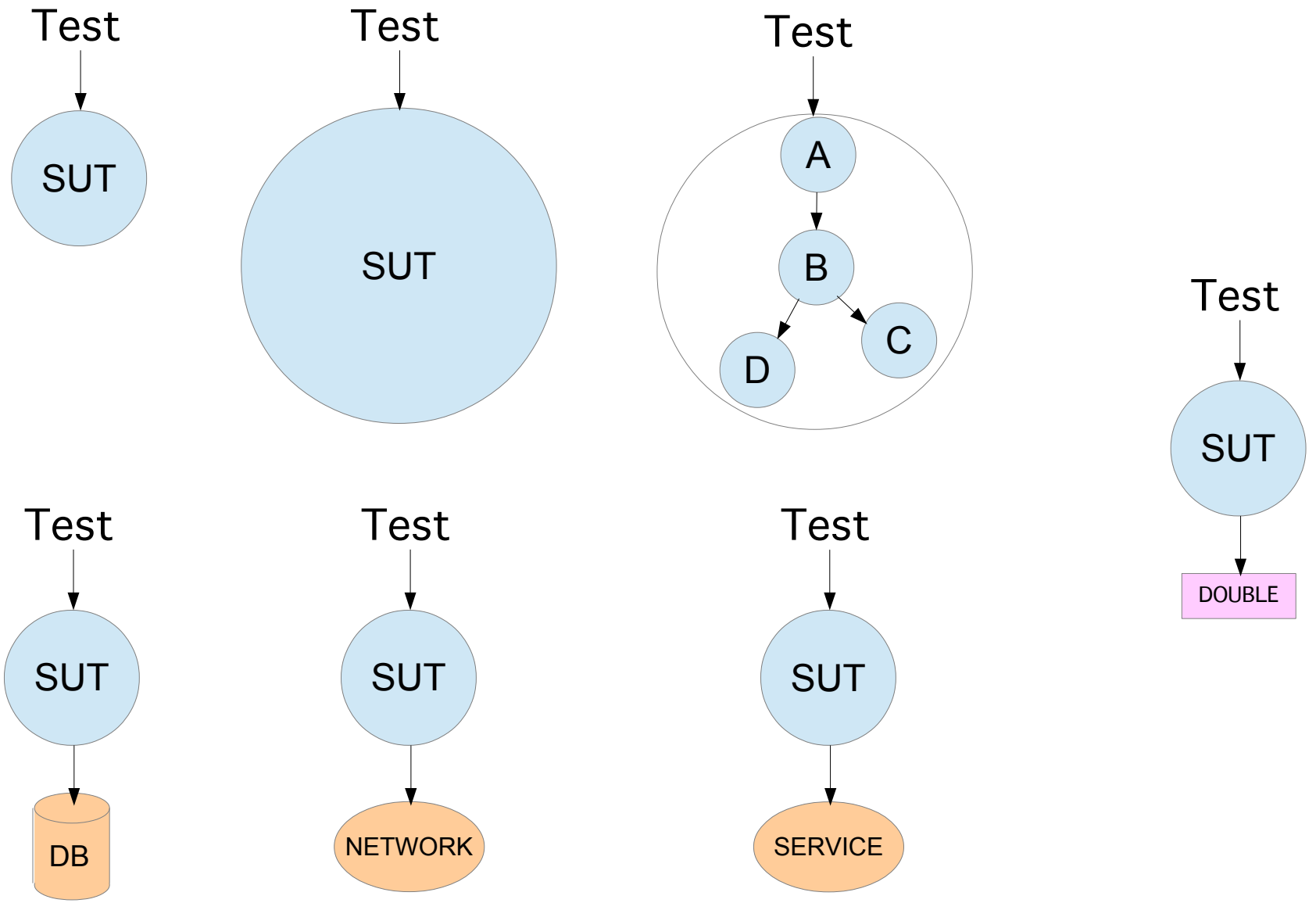
What do we test? (cont)

Don't obsess over which type of test you're writing!

- "Unit" tests turn into "integration" tests all the time!

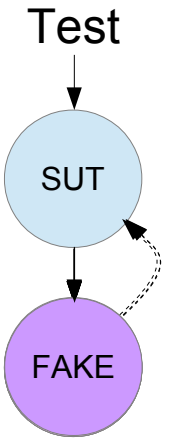


What do we test? (cont)



Types of test doubles

SUT	System Under Test
DOC	Depended-on Component
Indirect Output	any message sent to a DOC
Indirect Input	any data received from a DOC
Double	umbrella term for any DOC that is controlled by the test
Mock	double used to test indirect outputs (fails immediately)
Spy	double used to test indirect outputs (does not fail immediately)
Stub	double used to provide indirect inputs to the SUT
Fake	double that is a full implementation of the DOC being replaced, but is simplified and customizable
Dummy	double that is just a placeholder; it will not be used by the SUT during the test



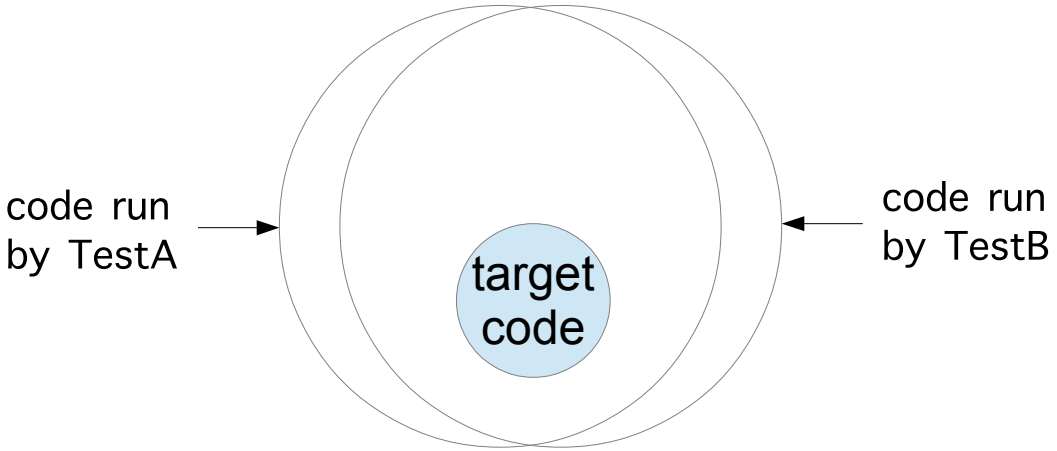
What does this get us?

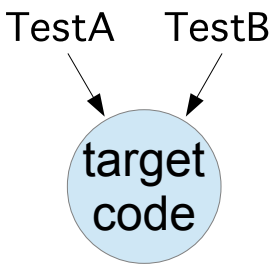
Faster tests

An integration test exercises a lot of “incidental” code which is not the target of the test.

That code can be very slow or have a costly setup, and tends to grow as new features are added.

Refactoring to more unit-level tests allows us to run only the code we're interested in testing, avoiding unnecessary overhead.





What does this get us? (cont)

More test coverage

Unit-like tests run much faster than integration tests, so it becomes possible to more thoroughly test the target code.

Cases that were previously considered “not worth it” can be now be enumerated.

What does this get us? (cont)

Pinpoint debugging

Because the SUT is small and isolated, the search for defective code is usually much easier.

HINT: It's probably what you just wrote!
(You've been running your tests as you code, right?)

What does this get us? (cont)

Isolation from error sources

Network outages, flaky services, deployment downtimes, other users' activities, etc., could cause dependencies to have unexpected, intermittent behaviors.

Debugging an intermittent problem can be extremely time-consuming and frustrating.

Isolated unit-like tests greatly reduce, if not entirely eliminate, this issue.

Isolation from extraneous details

What does this get us? (cont)

Ability to simulate errors

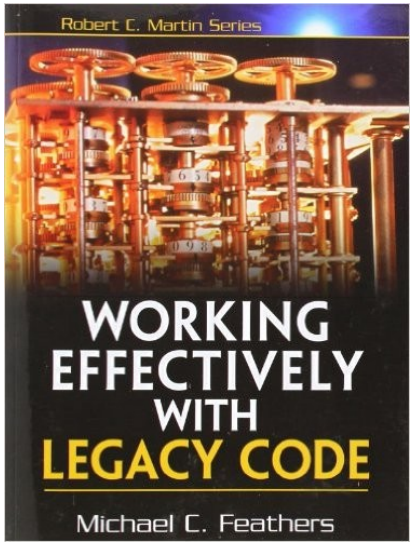
Sometimes we need to test the behavior of our system in the presence of network outages or other types of errors.

It might not be practical, or even possible, to create a situation where the error will occur reliably.

Refactoring a large SUT into smaller pieces creates **test seams** where **test doubles** can be polymorphically injected.

A test double can be coded to simulate an error condition, allowing us to observe whether or not the SUT handles it correctly.

Feathers



What does this get us? (cont)

Ability to make progress with less information and/or coordination

When DOCs are not yet implemented, or even defined, we can still make progress by using test doubles.

This is sometimes called “interface discovery”.

Once the “real” implementation becomes available, it can be plugged in directly (if its interface matches that of the double) or an adapter can be created.

On larger projects, this allows teams/individuals to work in parallel with relatively little coordination. It also enables emergent designs.

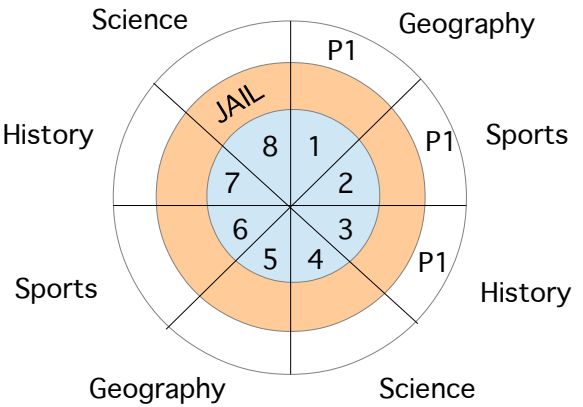
Mechanics of Testing

A simple game

The board is circular, with each slice representing a subject area. The subjects are TBD and are repeated in order around the board TBD number of times.

All players start at position one with zero gold. The game proceeds in rounds until one or more victors (players with five golds) are found, at which point the game ends. Each round consists of all players taking one turn.

A player's turn consists of rolling the dice. If the player is not in jail, the player advances the indicated number of positions and answers a question on that position's subject. If the answer is correct, the player receives one gold. Otherwise, the player is placed in jail. If the player is already in jail and the roll is even, the player remains in jail. If the player is already in jail and the roll is odd, the player answers a question. If the answer is correct, the player gets out of jail but receives no gold. If the answer in incorrect, the player remains in jail. Either way, the player cannot advance when starting a turn in jail.



P1: +1 gold

A TBD internet service will provide the subjects and questions.

The exact design of the board and other UI/UX issues are TBD.

Desired API:

```
Game.new.play
```

Where to start?

Write some tests!

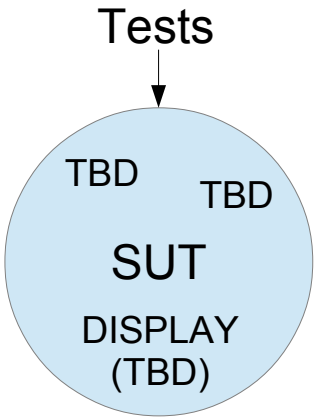
Writing tests forces you to think clearly about what you're trying to build and how it will work.

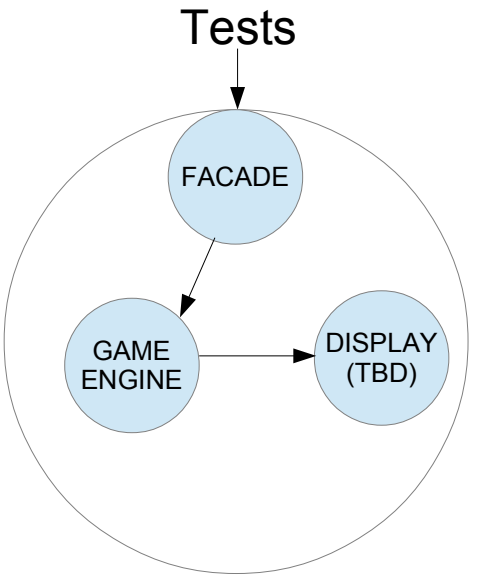
It doesn't take very long to realize that we're pretty stuck, given that the game's API doesn't have any dependency injection:

```
Game.new.play
```

The display is also TBD, so we don't even have a place to go look for expected behaviors.

We'll work around these limitation by using a facade (to preserve the API) and having it delegate to our own dependency-injectable classes behind the scenes.



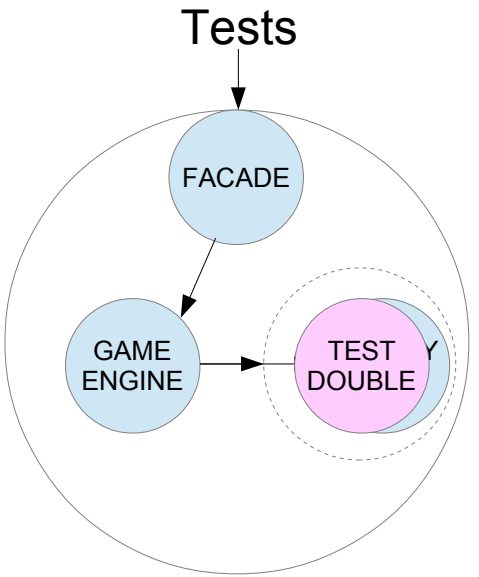


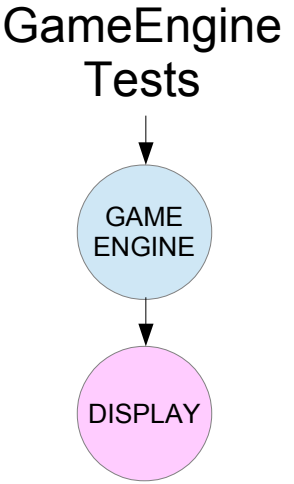
But there's no display!

No problem - we'll make an abstract display type!

We can use test doubles to determine what features the display type needs to support, and create the real production display class when the dust has settled.

We'll need to check that the `GameEngine` is sending the correct messages to the `Display` test double.





Show me the code!

```
2  class Game
3    def initialize
4      @display      = ProductionDisplay.new
5      @game_engine = GameEngine.new(display: @display)
6    end
7
8    def play
9      @game_engine.play
10   end
11 end
12
13 Game.new.play
```

```
2  RSpec.describe 'GameEngine' do
3    let(:game_engine) { GameEngine.new(display: display) }
4
5    let(:display) { double }
6
7    ## ...???...
8  end
```

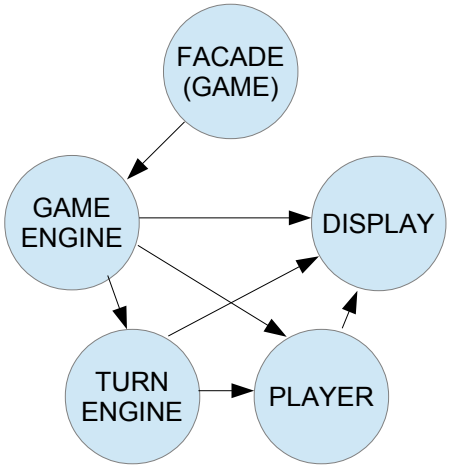
Progress, but we're still stuck...

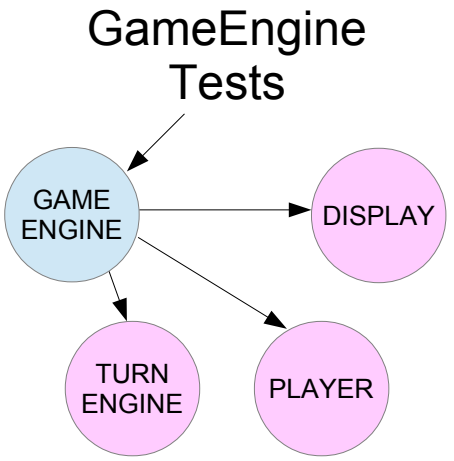
More decomposition

The missing UI/UX design prevents us from knowing exactly what should be displayed, and when.

Plus, running an entire game in every test will get annoying really fast...

We'll focus on the core business logic (the game loop), and just assume that the players are already determined.





Show me the code!

```
68  class Game
69    def initialize
70      @display      = ProductionDisplay.new
71      @turn_engine  = TurnEngine.new(display: @display)
72      @players      = 4.times.map{ ProductionPlayer.new(display: @display) }
73      @game_engine  = GameEngine.new(display:     @display,
74                                     turn_engine: @turn_engine,
75                                     players:     @players)
76    end
77
78    def play
79      loop do
80        break unless @game_engine.play
81      end
82    end
83  end
84
85  Game.new.play
```

Show me the code! (cont)

```
RSpec.describe 'GameEngine' do
  let(:game_engine) {
    GameEngine.new(display:     display,
                   players:     players,
                   turn_engine: turn_engine)
  }

  context 'when the game setup is invalid' do
    context 'because there are no players' do
      xit 'raises an exception'
    end
  end
  context 'when the game setup is valid' do
    context 'and victory conditions have been met' do
      context 'by a single player' do
        xit 'tells the display to indicate victory'
        xit 'returns false'
      end
      context 'by multiple players' do
        xit 'tells the display to indicate victories'
        xit 'returns false'
      end
    end
    context 'and victory conditions have NOT been met' do
      xit 'processes each player turn in order'
      xit 'returns true'
    end
  end
end
```

Show me the code! (cont)

```ruby
RSpec.describe 'GameEngine' do
  let(:game_engine) {
    GameEngine.new(display:     display,
                   players:     players,
                   turn_engine: turn_engine)
  }

  context 'when the game setup is invalid' do
    context 'because there are no players' do
      let(:display)     { double }
      let(:turn_engine) { double }
      let(:players)     { [] }

      it 'raises an exception' do
        expect{
          game_engine.play
        }.to raise_error(GameEngine::InvalidStateError)
      end
    end
  end
end
```

```ruby
class GameEngine
  class InvalidStateError < StandardError; end

  def initialize(display:, players:, turn_engine:)
  end

  def play
    raise InvalidStateError.new("there are no players")
  end
end
```

Adding more tests

```
18  RSpec.describe 'GameEngine' do
19    let(:game_engine) {
20      GameEngine.new(display: display, players: players, turn_engine: turn_engine)
21    }
22    let(:turn_engine) { double }
23    let(:display) {
24      dbl = double
25      allow(dbl).to receive(:show_victory).with(players: anything)
26      dbl
27    }
28    let(:alice) {
29      dbl = double
30      allow(dbl).to receive(:gold).and_return(4)
31      dbl
32    }
33    let(:bob) {
34      dbl = double
35      allow(dbl).to receive(:gold).and_return(5)
36      dbl
37    }
38
39    context 'when the game setup is valid' do
40      context 'and victory conditions have been met' do
41        context 'by a single player' do
42          let(:players) { [alice, bob] }
43
44          it 'tells the display to indicate victory' do
45            game_engine.play
46            expect(display).to have_received(:show_victory).with(players: [bob])
47          end
48          it 'returns false' do
49            expect(game_engine.play).to eq(false)
50          end
```

And making them pass

```ruby
class GameEngine
  class InvalidStateError < StandardError; end

  def initialize(display:, players:, turn_engine:)
    @display = display
    @players = players
  end

  def play
    raise InvalidStateError.new("there are no players") if @players.none?

    @display.show_victory(players: [@players.last])
    return false
  end
end
```

```
GameEngine
  when the game setup is valid
    and victory conditions have been met
      by a single player
        tells the display to indicate victory
        returns false
  when the game setup is invalid
    because there are no players
      raises an exception

Finished in 0.01395 seconds (files took 0.12834 seconds to load)
3 examples, 0 failures
```

Rinse...

```ruby
    context 'when the game setup is valid' do
      context 'and victory conditions have been met' do
        context 'by multiple players' do
          let(:players) { [bob, alice, edith] }

          it 'tells the display to indicate victories' do
            game_engine.play
            expect(display).to have_received(:show_victory).with(players: [bob, edith])
          end
          it 'returns false' do
            expect(game_engine.play).to eq(false)
          end
        end
```

```
Failures:

  1) GameEngine when the game setup is valid and victory conditions have been met by multiple players tells the display to indicate victories
     Failure/Error: expect(display).to have_received(:show_victory).with(players: [bob, edith])

       #<Double (anonymous)> received :show_victory with unexpected arguments
         expected: ({:players=>[#<Double (anonymous)>, #<Double (anonymous)>]})
              got: ({:players=>[#<Double (anonymous)>]})
       Diff:
       @@ -1,2 +1,2 @@
       -[{:players=>[#<Double (anonymous)>, #<Double (anonymous)>]}]
       +[{:players=>[#<Double (anonymous)>]}]

     # ./spec/engine/integration/code6_spec.rb:51:in `block (5 levels) in <top (required)>'

Finished in 0.02684 seconds (files took 0.08831 seconds to load)
5 examples, 1 failure

Failed examples:

rspec ./spec/engine/integration/code6_spec.rb:49 # GameEngine when the game setup is valid and victory conditions have been met by multiple pla
yers tells the display to indicate victories
```

Lather...

```ruby
class GameEngine
  class InvalidStateError < StandardError; end

  def initialize(display:, players:, turn_engine:)
    @display = display
    @players = players
  end

  def play
    raise InvalidStateError.new("there are no players") if @players.none?

    victorious_players = @players.select{|player| player.gold >= 5}
    @display.show_victory(players: victorious_players)

    return false
  end
end
```

```
GameEngine
  when the game setup is valid
    and victory conditions have been met
      by multiple players
        tells the display to indicate victories
        returns false
      by a single player
        tells the display to indicate victory
        returns false
  when the game setup is invalid
    because there are no players
      raises an exception

Finished in 0.00931 seconds (files took 0.08601 seconds to load)
5 examples, 0 failures
```

Repeat...

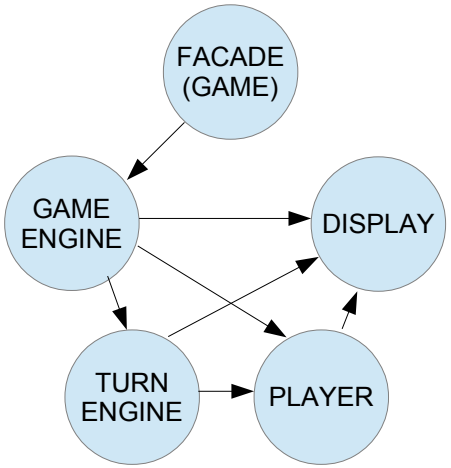
```
97    context 'and victory conditions have NOT been met' do
98      let(:players) { [alice, charlie, danny] }
99
100     it 'processes each player turn in order' do
101       game_engine.play
102       expect(turn_engine).to have_received(:process).with(player: alice).ordered
103       expect(turn_engine).to have_received(:process).with(player: charlie).ordered
104       expect(turn_engine).to have_received(:process).with(player: danny).ordered
105     end
106
107     it 'returns true' do
108       expect(game_engine.play).to eq(true)
109     end
110   end
```

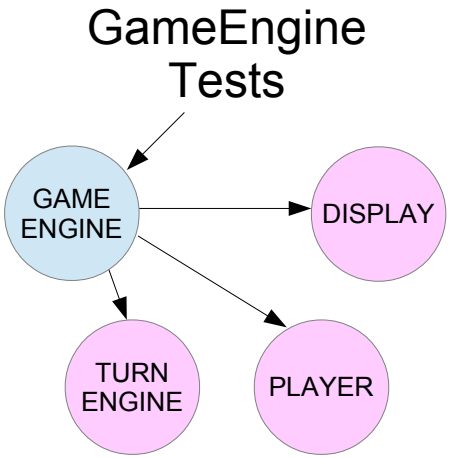
```
29  RSpec.describe 'GameEngine' do
30    let(:game_engine) {
31      GameEngine.new(display: display, players: players, turn_engine: turn_engine)
32    }
33
34    let(:display) {
35      dbl = double
36      allow(dbl).to receive(:show_victory).with(players: anything)
37      dbl
38    }
39    let(:turn_engine) {
40      dbl = double
41      allow(dbl).to receive(:process).with(player: anything)
42      dbl
43    }
44    let(:alice) {
45      dbl = double
```

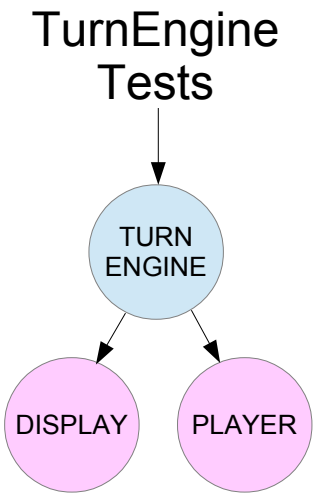
And so on...

```
2  class GameEngine
3    class InvalidStateError < StandardError; end
4
5    def initialize(display:, players:, turn_engine:)
6      @display     = display
7      @players     = players
8      @turn_engine = turn_engine
9    end
10
11   def play
12     raise InvalidStateError.new("there are no players") if @players.none?
13
14     victorious_players = @players.select{|player| player.gold >= 5}
15     if victorious_players.any?
16       @display.show_victory(players: victorious_players)
17       return false
18     end
19
20     @players.each do |player|
21       @turn_engine.process(player: player)
22     end
23
24     return true
25   end
26 end
```

Turning to TurnEngine







Turning to TurnEngine (cont)

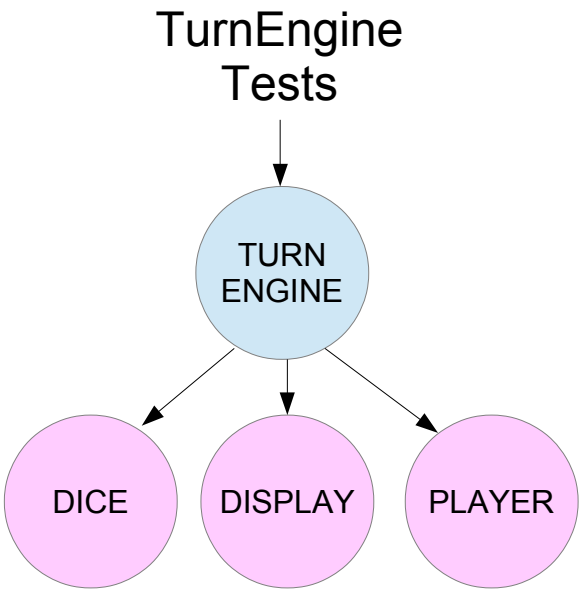
```
when the current player is in jail
  and the roll is even
    the display shows the roll
    the display shows that the current player did NOT get out of jail
    the current player is NOT asked a question
    the current player's position is NOT updated
```

```
and the roll is odd
  and the player answers the question correctly
    the display shows the roll
    the current player is asked a question
    the display shows that the answer was correct
    the current player's position is NOT updated
```

```
and the player answers the question incorrectly
  the display shows the roll
  the current player is asked a question
  the display shows that the answer was incorrect
```

Turning to TurnEngine

```
210  RSpec.describe 'TurnEngine' do
211    let(:turn_engine) {
212      TurnEngine.new(display: display,
213                     dice:    dice)
214    }
215
216    let(:roll) { 1 }
217    let(:dice) {
218      dbl = double
219      allow(dbl).to receive(:roll).and_return(roll)
220      dbl
221    }
```




```
253      context "and the roll is even" do
254        let(:roll) { 2 }
255
256        it "the display shows the roll" do
257          turn_engine.process(player: player)
258          expect(display).to have_received(:show_roll).with(player: player, roll: roll)
259        end
```

TurnEngine test details

```
248   context "when the current player is in jail" do
249     before(:each) do
250       allow(player).to receive(:in_jail?).and_return(true)
251     end
252
253     context "and the roll is even" do
254       let(:roll) { 2 }
255
256       it "the display shows the roll" do
257         turn_engine.process(player: player)
258         expect(display).to have_received(:show_roll).with(player: player, roll: roll)
259       end
260       it "the display shows that the current player did NOT get out of jail" do
261         turn_engine.process(player: player)
262         expect(display).to have_received(:show_remain_in_jail).with(player: player)
263       end
264       it "the current player is NOT asked a question" do
265         turn_engine.process(player: player)
266         expect(player).to_not have_received(:answer_question)
267       end
268       it "the current player's position is NOT updated" do
269         turn_engine.process(player: player)
270         expect(player).to_not have_received(:advance_position)
271       end
272       it "the current player does NOT get any gold" do
273         turn_engine.process(player: player)
274         expect(player).to_not have_received(:add_one_gold)
275       end
276       it "the current player does NOT get out of jail" do
277         turn_engine.process(player: player)
278         expect(player).to_not have_received(:go_to_jail)
279       end
280     end
```

TurnEngine class

```
167  class TurnEngine
168    def initialize(display:, dice:)
169      @display = display
170      @dice    = dice
171    end
172
173    def process(player:)
174      roll = dice.roll
175      display.show_roll(player: player, roll: roll)
176
177      if player.in_jail?
178        if roll % 2 == 0
179          display.show_remain_in_jail(player: player)
180        else
181          if player.answer_question
182            display.show_correct_answer(player: player)
183            player.get_out_of_jail
184            display.show_get_out_of_jail(player: player)
185          else
186            display.show_incorrect_answer(player: player)
187            display.show_remain_in_jail(player: player)
188          end
189        end
190      else
191        player.advance_position(roll: roll)
192        display.show_player_position(player: player)
193        if player.answer_question
194          display.show_correct_answer(player: player)
195          player.add_one_gold
196        else
197          display.show_incorrect_answer(player: player)
198          player.go_to_jail
199          display.show_now_in_jail(player: player)
200        end
201      end
202    end
```

Our doubles are growing

```
223   let(:display) {
224     dbl = double
225     allow(dbl).to receive(:show_roll).with(player: anything, roll: anything)
226     allow(dbl).to receive(:show_correct_answer).with(player: anything)
227     allow(dbl).to receive(:show_incorrect_answer).with(player: anything)
228     allow(dbl).to receive(:show_now_in_jail).with(player: anything)
229     allow(dbl).to receive(:show_remain_in_jail).with(player: anything)
230     allow(dbl).to receive(:show_get_out_of_jail).with(player: anything)
231     allow(dbl).to receive(:show_player_position).with(player: anything)
232     allow(dbl).to receive(:show_victory).with(players: anything)
233     dbl
234   }
235
236   let(:player) {
237     dbl = double
238     allow(dbl).to receive(:in_jail?).and_return(false)
239     allow(dbl).to receive(:go_to_jail)
240     allow(dbl).to receive(:get_out_of_jail)
241     allow(dbl).to receive(:advance_position).with(roll: anything)
242     allow(dbl).to receive(:answer_question).and_return(true)
243     allow(dbl).to receive(:add_one_gold)
244     allow(dbl).to receive(:gold).and_return(3)
245     dbl
246   }
```

Things to watch out for

Test-induced Design Damage (TDD)

If taken too far, refactoring for testability can cause code to become harder to maintain.

Always be aware of what you're trying to test and why, so the costs never outweigh the benefits.

David Heinemeier Hansson (DHH)

Complicated test setup

The more complicated the test setup, the more brittle the test (usually).

Check for Law of Demeter violations, mystery guests, and spooky actions-at-a-distance and remove them if possible and appropriate.

Determine if the design needs to be adjusted, or if the test is really worth the effort.

Things to watch out for (cont)

Mocking value objects

Some objects are so trivial that mocking them is more complicated than just creating the real deal.

E.g., objects with no collaborators or special behaviors ("values"): numbers, strings, arrays, data structures, etc.

Change-detector tests

A test can be so tightly coupled to the implementation that the SUT cannot be refactored or altered in even trivial ways.

"The display shows a welcome message"
...versus...
"The display shows: Welcome to TrIvIa SmAckdOwn"

Things to watch out for (cont)

Using tests in lieu of code reviews

Testing is great and all, but code reviews can still catch at lot of mistakes. Just because the code works doesn't mean it's easy to understand.

C.A.R. “Tony” Hoare

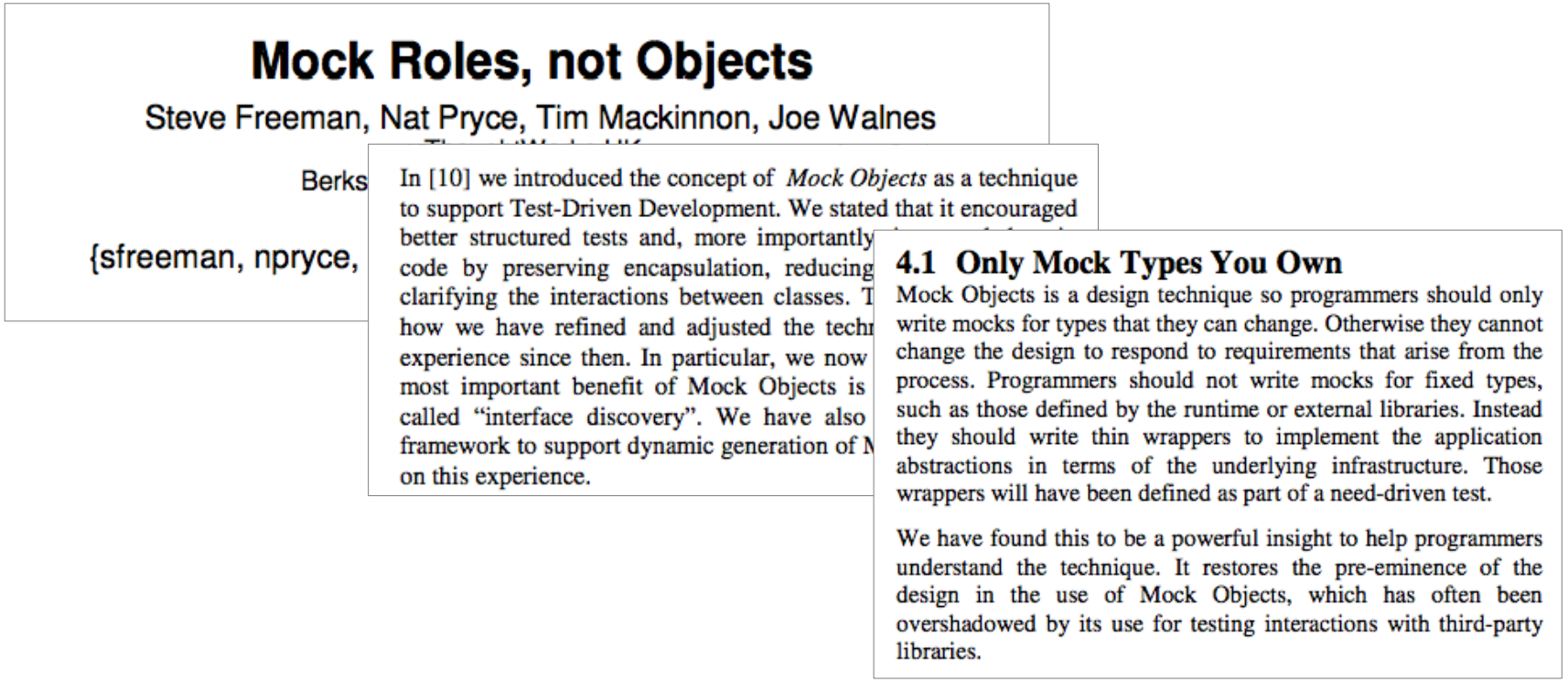
“There are two ways of constructing a software design: One way is to make it so **simple** that there **are obviously no deficiencies**, and the other way is to make it so **complicated** that there **are no obvious deficiencies**. The first method is far more difficult.”

"The real value of tests is not that they detect bugs in the code, but that they detect inadequacies in the methods, concentration, and skills of those who design and produce the code."

Things to watch out for (cont)

Do not mock objects you don't own.
Do NOT mock objects you don't own.
Do **NOT** mock objects you don't own.

Mocking objects you don't own almost always results in tests that don't actually test anything of consequence. Integration tests are needed at the edges of your system.



Mock Roles, not Objects

Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes

Berks

{sfreeman, npryce,

In [10] we introduced the concept of *Mock Objects* as a technique to support Test-Driven Development. We stated that it encouraged better structured tests and, more importantly
code by preserving encapsulation, reducing
clarifying the interactions between classes. T
how we have refined and adjusted the techn
experience since then. In particular, we now
most important benefit of Mock Objects is
called "interface discovery". We have also
framework to support dynamic generation of M
on this experience.

4.1 Only Mock Types You Own

Mock Objects is a design technique so programmers should only write mocks for types that they can change. Otherwise they cannot change the design to respond to requirements that arise from the process. Programmers should not write mocks for fixed types, such as those defined by the runtime or external libraries. Instead they should write thin wrappers to implement the application abstractions in terms of the underlying infrastructure. Those wrappers will have been defined as part of a need-driven test.

We have found this to be a powerful insight to help programmers understand the technique. It restores the pre-eminence of the design in the use of Mock Objects, which has often been overshadowed by its use for testing interactions with third-party libraries.

Closing Thoughts

Types of testing

Example-based

Specific examples are enumerated, and the expected behaviors for each are described.

Everything in this talk falls into this category.

Property-based

System invariants, preconditions, and postconditions are described, and the computer searches for examples meeting the preconditions that break the invariants or postconditions.

Examples include hypothesis (python), ScalaCheck (scala) - both derived from Haskell's QuickCheck

Mutation-based

Code is “surgically” altered by the computer (creating a “mutant”) to see if the test suite detects the change.

Alternatives to testing

Provable code

It is possible, in some cases, to ensure certain code behaviors via language guarantees.

Example: If TaskA must come before TaskB, create an object whose constructor performs TaskA and pass it to the code that performs TaskB.

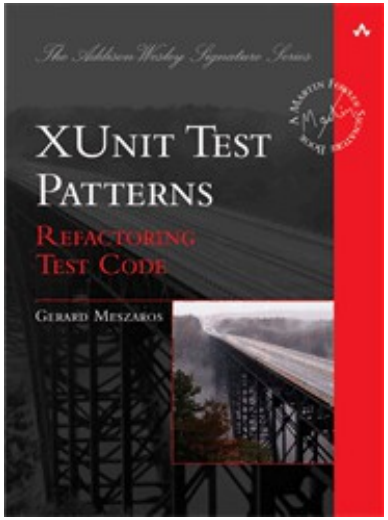
Michael L Perry

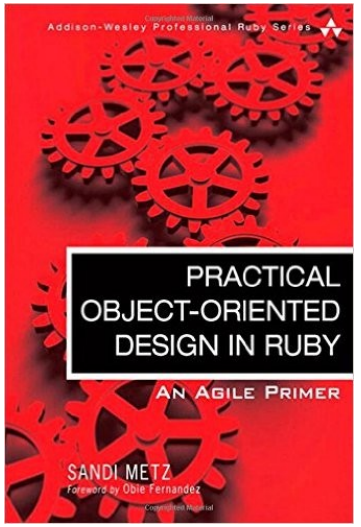
Interface redesign

If a SUT is difficult to test because it has a complicated specification (lots of rules, customizations, potential ways to misuse, error conditions, etc.), sometimes redesigning it can greatly reduce its complexity.

This is one example of how testing, or even the threat of testing, can help improve the design and reliability of code.

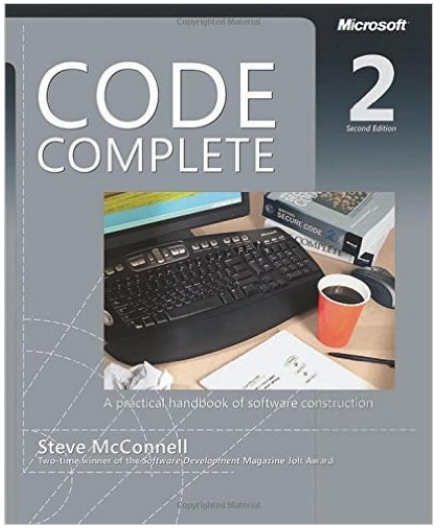
More resources





Google Clean Code Talks

Misko Havery



Clarity of code comes from clarity of thought.

Thank you for coming!

Attributions

Image of Michael Feathers (blog):
https://www.goodreads.com/author/show/25201.Michael_C_Feathers/blog
Image of DHH (Wikipedia - James Duncan Davidson/O'Reilly Media, Inc.):
https://commons.wikimedia.org/wiki/File:David_Heinemeier_Hansson.jpg
Image of C.A.R. Hoare (Wikipedia - Rama):
https://commons.wikimedia.org/wiki/File:Sir_Tony_Hoare_IMG_5125.jpg
Image of Michael L. Perry (GitHub):
https://github.com/michaellperry
Image of XUnit Test Patterns:
http://xunitpatterns.com/
Image of Practical Object-Oriented Design in Ruby:
http://www.sandimetz.com/products/
Image of Code Complete 2 (Amazon.com):
https://www.amazon.com/Code-Complete-Practical-Handbook-Construction/dp/0735619670
Image of CleanCoders logo:
https://cleancoders.com/videos
Image of Destroy All Software:
https://www.destroyallsoftware.com/screencasts
Image of Misko Havery:
http://misko.hevery.com/about/