functional testing lessons learned

Jay Fields
DRW Trading

testing is hard

testing is hard

over specification

testing is hard

RSpec, Selenium, test/unit or ...?

testing is hard

subsystem dependencies

testing is hard

reference and state data

bigger issues

testing themes

testing themes

test code is as important as application code

testing themes

no automated test suite can replace exploratory testing

testing themes

create tests that provide a positive return on investment
(100% coverage is not the right goal)

testing themes

prove the system works as expected

too many functional tests?

over specification

don’t verify corner cases in functional tests

```
class ProcessState
  def initialize(state)
    case state
    when "running" then extend(RunningState)
    when "paused" then extend(PausedState)
    when "complete" then extend(CompleteState)
    end
  end
end
```

the constructor logic should be unit tested in isolation and not as part of a functional test

unit test suite

```
Expectations do
  expect RunningState do
    ProcessState.new "running"
  end

  expect PausedState do
    ProcessState.new "paused"
  end

  expect CompleteState do
    ProcessState.new "complete"
  end
end
```

functional test suite

```
test "state should be set when returned from the database" do
  process = Process.find_paused(:first)
  assert_equal true, process.is_a?(PausedState)
end
```

unit tests

functional tests

RSpec, Selenium, test/unit or ...?

test/unit

test/unit

bad

QA unfriendly (syntax is ugly)

test/unit

bad object oriented (tests are procedural)

test/unit

good

familiar to developers

test/unit

good

granular defect localization

test/unit

conclusion: decent option

selenium

selenium

bad

slow and browser bugs
can cause failures

selenium

bad

hard to find the defect location

selenium

good

runs against the full stack

selenium

good

easy for devs and testers to use

selenium

conclusion: too brittle and slow

RSpec

RSpec

bad

steep learning curve

RSpec

bad

hard to extend

RSpec

good

encourages behavioral testing

RSpec

good

devs and testers can use

RSpec

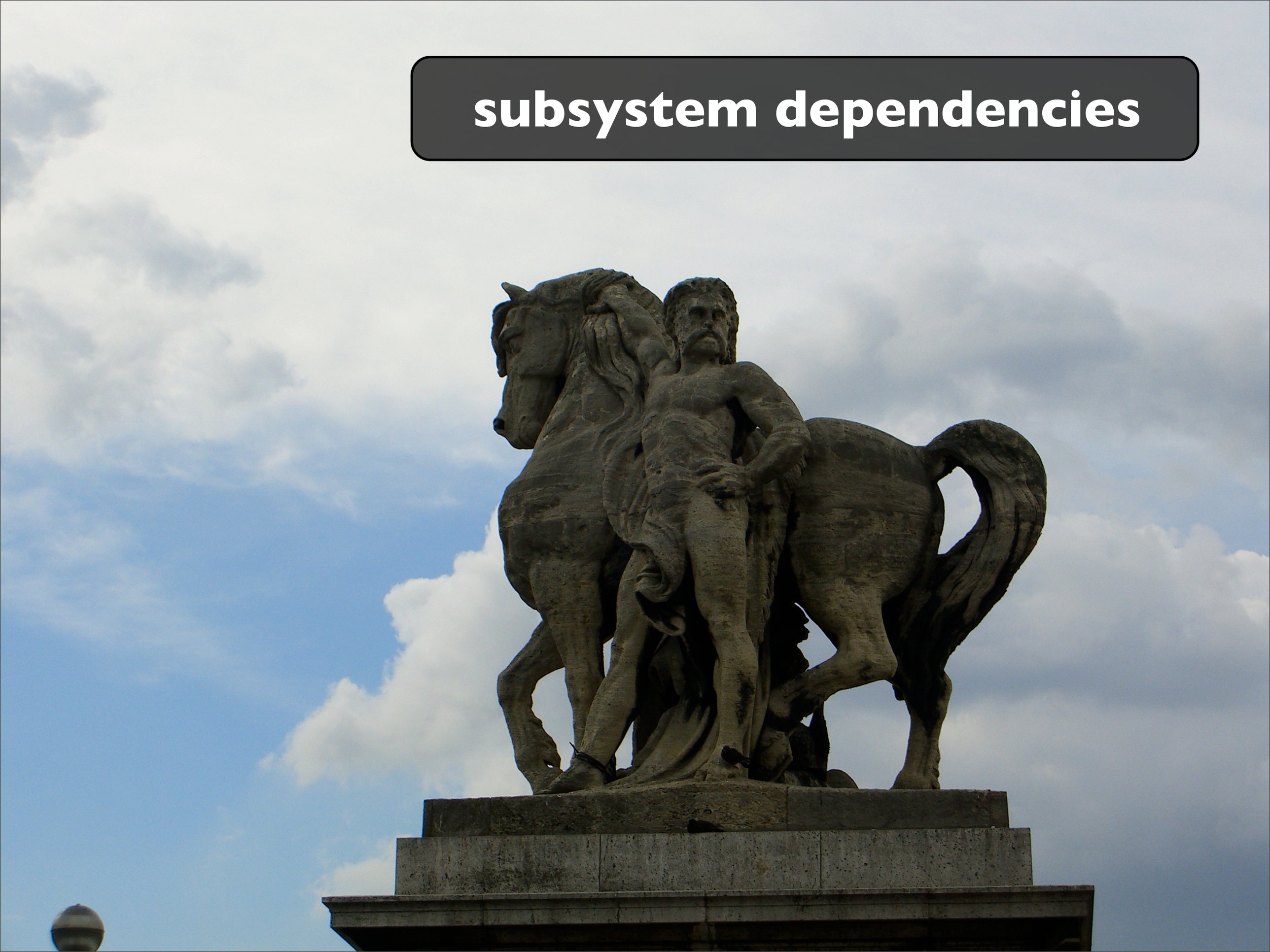
conclusion: best available solution

unit tests

functional tests

smoke tests

subsystem dependencies



subsystem dependencies

subsystem dependencies

you should test against subsystems you own

subsystem dependencies

database, services you've written, etc

subsystem dependencies

you should not test against subsystems you don't own

subsystem dependencies

credit card vendor api, google maps api, twitter api, etc

subsystem dependencies

if you don't own it, stub it

stubbing subsystems

stubbing subsystems

create a gateway to the subsystem

stubbing subsystems

in functional tests, load a stub version of the gateway that always returns the same (valid) information

stubbing subsystems

create an exterals test suite to exercise the 3rd party api

stubbing subsystems

application gateway

```
class FeedBurnerGateway
  def self.create_feed
    # connect to feedburner
    # create new feed
  end

  def self.feeds
    # connect to feedburner
    # request a list of feeds
  end
end
```

stub gateway

```
class FeedBurnerGateway
  def self.create_feed
  end

  def self.feeds
    [Feed.new("feed one", "myblog.com")]
  end
end
```

unit tests

functional tests

smoke tests

external tests

state & reference data

state & reference data

state & reference data

reference data almost never changes

state & reference data

50 states, partner names, brands you carry, etc

state & reference data

state data changes as a user interacts with the system

state & reference data

shopping cart contents, top sellers list, vip status, etc

state & reference data

reference data and state data are different and should be treated as such

reference data

reference data

can often be stored in yaml files

reference data

you can preload a database with reference data and work with the dump instead of the yaml files

reference data

should be loaded once and reused between all tests

reference data

if you have a vast amount of reference data you may need to work with a subset when testing

reference data

fixtures are a decent solution

state data

state data

a factory can greatly simplify data creation

state data

a factory allows you to easily create a valid instance of any of your domain models

state data

tests create the state data they need to execute

state data

changes to state data within one test will not cause other tests to fail

state data

fixtures are a terrible solution

test factory

```
class Factory
  def self.create_car(attributes)
    defaults = {
      :color => :green
      :engine => create_engine
      :make => "mercedes"
      :year => 2008
    }
    Car.create!(defaults.merge(attributes))
  end

  # ... create method per model
end
```

questions?

where to go from here

the path forward

the path forward

move corner case testing to unit tests

the path forward

remove painful tests that are providing little value
(writing tests is easy, deleting tests is hard)

the path forward

as tests break migrate them to better written RSpec tests

the path forward

create stubs for all systems which are used but not owned
(loading stubs is as easy as “require ‘feedburner_stub’”)

the path forward

break data into state or reference data and refactor tests to rely on reference data, but create it's own state data

Jay Fields jay@jayfields.com
http://blog.jayfields.com
DRW Trading http://drwtrading.com