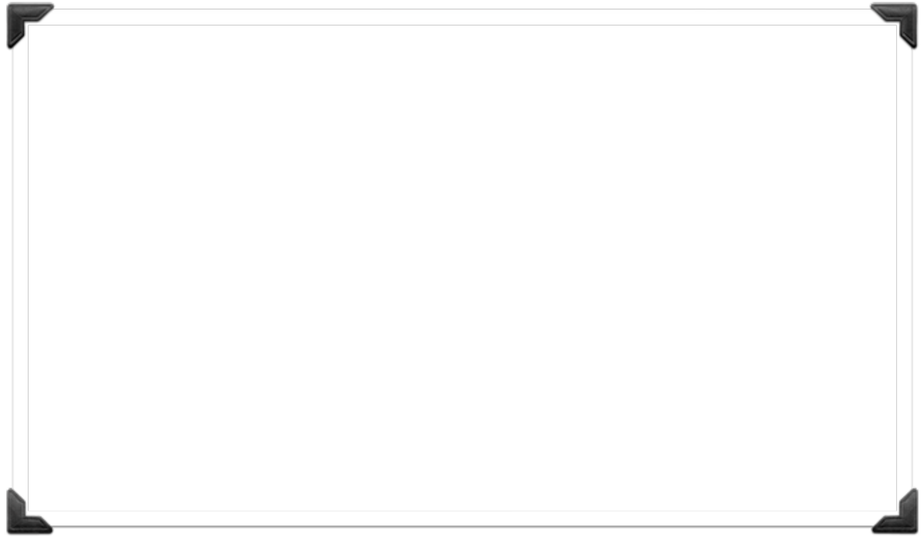
Test Doubles

Designing and Maintaining Software (DAMS)

Louis Rose



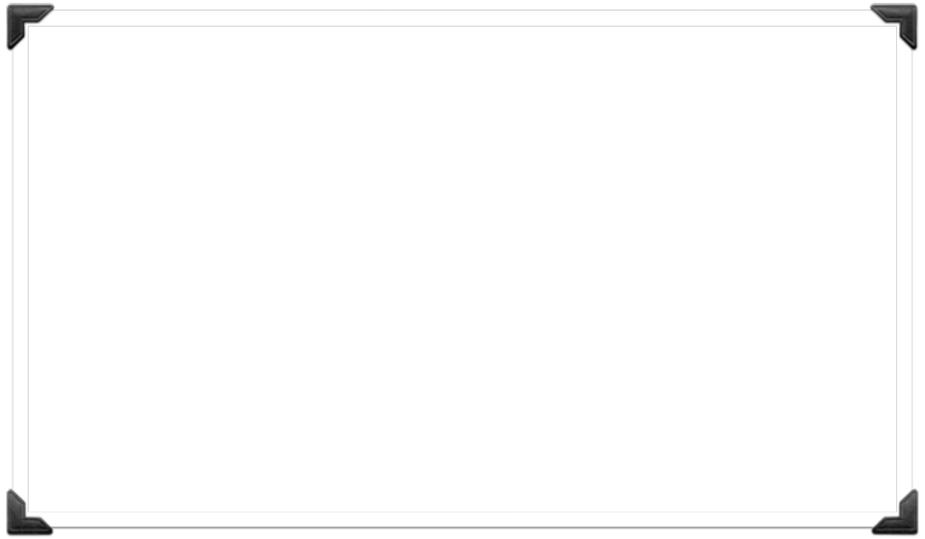
Problem: how do we test this code?

This class charges a customer using a service called “Braintree” (https://www.braintreepayments.com)

```
class Subscription
  attr_reader :id, :logger, :payments

  def initialize(id)
    @id = id
    @logger = Logger.new
    @payments = Braintree.new
  end

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end
```



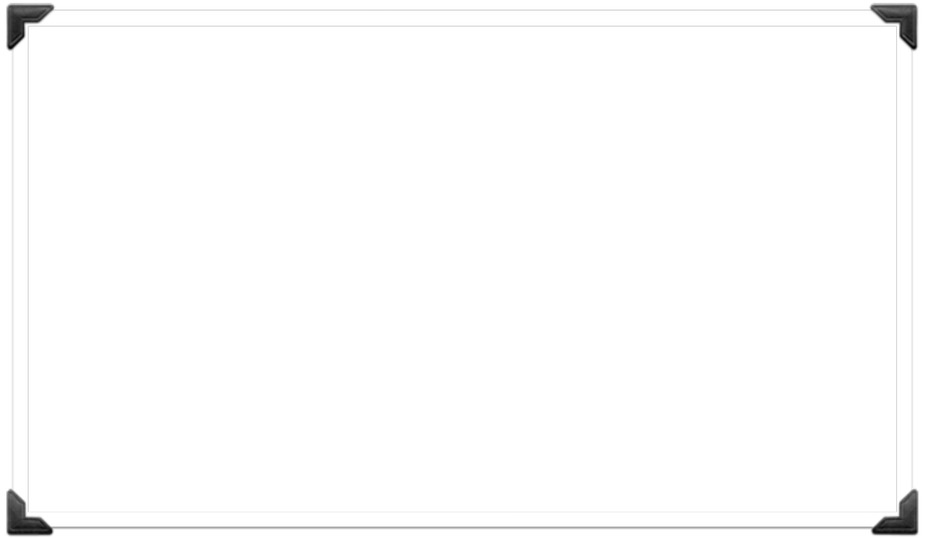
Problem: how do we test this code?

Telling Braintree to charge a customer in our tests is probably a bad idea!

```
class Subscription
  attr_reader :id, :logger, :payments

  def initialize(id)
    @id = id
    @logger = Logger.new
    @payments = Braintree.new
  end

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end
```



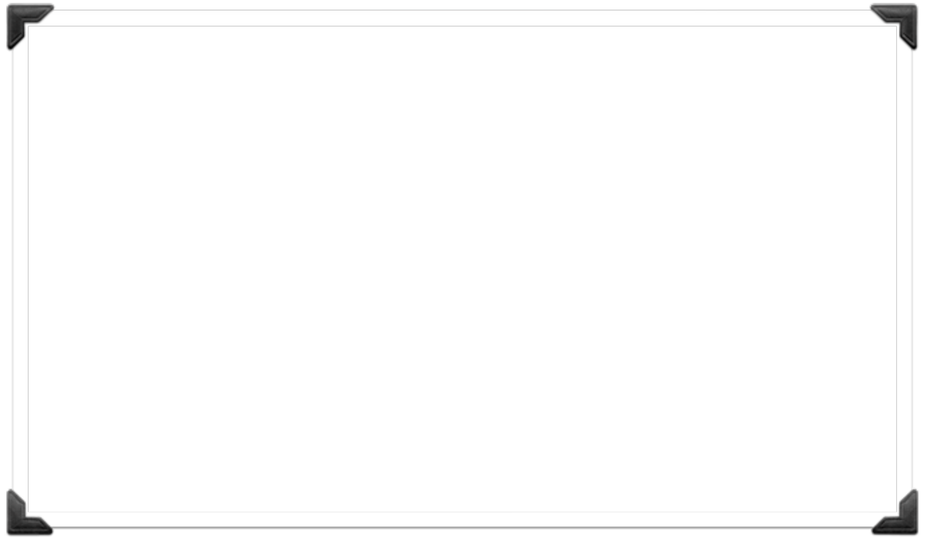
Solution: use a fake

A fake is an object that acts exactly like the real component, but does not affect our production environment.

```
class Subscription
  attr_reader :id, :logger, :payments

  def initialize(id, payments = Braintree.new)
    @id = id
    @logger = Logger.new
    @payments = payments
  end

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end
```



Solution: use a fake

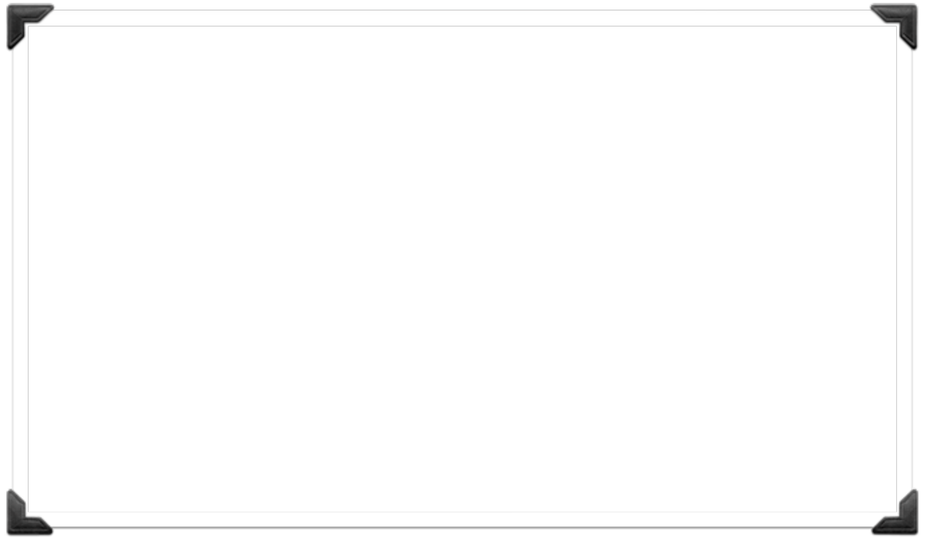
A fake is an object that acts exactly like the real component, but does not affect our production environment.

```
class Subscription
  attr_reader :id, :logger, :payments

  def initialize(id, payments = Braintree.new)
    @id = id
    @logger = Logger.new
    @payments = payments
  end

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end

fake = Braintree::Sandbox.new
Subscription.new(42, fake).bill(500)
expect(fake.exists(subscription_id: 42)).to be_truthy
```



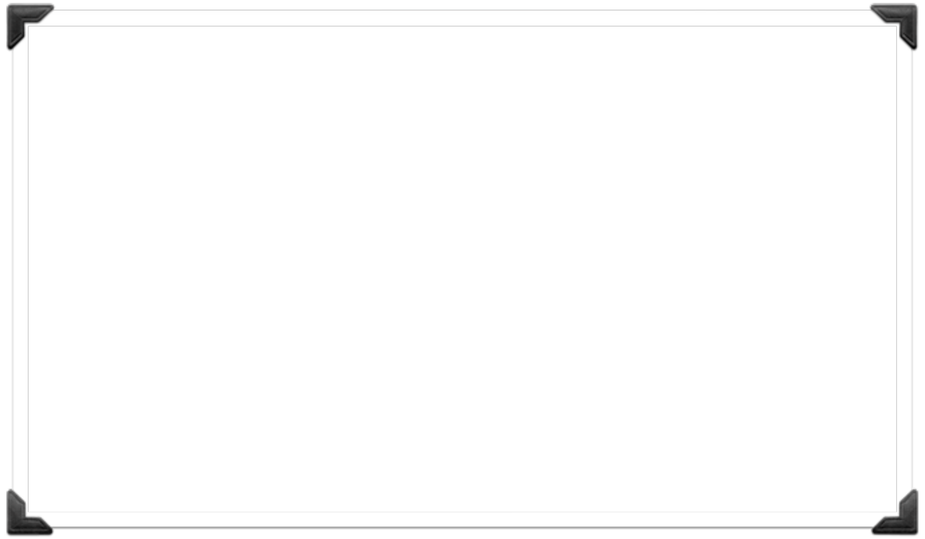
Problem: noisy tests cases

None of our tests care about whether or not logging messages are printed.

```
class Subscription
  attr_reader :id, :logger, :payments

  def initialize(id, payments = Braintree.new)
    @id = id
    @logger = Logger.new
    @payments = payments
  end

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end
```



Solution: use a dummy

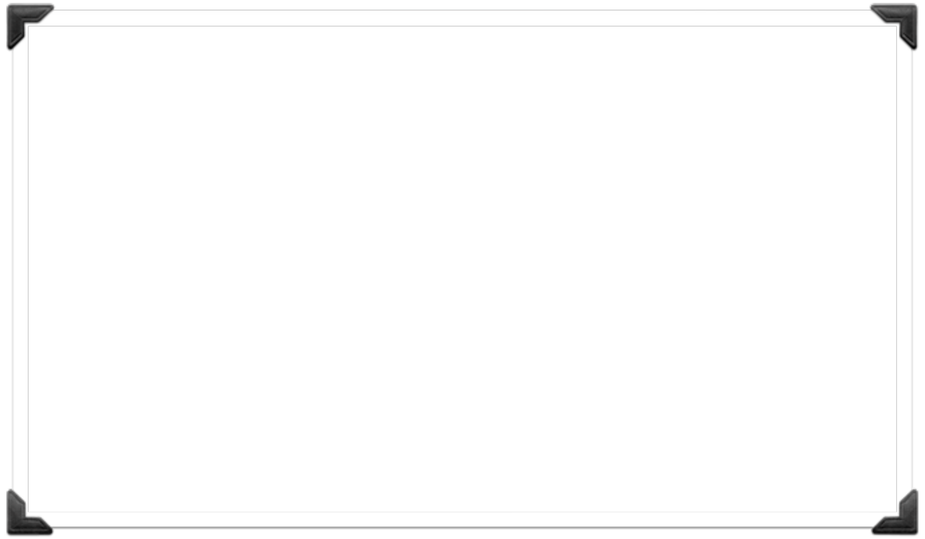
Inject a dummy logger. Dummies respond to messages (like print) by doing nothing.

```ruby
class Subscription
  attr_reader :id, :logger, :payments

  def initialize(id, logger = Logger.new, payments = Braintree.new)
    @id = id
    @logger = logger
    @payments = payments
  end

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end

class DummyLogger; def print(message); end; end
dummy = DummyLogger.new
Subscription.new(42, Braintree::Sandbox.new, dummy).bill(500)
```



Solution: use a dummy

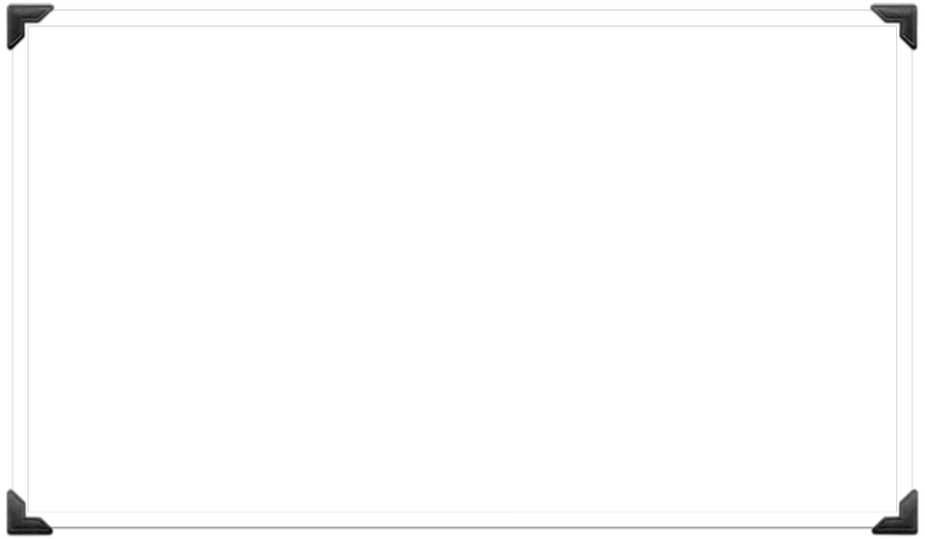
Rather than write our own Logger class, we can instead use RSpec to create a dummy.

```
class Subscription
  attr_reader :id, :logger, :payments

  def initialize(id, logger = Logger.new, payments = Braintree.new)
    @id = id
    @logger = logger
    @payments = payments
  end

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end

dummy = double("SilentLogger").as_null_object
Subscription.new(42, Braintree::Sandbox.new, dummy).bill(500)
```

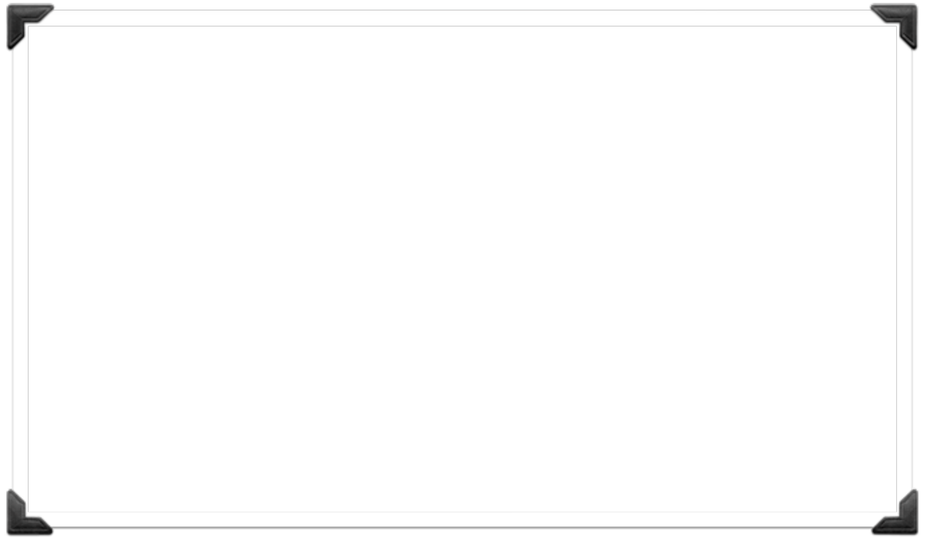


Problem: slow unit tests

Our **unit** tests are slow because they call out to an external service, and they fail when that service is unavailable.

```
class Subscription
  ...

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end
```



Solution: use stubs for queries

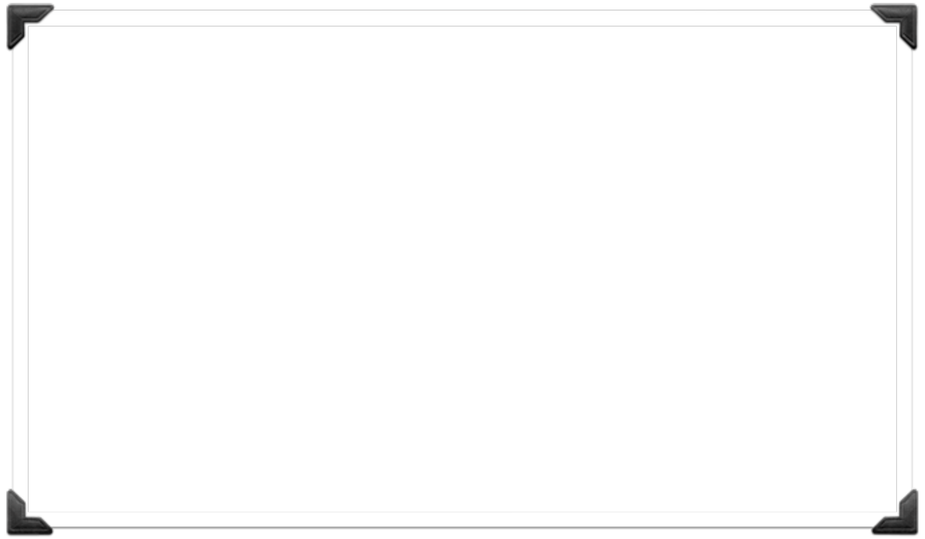
When we are testing the way that a unit behaves when a condition is met, use a stub to setup the condition.

```
class Subscription
  ...

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end

logger = double("SilentLogger").as_null_object

`
```



Problem: slow unit tests

Our **unit** tests are slow because they call out to an external service, and they fail when that service is unavailable.

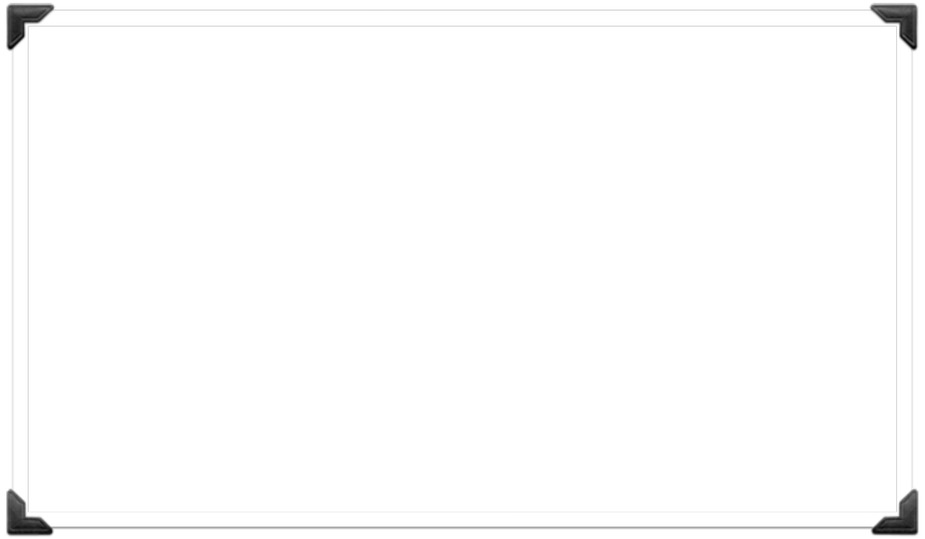
```
class Subscription
  ...

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end

logger = double("SilentLogger").as_null_object

payments = double("Payments")
allow(payments).to_receive(:exists).and_return(false)

Subscription.new(42, logger, payments).bill(500)
expect(fake.exists(subscription_id: 42)).to be_truthy
```



Solution: use mocks for commands

When we are testing that a collaborator must receive a specific message, use a mock.

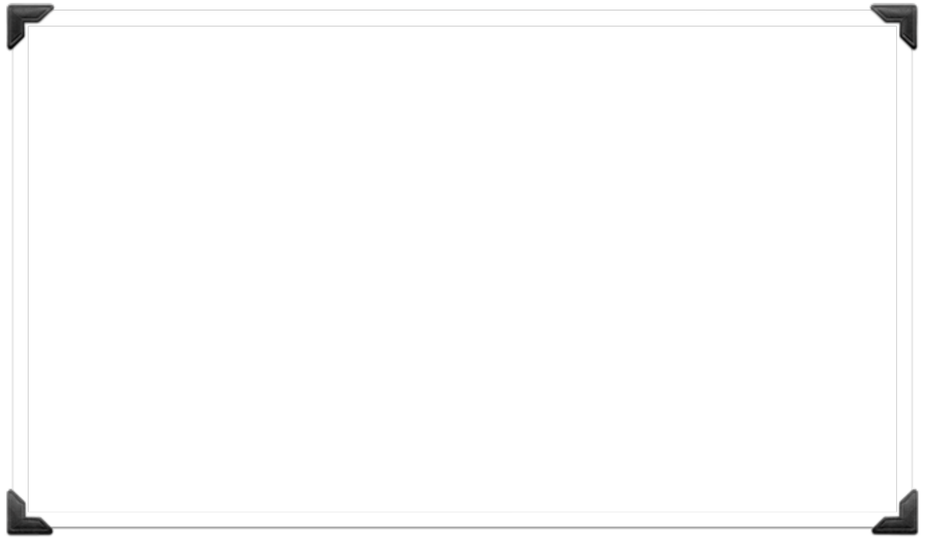
```
class Subscription
  ...

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end

logger = double("SilentLogger").as_null_object

payments = double("Payments")
allow(payments).to_receive(:exists).and_return(false)
expect(payments).to_receive(:charge)
                .with(subscription_id: 42, amount: 500)

Subscription.new(42, logger, payments).bill(500)
```



Solution: use spies for commands

When we are testing that a collaborator must receive a specific message, use a spy for a more natural test case.

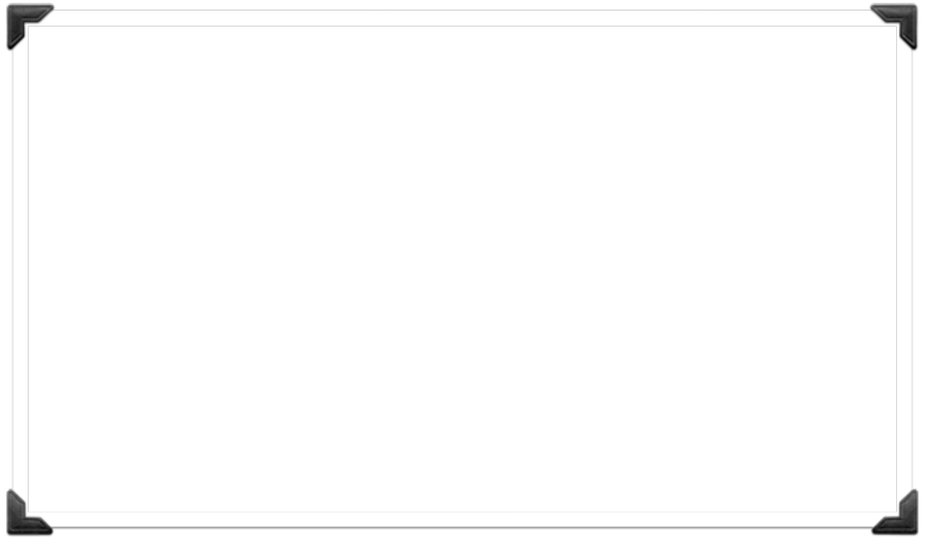
```
class Subscription
  ...

  def bill(amount)
    unless payments.exists(subscription_id: id)
      payments.charge(subscription_id: id, amount: amount)
      logger.print "Billed subscription #{id}"
    end
  end
end

logger = double("SilentLogger").as_null_object

payments = spy("Payments")
allow(payments).to_receive(:exists).and_return(false)

Subscription.new(42, logger, payments).bill(500)
expect(payments).to_have_received(:charge)
                .with(subscription_id: 42, amount: 500)
```



Summary

Fake	Replace an external service with a test-equivalent
Dummy	Replace an unimportant collaborator with a no-op
Stub	Replace a query with a canned response
Mock	Replace a command with an expectation (assertion)
Spy	Replace a command with an expectation (assertion)

Resources

- https://speakerdeck.com/skmetz/magic-tricks-of-testing-railsconf

http://rspec.info/documentation/3.3/rspec-mocks