THE EFFECT OF ROLE-BASED MOCKS ON TEST COUPLING IN INTERPRETED LANGUAGES

CHRISTIAN TREPPO

To Test A Mocking Bird

Master of Science (MSc)
Web Technologies
School Of Computing
National College of Ireland

August 10, 2014 – version 0.1

Christian Treppo: *The Effect Of Role–Based Mocks on Test Coupling in Interpreted Languages,* To Test A Mocking Bird, © August 10, 2014

If you just learn a single trick, Scout, you'll get along a lot better with all kinds of folks. You never really understand a person until you consider things from his point of view... Until you climb inside of his skin and walk around in it.

— Atticus Finch in *To Kill A Mockingbird (1962)*

ABSTRACT

Explicit Interfaces – sometimes also called Protocols – are a common feature of dynamically and statically typed, compiled languages. At compile time, an object's method signatures are checked against the signatures defined in the interface to ensure type compliance. Such an object's client object can then use any instance implementing this protocol with the type safety, the compiler provides. Programming to an interface instead of programming to concrete implementations is a practice, that decreases coupling between components and therefore increases maintainability. In the test driven development it is the foundation for the use of Mock Objects, that stand in for a collaborator and implement the same interface, to isolate the object under test from all other parts of the system it is connected to. Mock Objects are an effective tool to drive software design, as they can stand in for objects, that are not even implemented yet, and lead to the discovery of new collaborators and their interfaces.

In interpreted languages, that are compiled with a Just–In–Time compiler at runtime, strict type safety with interfaces is not possible. Nevertheless it is common practice – often referred to as Duck Typing – to program to a collaborator's interface, even though it is an implicitly and not externally defined interface. Mock Objects in the tests then also implement that same interface. The absence of static type checking can lead to the problem, that the Mock Object's interface and the real object's interface diverge without noticing – the tests would still pass as they use the Mock Object and really test the wrong behavior. This can be hard to debug and make the tests unreliable. Further it creates coupling between the test and the concrete implementation of a collaborator, which in turn decreases maintainability of a system.

By implementing explicitly defined Protocols – or Roles – in a testing environment, that Mock Objects and concrete objects are checked against, this divergence and coupling could be avoided, making tests and mock objects more reliable and maintainable.

ACKNOWLEDGMENTS

Put your acknowledgments here.

Many thanks to everybody who already sent me a postcard!

Regarding the typography and other help, many thanks go to Marco Kuhlmann, Philipp Lehman, Lothar Schlesier, Jim Young, Lorenzo Pantieri and Enrico Gregorio[1], Jörg Sommer, Joachim Köstler, Daniel Gottschlag, Denis Aydin, Paride Legovini, Steffen Prochnow, Nicolas Repp, Hinrich Harms, Roland Winkler, Jörg Weber, and the whole LaTeX-community for support, ideas and some great software.

Regarding LyX: The LyX port was intially done by *Nicholas Mariette* in March 2009 and continued by *Ivo Pletikosić* in 2011. Thank you very much for your work and the contributions to the original style.

1 Members of GuIT (Gruppo Italiano Utilizzatori di TeX e LaTeX)

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

DSL	Domain Specific Language
SUT	Subject Under Test
CDCT	Change Dependency between Collaborator Classes and Test
CDBC	Change Dependency between Classes

1 LITERATURE REVIEW

1.1 INTRODUCTION

1.2 MEASURING COUPLING IN OBJECT–ORIENTED PROGRAMS

1.3 COUPLING AND MAINTAINABILITY

1.4 TESTING WITH MOCKS

1.5 CONCLUSION

2 BACKGROUND

2.1 INTRODUCTION

2.2 PROBLEM DEFINITION

For professional software projects, automated software unit testing is an essential part of development. It can be helpful in checking and asserting the logical behavior of a software unit or module, by testing it in isolation from the rest of the system. To isolate the software unit or Subject Under Test (SUT) from all other units it collaborates with (instead of instantiating them in the test), testing mocks can be used to represent the interface of the collaborators for the sake of asserting all method calls are done correctly and, if needed, to return fake responses, that the SUT uses to finish its execution. Mocking has three useful properties:

1. They can bring down testing execution time by replacing the need for time–consuming processes (e. g.database calls or IO in general)
2. In the design phase, mocks can stand in for collaborators, that do not exist in the system yet, and can therefore be a useful tool to discover the interface of these new classes.
3. Because they isolate the SUT from its neighbors, changes to collaborators, that don't change their interfaces, require no follow–up work to update the SUT or the test. They decrease coupling and increase a system's maintainability.

In programming languages with explicit interface definitions, such as Java, Clojure or C#, classes can and mocks to be created by deriving from such an interface definition. The compiler then takes care, that all classes derived from an interface, correctly implement the methods defined in the interface. Therefore mocks and the classes they stand in for, always comply to their interface. In interpreted languages with dynamic type systems, such a guarantee is not possible, as there are no explicit interfaces and there is no compilation step, that executes type checks to discover type errors or mismatches upfront. In the worst case, such mismatches are only caught in a running system, that is already in productive use, when runtime errors occur.

This research will investigate the implementation of role–based mocks to enable interface guarantees and safe, isolated tests in interpreted lan-

Listing 1: An example Rspec test

```ruby
describe 'Logger' do
  subject { described_class.new }

  it 'has a default log level' do
    expect(logger.level).to eq 'error'
  end
end
```

guages. Role–based mocks will then be evaluated in terms of the effect they have on the coupling of unit tests.

2.3 RUBY FOR WEB APPLICATIONS

2.4 TESTING WITH RSPEC–MOCKS

After deciding on the Ruby language for the implementation and evaluation of the role–based mocking library, it was necessary to have a look at the ecosystem of testing frameworks, that are in use in Ruby and Ruby on Rails web applications.

The prevalent Ruby on Rails framework endorses the testing library Minitest, which is built into the Ruby language itself and is – as the name suggests – a testing framework with basic testing features. But Minitest is not the only supported testing framework, the hot contender is Rspec.

Rspec is a fully featured testing framework, that comes with a powerful Domain Specific Language (DSL) for describing test cases, an assertion library, that provides flexible and readable test assertions and its own mocking library – Rspec Mocks. Rspec has been a very successful in the Ruby community because of its expressive DSL (see Listing 1), which has been copied by testing frameworks in other languages (e. g.Nspec in .NET).

And indeed all the open source Ruby web applications, that were sampled for this research (see Appendix A), use the Rspec testing framework for their automated unit–level tests. In order to evaluate the effect of role–based mocks on test maintainability, it became obvious, that the mocking library to be built has to integrate with Rspec.

Since Rspec has a built–in mocking library of its own – Rspec–Mocks – the change dependency attribute for the sampled unit test files can be compared between the built–in library and the research project library.

Rspec in its latest version has three different kinds of mocks, that each cater for a slightly different use case:

METHOD STUBS are a way to intercept a method call to a real class and define a fake return value, that is used in the SUT. Stubs use real classes in the test and therefore lead to tight coupling between the test and the stubbed collaborator – e. g.if the collaborator's name changes, every test, where is used for stubbing, has to be changed accordingly. See Listing 2 for an example.

Listing 2: An example Rspec test using a method stub

```ruby
describe 'UserUpdater' do
  subject { described_class.new }
  let(:user) { Factory.create(:user) }

  it 'returns the updated user' do
    allow(UserRepo).to receive(:find).with(1).and_return(user)
    updated_user = subject.update(1, address: 'new address')

    expect(updated_user.address).to equal 'new address'
  end
end
```

DOUBLES are mocks, which are created ad hoc and can be used instead of any object. They are used as dummy collaborators to the SUT to test for message passing between the two classes. Any expected message call to the double can also return a value (see Listing 3 for the usage of doubles). Doubles are in fact a way to test a role rather than a concrete implementation – the concrete collaborator implementation might be exchanged, but as long as the interface is still the same, the SUT and the test don't have to be touched. Unfortunately doubles are not verified: If a double is expected to receive a method call to a method, which does not exist on the real collaborator, that is being mocked (e. g.it has been renamed), the test using the wrong method name will still pass. That means, that doubles and the classes they represent can get out of sync. In the worst case, this error will only be discovered at runtime in production!

Listing 3: An example Rspec test using a double

```ruby
describe 'UserFinder' do
  let(:user_repo) { double('UserRepo') }
  let(:user) { Factory.create(:user) }
  subject { described_class.new(user_repo) }

  it 'finds a user by id in the repo' do
    expect(user_repo).to receive(:find).with(1).and_return(user)

    found_user = subject.find(1)

    expect(found_user).to equal user
  end
end
```

VERIFIED MOCKS have been introduced in the current version of Rspec–Mocks. They were added to avoid the aforementioned problem of out of sync doubles by checking if the called method really exists on the mocked collaborator. Therefore if the method of the collaborator is – for example – renamed, the test using the verified mock will fail. Unfortunately verified mocks have the same disadvantages as method stubs – they are tied to a concrete implementations, rather than to an interface and therefore have a high change dependency. See Listing 4 for an example using verified mocks.

Listing 4: An example Rspec test using a verified mock

```ruby
describe 'UserUpdater' do
  let(:logger) { instance_double('Logger') }
  subject { described_class.new(logger) }

  it 'logs every update' do
    expect(logger).to receive(:log).with('update address: new
        address')

    subject.update(1, address: 'new address')
  end
end
```

In the sampled web applications only the first two kinds of mocks – stubs and doubles – are used, probably because verified mocks have been added only recently.

Compared to the three available mock types in Rspec, role–based have two benefits:

1. Unlike stubs and verified mocks they are not tied to a concrete implementation but only to an interface.

2. They cannot get out of sync like doubles do because the collaborators are checked to conform to the defined role.

This comes at the cost of maintaining roles as an explicit interface definition in the tests. But since interfaces are less likely to change than concrete implementations, the maintenance cost of roles is low.

2.5 CONCLUSION

3 SYSTEM MODEL

3.1 INTRODUCTION

3.2 REQUIREMENTS FOR ROLE–BASED MOCKS

defining roles checking conformance – method names, arity, instance and class methods mocking roles – method call expectations, call undefined methods, arity, define expected arguments, fake return values

Reflection

Role–based mocks can be seen as a contract between tests and classes and the guarantee, that they stick this contract. Let's have a look at the functional requirements from the a practical perspective:

In the design phase following a test–driven development approach, a developer writes a test for the functionality and the behavior of a new User class, that should send a notification message, when it is suspended. This test is written before the actual User class is implemented:

Listing 5: An example Rspec test using a double

```ruby
require 'spec_helper'
require 'user'
require 'roles/notifier'

RSpec.describe User, '#suspend!' do
  it 'notifies the console' do
    notifier = role_double("Notifier")

    expect(notifier).to receive(:notify).with("suspended")

    user = User.new(notifier)
    user.suspend!
  end
end
```

While writing the test, it becomes obvious, that the User class has a collaborator for notifications, that is expected to receive the "notify" method call with a message as an argument, when a user is being suspended. This often called "Interface discovery", where the interface of classes is discovered through the tests of classes, that access this interface. The collaborator can be any class implementing this method with one message argument, it could be a notifier, that sends an email or write a log message, but what the notifier does with the message is not relevant to the user class. To isolate the user class from a concrete implementation of a notifier, a mock, that represents a notifier role, is used (`role_double("Notifier")`). This step makes the User class and its test independent of outside changes – for example the notifier in the system could be altered from a log notifier to an email notifier, the User class and the test would not need to be changed as long as the new notifier sticks to the same interface.

3.3 DESIGN AND IMPLEMENTATION

3.4 CONCLUSION

4 RESEARCH AND ANALYSIS

4.1 INTRODUCTION

4.2 METHODOLOGY

To measure the effect of role–based mocks on maintainability, this study will use a code metric derived from the Change Dependency between Classes (CDBC) attribute presented in Hitz and Montazeri (1995). It determines the potential amount of needed maintenance work in Client Class CC, when a Server Class SC changes.

$$A = \sum_{\substack{\text{accesses } i \text{ to}\\ \text{implementation}}} \alpha_i + (1-k) \times \sum_{\substack{\text{accesses } i \text{ to}\\ \text{interface}}} \alpha_i$$

$$CDBC(CC, SC) = \min(n, A)$$

where n is the number of methods in the CC, that are potentially affected by a change to the Server Class SC and $0 < k < 1$ is the Interface Stability Factor, that grades the interface's stability – stable interfaces like those in a third party library, that are unlikely to change have a low k value.

In terms of test files, the client class and server class are the Test File (TF) and the SUT's collaborator classes ($SUTC$) respectively. In order to analyze the change dependency between a Test File and all the SUT's collaborators, the factor n can be dropped because test files are not Object–Oriented classes with methods but rather procedural scripts, that are run sequentially. For the purpose of this study, it is not interesting to calculate the *potential* amount of follow–up work, as with the CDBC, but rather the *effective* amount. Therefore it is possible to calculate this change dependency by simply counting the calls to the methods of $SUTC$ in the test file. This modified change dependency attribute – the Change Dependency between Collaborator Classes and Test (CDCT) – can be expressed as follows:

$$CDCT(TF, SUTC) = \sum_{impl} \alpha_{impl} + (1-k) \times \sum_{if} \alpha_{if}$$

where α_{impl} are the accesses to methods on concrete implementations of $SUTC$, $0 < k < 1$ is the Interface Stability Factor and α_{if} are the accesses to interface mocks of $SUTC$.

For the calculation of the average CDCT ($ACDCT$) for a large array of test files, we can assume that the Interface Stability Factor $0 < k < 1$ has the value of 0.5 on average. Thus the $ACDCT$ can be calculated as follows:

$$ACDCT(TF, SUTC) = \frac{\sum_n \sum_{impl} \alpha_{impl} + 0.5 \times \sum_n \sum_{if} \alpha_{if}}{n}$$

where n is the number of test files.

To evaluate the effect of role–based mocks on the change dependency of unit tests in web applications, this study will sample test files from real–world web applications. The average CDCT for these test files is calculated twice by counting all calls to implementation methods and to mock methods before and after the application of role–based mocks.

Because of the success of the open source movement in recent years and popularity the source code hosting platform Github, which is free for open source projects (and is itself mainly written in Ruby), it is easy to get access to the source code of many Ruby web projects. For this study 68 Rspec test files from 8 popular open source Ruby web applications have been sampled.

4.3 ANALYSIS

The analysis of the sampled Rspec test files showed that much more tests in the wild use stubbing of method on concrete classes, than interface mocks (Table 1), which results in high maintenance costs.

This can be shown by the average CDCT:

$$\begin{aligned} ACDCT(TF, SUTC) &= \frac{281 + 0.5 \times 149}{68} \\ &= 5.227941176 \end{aligned}$$

So on average, the test files of the sampled, popular Ruby test files have a change dependency on the SUT's collaborator's of 5.227941176.

	SUM	AVERAGE
Calls to Implementation Method	281	4.13235294117647
Calls to Interface Mock Method	149	2.19117647058824

Table 1: Method calls in 68 test files of 8 ruby web projects

When all accesses to concrete implementations of SUTC are replaced by checked role–based mocks, the change dependency improves:

$$\begin{aligned}\mathsf{ACDCT}(\mathsf{TF}, \mathsf{SUTC}) &= \frac{0 + 0.5 \times 430}{68} \\ &= 3.161764706\end{aligned}$$

This clearly shows, that role–based mocks could improve the change dependency and therefore decrease coupling and increase maintainability of software tests for web applications in an interpreted language by better isolating tests from collaborators.

But increased maintainability is not the only advantage of role–based mocks – the also ensure that the interface of the mocks and the concrete classes implementing the roles never get out of sync.

4.4 CONCLUSION

5 CONCLUSION AND FUTURE WORK

Future Work: contract based testing

APPENDIX A

A.1 COUPLING IN TEST FILES OF OPEN SOURCE RUBY WEBAPPLICATIONS

Number in column *S* count calls to stubbed methods of concrete implementations.
Numbers in column *M* count calls to methods of unchecked interface mocks.
Column *C* holds the original CDCT value.
The CDCT value after the application of role–based mocks is shown in column *CI*

GITLAB (GitLab B.V., 2014)

TEST FILES	S	M	C	CI
spec/lib/gitlab/ldap/ldap_adapter_spec.rb	0	3	1.5	1.5
spec/lib/gitlab/ldap/ldap_user_auth_spec.rb	11	0	11	5.5
spec/services/projects/transfer_service_spec.rb	3	0	3	1.5
spec/services/notification_service_spec.rb	26	0	26	13
spec/lib/gitlab/ldap/ldap_access_spec.rb	4	0	4	2
spec/models/gemnasium_service_spec.rb	1	0	1	0.5
spec/services/projects/update_service_spec.rb	1	1	1.5	1
spec/models/milestone_spec.rb	3	0	3	1.5
spec/lib/extracts_path_spec.rb	0	3	1.5	1.5
spec/services/fork_service_spec.rb	0	1	0.5	0.5
spec/models/merge_request_spec.rb	3	0	3	1.5
spec/lib/oauth_spec.rb	0	4	2	2
spec/services/git_push_service_spec.rb	18	0	18	9
spec/lib/gitlab/backend/shell_spec.rb	1	1	1.5	1
spec/models/forked_project_link_spec.rb	0	1	0.5	0.5

COPYCOPTER (Thoughtbot, Inc., 2014)

TEST FILES	S	M	C	CI
spec/models/project_spec.rb	3	0	3	1.5

DISCOURSE (Civilized Discourse Construction Kit, Inc., 2014)

TEST FILES	S	M	C	CI
spec/components/cooked_post_processor_spec.rb	27	0	27	13.5
spec/components/url_helper_spec.rb	5	3	6.5	4
spec/components/discourse_updates_spec.rb	4	0	4	2
spec/services/group_message_spec.rb	11	0	11	5.5
spec/services/user_updater_spec.rb	3	2	4	2.5
spec/services/user_blocker_spec.rb	8	0	8	4

REFINERY CMS (Resolve Digital, 2014)

TEST FILES	S	M	C	CI
pages/spec/lib/refinery/pages/url_spec.rb	5	6	8	5.5
pages/spec/presenters/refinery/pages/content_presenter_spec.rb	0	15	7.5	7.5
resources/spec/models/refinery/resource_spec.rb	1	0	1	0.5

FAT-FREE CRM (Dvorkin, 2014)

TEST FILES	S	M	C	CI
spec/models/polymorphic/task_spec.rb	1	0	1	0.5
spec/models/entities/opportunity_spec.rb	1	0	1	0.5
spec/models/users/user_spec.rb	2	0	2	1
spec/models/users/abilities/user_ability_spec.rb	1	0	1	0.5
spec/mailers/subscription_mailer_spec.rb	5	0	5	2.5
spec/models/fields/field_spec.rb	0	1	0.5	0.5
spec/lib/mail_processor/dropbox_spec.rb	3	0	3	1.5
spec/models/fields/custom_field_pair_spec.rb	0	4	2	2
spec/lib/secret_token_generator_spec.rb	9	0	9	4.5
spec/lib/mail_processor/base_spec.rb	1	19	10.5	10
spec/models/fields/custom_field_spec.rb	9	2	10	5.5
spec/models/observers/entity_observer_spec.rb	0	9	4.5	4.5
spec/lib/fields_spec.rb	2	2	3	2
spec/models/fields/custom_field_date_pair_spec.rb	0	10	5	5

FULCRUM (Locke, 2014)

TEST FILES	S	M	C	CI
spec/models/story_observer_spec.rb	17	0	17	8.5
spec/models/note_spec.rb	8	0	8	4
spec/models/story_spec.rb	2	0	2	1
spec/models/project_spec.rb	1	0	1	0.5
spec/models/note_spec.rb	1	1	1.5	1
spec/models/story_observer_spec.rb	4	2	5	3

ERRBIT (Errbit Team, 2014)

TEST FILES	S	M	C	CI
spec/models/notification_service/campfire_service_spec.rb	0	1	0.5	0.5
spec/models/notification_service/hoiio_service_spec.rb	0	1	0.5	0.5
spec/models/notification_service/pushover_service_spec.rb	0	1	0.5	0.5
spec/models/notification_service/hipchat_service_spec.rb	0	2	1	1
spec/models/notification_service/gtalk_service_spec.rb	17	12	23	14.5
spec/interactors/problem_destroy_spec.rb	4	0	4	2
spec/models/problem_spec.rb	1	0	1	0.5
spec/models/issue_trackers/fogbugz_tracker_spec.rb	0	2	1	1
spec/models/error_report_spec.rb	0	1	0.5	0.5
spec/models/notice_observer_spec.rb	12	0	12	6
spec/models/notification_service/hubot_service_spec.rb	1	0	1	0.5
spec/models/notification_service/webhook_service_spec.rb	1	0	1	0.5
spec/models/notification_service/flowdock_service_spec.rb	1	0	1	0.5
spec/interactors/issue_creation_spec.rb	2	0	2	1
spec/models/notification_service/slack_service_spec.rb	2	0	2	1

SQUARESQUASH (Square, Inc., 2014)

TEST FILES	S	M	C	CI
spec/controllers/sessions_controller_spec.rb	1	4	3	2.5
spec/lib/known_revision_validator_spec.rb	0	5	2.5	2.5
spec/models/deploy_spec.rb	0	1	0.5	0.5
spec/controllers/additions/ldap_authentication_helpers_spec.rb	15	24	27	19.5
spec/controllers/api/v1_controller_spec.rb	0	1	0.5	0.5
spec/models/occurrence_spec.rb	16	4	18	10
spec/lib/service/pager_duty_spec.rb	2	0	2	1
spec/lib/workers/occurrences_worker_spec.rb	1	0	1	0.5

BIBLIOGRAPHY

Civilized Discourse Construction Kit, Inc. Discourse, 08 2014. URL https://github.com/discourse/discourse. (Cited on page 16.)

Michael Dvorkin. Fat free crm, 08 2014. URL https://github.com/fatfreecrm/fat_free_crm. (Cited on page 16.)

Errbit Team. Errbit, 08 2014. URL https://github.com/errbit/errbit. (Cited on page 17.)

GitLab B.V. Gitlab, 08 2014. URL https://github.com/gitlabhq/gitlabhq. (Cited on page 15.)

Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, volume 50, pages 75–76, 1995. (Cited on page 9.)

Malcolm Locke. Fulcrum, 08 2014. URL https://github.com/malclocke/fulcrum. (Cited on page 17.)

Resolve Digital. Refinery cms, 08 2014. URL https://github.com/refinery/refinerycms. (Cited on page 16.)

Square, Inc. Squaresquash web client, 08 2014. URL https://github.com/SquareSquash/web. (Cited on page 18.)

Thoughtbot, Inc. Copycopter server, 08 2014. URL https://github.com/copycopter/copycopter-server. (Cited on page 15.)

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`http://code.google.com/p/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`

DECLARATION

I hereby certify, that this material which I now submit for assessment leading to the award of Master of Science in Web Technology is entirely my own work and has not been taken from the work of others—save and to the extent that such work has been cited and acknowledged within the text of my work.

Dublin, August 10, 2014

Christian Treppo