zeroflag / BabyMock2

PUBLIC

5 Watch this project

Overview Source Commits Contributors Watchers

Project infos

License	MIT
Tags	mock,tdd,mocking,unit testing
Creation date	Sat Dec 21 2013
Website	

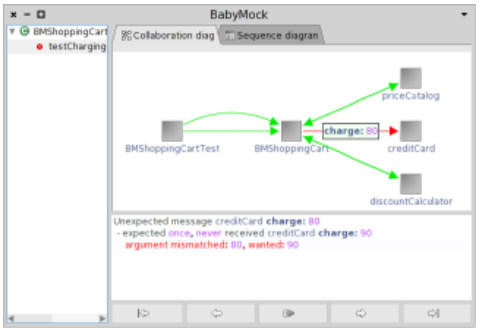
Monticello registration

```
MCHttpRepository
    location: 'http://smalltalkhub.com/mc/zeroflag/BabyMock2/main'
    user: ''
    password: ''
```

About BabyMock2

BabyMock, a visual mock object library

BabyMock is a visual mock object library for Pharo Smalltalk, that supports test-driven development. Like other mock object libraries, BabyMock lets you design and test the interactions between the objects in your programs. BabyMock makes these interactions visible, which can help you to get a good mental picture about the object relationships and the inter-object protocol.



BabyMock requires at least Pharo 3.0. You can use this site for reporting issues.

Getting started

Installation

You can install BabyMock directly from the Configuration browser by selecting *Tools/Configuration browser/BabyMock2/Install stable version*. Or alternatively you can execute the following script.

```
Gofer new
    smalltalkhubUser: 'zeroflag' project: 'BabyMock2';
    package: 'ConfigurationOfBabyMock2';
    load.

((Smalltalk at: #ConfigurationOfBabyMock2) project version: '2.0') load.
```

Write a unit test case

The recommended way to use BabyMock is to extend your test from BabyMock2TestCase, which is a subclass of the SUnit TestCase. After that you can run your test like any other SUnit based unit test.

```
BabyMock2TestCase subclass: #PublisherTest
    instanceVariableNames: ''
```

```
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Demo'
```

We are going to write a test for a simple publisher/subscriber system. Subscribers can be registered to the publisher in order to receive notifications about events. We want to test the publisher, but to do so we need a mocked subscriber in order to check the outgoing messages coming from the publisher.

```
PublisherTest>>testNotifiesSubscriberAboutAnEvent
    [...]
```

Creating mocks

Mocks can be created by sending a *mock:* message to the *protocol* object with an arbitrary name string. The protocol is an instance variable in *BabyMock2TestCase*, and it is used to define expectations and create mocks.

```
subscriber := protocol mock: 'subscriber'.
```

Testing the interaction

The inter-object protocol between the publisher and the subscriber is fairly simple. If we publish an event using the publisher, it will notify all the registered subscribers. These outgoing messages are what we are interested in.

In this testcase we are registering one subscriber mock to the publisher, and we expect the publisher to send the *eventReceived:* message with the correct argument to the subscriber mock.

```
| event publisher subscriber |

event := 'test event'.
publisher := Publisher new.
subscriber := protocol mock: 'subscriber'.
publisher addSubscriber: subscriber.

protocol describe
    once: subscriber recv: #eventReceived:; with: event.

publisher publish: event.
```

We start describing the inter-object protocol by sending the *describe* message to the *protocol* object, then we continue defining the expectations on the mocks.

The test will fail:

- if the *eventReceived* message was not sent to the subscriber during the test
- if the *eventReceived* message was sent more than once
- if the argument of the message was different
- if an other, unexpected message was sent to the subscriber

Note: You can have as many *protocol describe* block as you want.

Visual mode

To enable the visual mode send a *visualize:* message to the *protocol* with the object under test as a parameter.

```
protocol visualize: publisher.
```

You can put this row in the *setUp* method which will affect all tests, or at the *beginning* of the test method you want to visualize.

BabyMock intercepts the messages between the test, the object under test and the mocks. After you execute a unit test, it will display an animation about the interactions.

Arrows with different colors indicate expected, unexpected or missing messages. An allowed or expected message is represented by a green arrow. Unexpected messages are red, and expected but missing messages are gray. A yellow arrow indicates an Exception.

Watch this video to get an idea how does it look like in practice.

Usage

Syntax overview

First thing you need to define on an expectation is its cardinality. Then you can append further restrictions (arguments, states) by cascading other messages.

Cardinalities

Expecting messages exactly *n* times

```
protocol describe
    once:  mock recv: #message1;
    twice: mock recv: #message2;
    exactly: 3 times: mock recv: #message3.
```

Expecting messages at least n times, or at most n times

```
protocol describe
    atMost:  2 times: mock recv: #message1;
    atLeast: 1 times: mock recv: #message2;
    atLeast: 1 atMost: 3 times: mock recv: #message3.
```

Allowing messages any number of times

Use the *allow:recv:* when an exact cardinality of a message is not important. Usually you want to use *allow:recv:* in case of query messages that return values, and *once:recv:*, *twice:recv:*, etc. in case of command messages that do side effects.

```
protocol describe
    allow: mock recv: #message.
```

If you don't specify any return value, nil will be returned.

Explicitly specifying an unexpected message

```
protocol describe
    never: mock recv: #message.
```

This expectation is equivalent of not having it at all. This is used to make tests more explicit and so easier to understand.

Returning values

```
protocol describe
    allow: mock recv: #theUltimateQuestion; => 42.
```

If you send the *theUltimateQuestion* to the mock object, 42 will be returned.

Consecutive answers

When you want to answer different values to the same message, you can do so by cascading the => message.

```
protocol describe
    allow: mock recv: #message;
    => 1;
    => 2;
    => 3.

mock message. "-> prints 1"
mock message. "-> prints 2"
mock message. "-> prints 3"
mock message. "-> prints 3"
```

After the third message the mock will keep returning the the last value.

Matching parameter values

```
protocol describe
    allow: calculator recv: #add:to:;
    with: 1 and: 2;
    => 3;
    allow: calculator recv: #add:to:;
    with: 3 and: 4;
    => 7.

calculator add: 1 to: 2. "-> prints 3".
calculator add: 3 to: 4. "-> prints 7"
```

Blocks as argument matchers

Any block that returns booleans, and has the correct number of arguments can be used as an argument matcher.

```
protocol describe
    allow: calculator recv: #sqrt:;
    with: [:number | number < 0];
    signals: MathDomainError.

calculator sqrt: -1. "-> will signal MathDomainError"
```

You can use different blocks for each arguments, or one block which has the same number of arguments as the method has.

```
protocol describe
    allow: calculator recv: #subtract:from:;
    with: [:a :b | a > b];
    signals: NegativeNotSupported.

calculator subtract: 3 from: 1. "-> will signal NegativeNotSupported"
```

```
protocol describe
    allow: calculator recv: #add:to:;
    with: [:a | a > 10] and: [:b | b < 20];
    => 'something'.
```

Allowing any argument

```
protocol describe
    allow: calculator recv: #add:to:;
    with: Anything and: 3;
    => 4.

calculator add: 'apple' to: 3. "this will return 4"
```

Ignoring an irrelevant mock object

```
protocol describe
   ignore: mock.
```

You can send any message to this mock object and the default answer will be nil.

You can achieve a similar effect by using *Anything* as a message.

```
protocol describe
   allow: mock recv: Anything.
```

Side effects

Occasionally you may want to do some side effect after a mock received a message. You can pass a block to the => message that will be executed.

```
protocol describe
   allow: mock recv: #message;
   => [ 'printing out side effect' traceCr ].

mock message. "this will print out the text"
```

If you want to return the block instead then you'll need to put a nested block inside an other one.

```
protocol describe
   allow: mock recv: #message;
   => [ [ 'this block will be returned' traceCr ] ].

mock message value. "this will print out the text"
```

Accessing to the message arguments from the return block

```
| items |
items := OrderedCollection new.

protocol describe
   allow: mock recv: #add:;
   => [:item | items add: item ].

mock add: 1.
mock add: 2.
items inspect. "items will contain 1 and 2"
```

Using states

States are handy if you want to bring your object under test in a particular state, and check whether an action was take in the correct state, without accessing the object's internals.

```
state := protocol states: 'machine state' startsAs: #ready.

protocol describe
    once: vendingMachine recv: #insertCoin:; with: 1 dollar;
    when: state is: #ready; then: state is: #hasCoin;
    once: vendingMachine recv: #press:; with: 2;
    when: state is: #hasCoin;
    once: vendingMachine recv: #press:; with: 5;
    when: state is: #hasCoin.
```

The user of the vending machine could press the button "2" and "5" in any order, as long as the vending machine is in the #hasCoin state, the test will pass. What is important is that the insertCoin should happen first. States are useful for loosening ordering constraints (one event happens first and others follow it in any order).

Bouncing message

If you have a complex expectation, it is recommended to extract it to its own method. For example, instead of this,

```
protocol describe
    allow: priceCatalog recv: #priceOf:; with: #sku1; => 10;
    allow: priceCatalog recv: #priceOf:; with: #sku2; => 15;
```

```
    allow: priceCatalog recv: #priceOf:; with: #sku3; => 20;
    once: creditCard recv: #charge:; with: 45.
```

You can write this.

```
self prices: {#sku1 -> 10. #sku2 -> 30. #sku3 -> 50}.

protocol describe
    once: creditCard recv: #charge:; with: 45.
```

```
ShoppingCartTest>>prices: itemPriceAssocs
    itemPriceAssocs do: [:each |
        protocol describe
            allow: priceCatalog recv: #priceOf:;
            with: each key;
            => each value ]
```

However, you can put the *prices:* inside the expectations block if you want.

```
protocol describe
    prices: {#sku1 -> 10. #sku2 -> 30. #sku3 -> 50};
    once: creditCard recv: #charge:; with: 45.
```

If you send an unknown message to the syntax object, returned by the *protocol describe*, it will send back to the original sender if that responds to it.

Advices

- Don't mock value objects, built-ins, and objects you don't own. If your object under test needs an OrderedCollection, just instantiate a real one and use that one instead of a mock.
- Differentiate query messages from commands, by using the *allow:recv:* in case of the former. Usually you don't need to verify query messages.
- If you have too many mocks, or the protocol is messy then take the opportunity to think about the design.
- Read this paper.

Syntax cheat sheet

```
protocol describe
    once: mock recv: #aSymbol;
      with: 42 and: Anything and: [:x | x > 10];
      when: state is: stateName;
      then: state is: newState;
      => returnValue;
    twice: mock recv: #aSymbol;
      with: [:a :b | a > b];
      => [:a :b | ...];
    allow: mock recv: #aSymbol; signals: Error;
    exactly: 3 times: mock recv: #aSymbol;
    atMost:  3 times: mock recv: #aSymbol;
    atLeast: 1 times: mock recv: #aSymbol;
    atLeast: 1 atMost: 3 times: mock recv: #aSymbol.
```

- Attila Magyar, 2014