Don’t give up on mocking

- Why do people give up?
- Mocking: the big step from classic way of testing
- Let’s take a step back and don’t give up!

by Szczepan Faber (a certified mock tamer)

Interaction testing...

- State testing is asking: „what's your colour, Mr Object?"
- Interaction testing is asking: „Mrs Object, what did you say to Mr Object?"

The language

- The natural language of state testing are assertions
- The natural language of interaction testing is... mocking?

What’s a mock or a stub?

- It is a substitue of the real thing for the purposes of testing

Mocking...

- Is it a design tool for describing messaging patterns between abstract state machines?
- Is it a handy tool which lets me create mocks dynamically?

Giving up...

- The internet says mocking is cool
- Let’s find out why one would give up on mocking!

Why would one give up on mocking?

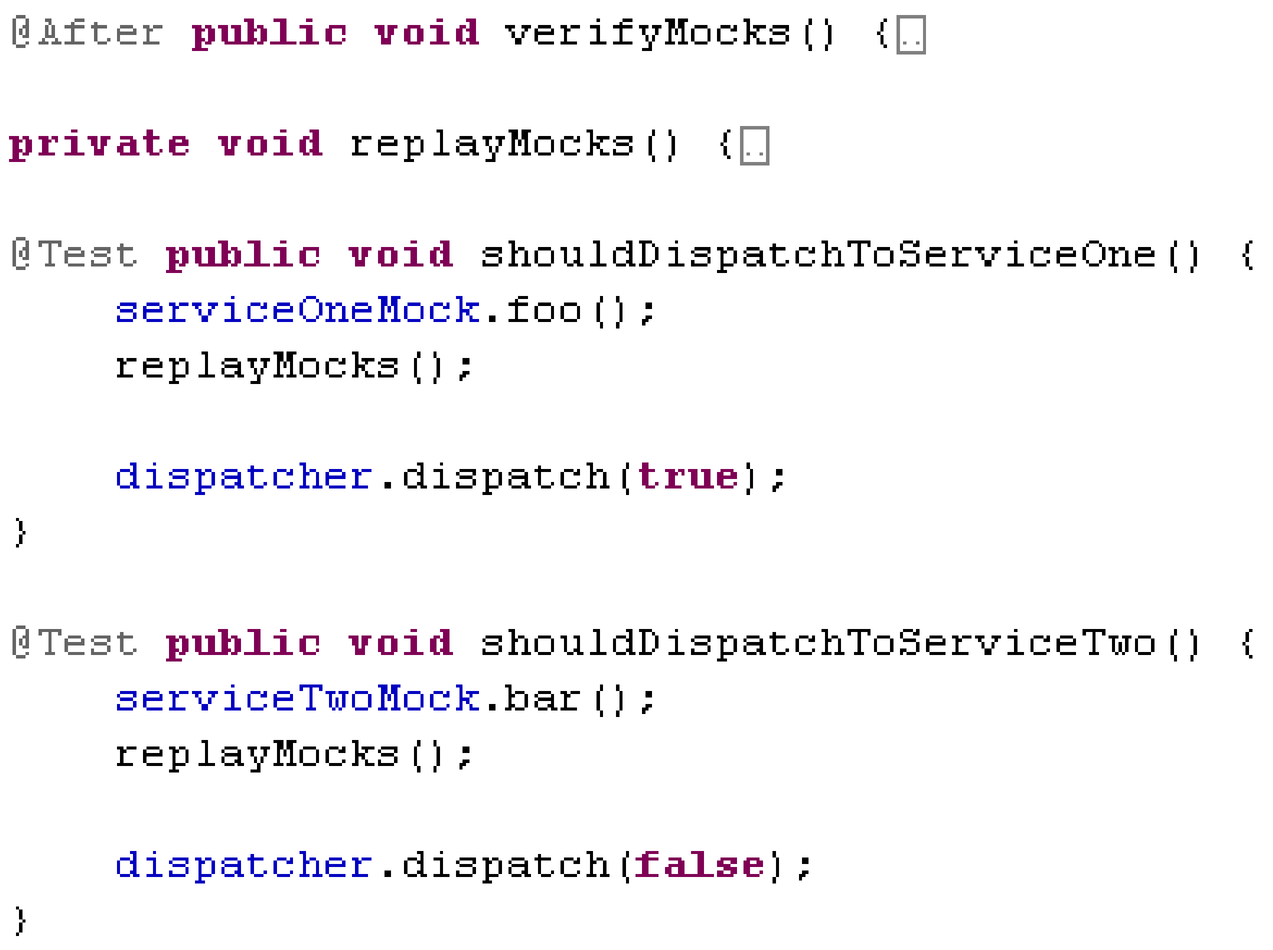
Why would one give up on mocking?

- because aggressive validation makes the tests brittle ☹

The code

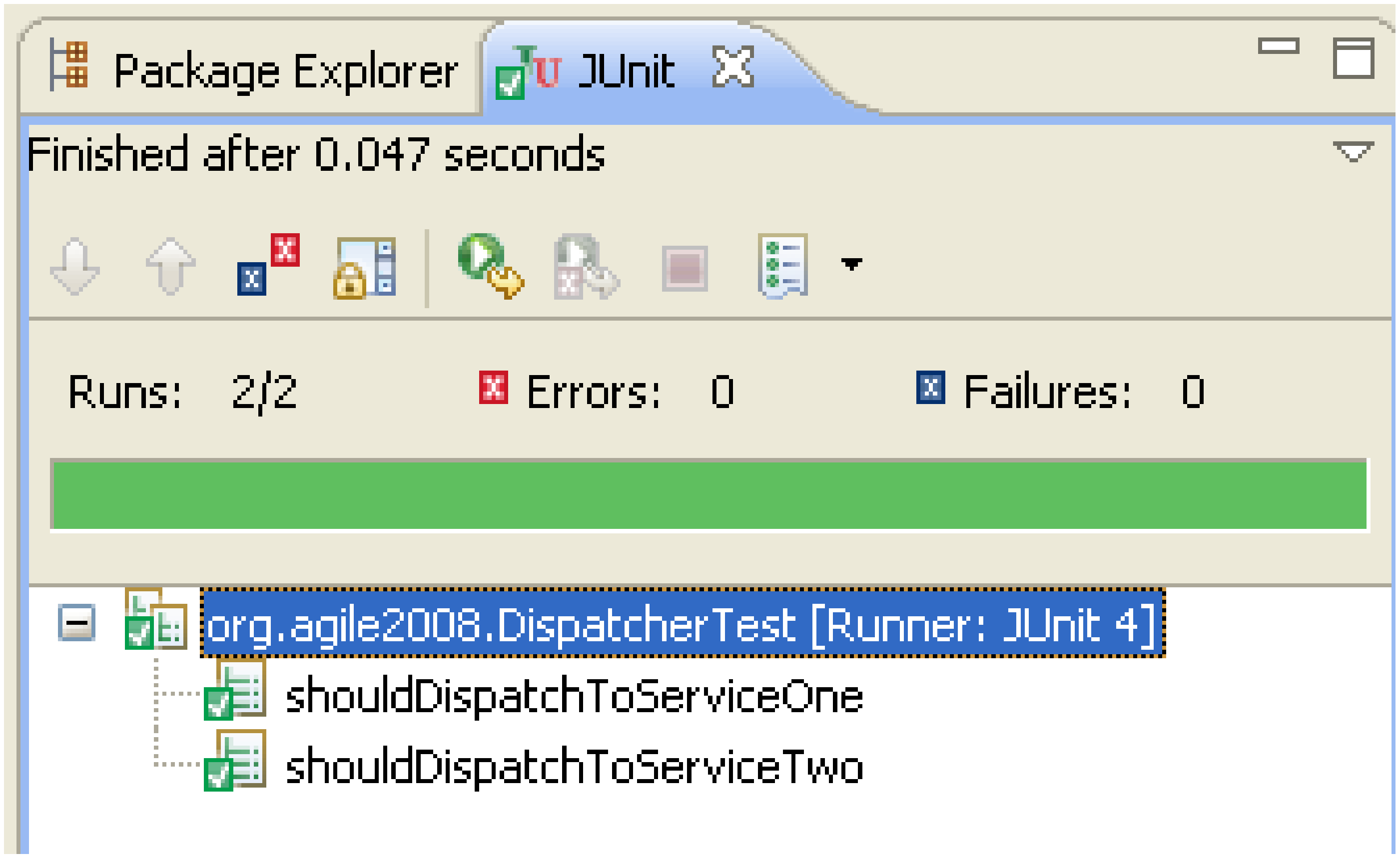
```
public void dispatch(boolean condition) {
    if (condition) {
        serviceOne.foo();
    } else {
        serviceTwo.bar();
    }
}
```

The test



```
@After public void verifyMocks() {⋯

private void replayMocks() {⋯

@Test public void shouldDispatchToServiceOne() {
    serviceOneMock.foo();
    replayMocks();

    dispatcher.dispatch(true);
}

@Test public void shouldDispatchToServiceTwo() {
    serviceTwoMock.bar();
    replayMocks();

    dispatcher.dispatch(false);
}
```

And the lovely green bar



Finished after 0.047 seconds

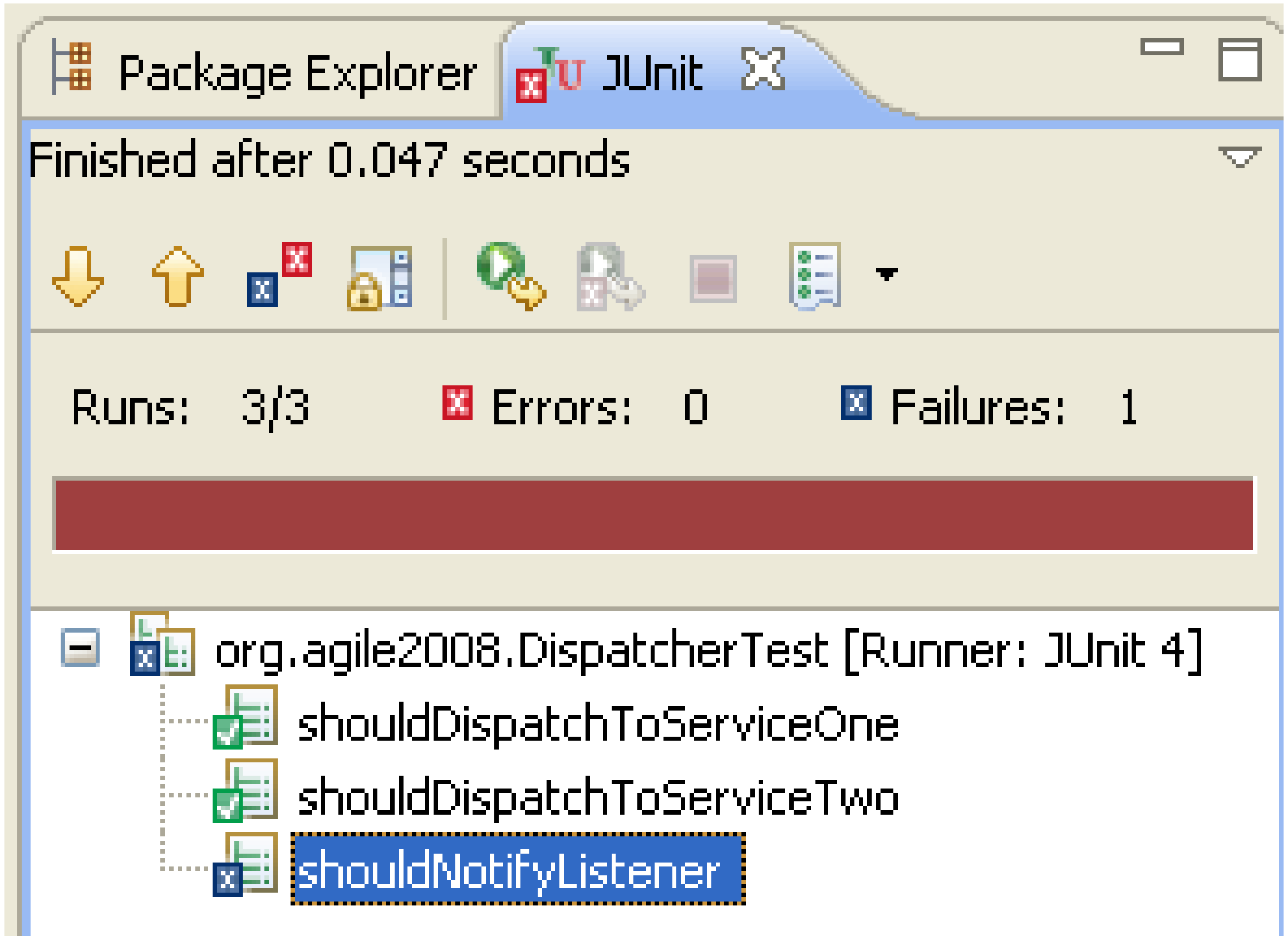
Runs: 2/2 Errors: 0 Failures: 0

```
org.agile2008.DispatcherTest [Runner: JUnit 4]
    shouldDispatchToServiceOne
    shouldDispatchToServiceTwo
```

TDD-ing a new feature (test)

```
@Test public void shouldNotifyListener() {
    listenerMock.notify("dispatched");
    replayMocks();

    dispatcher.dispatch(false);
}
```

The adorable red bar

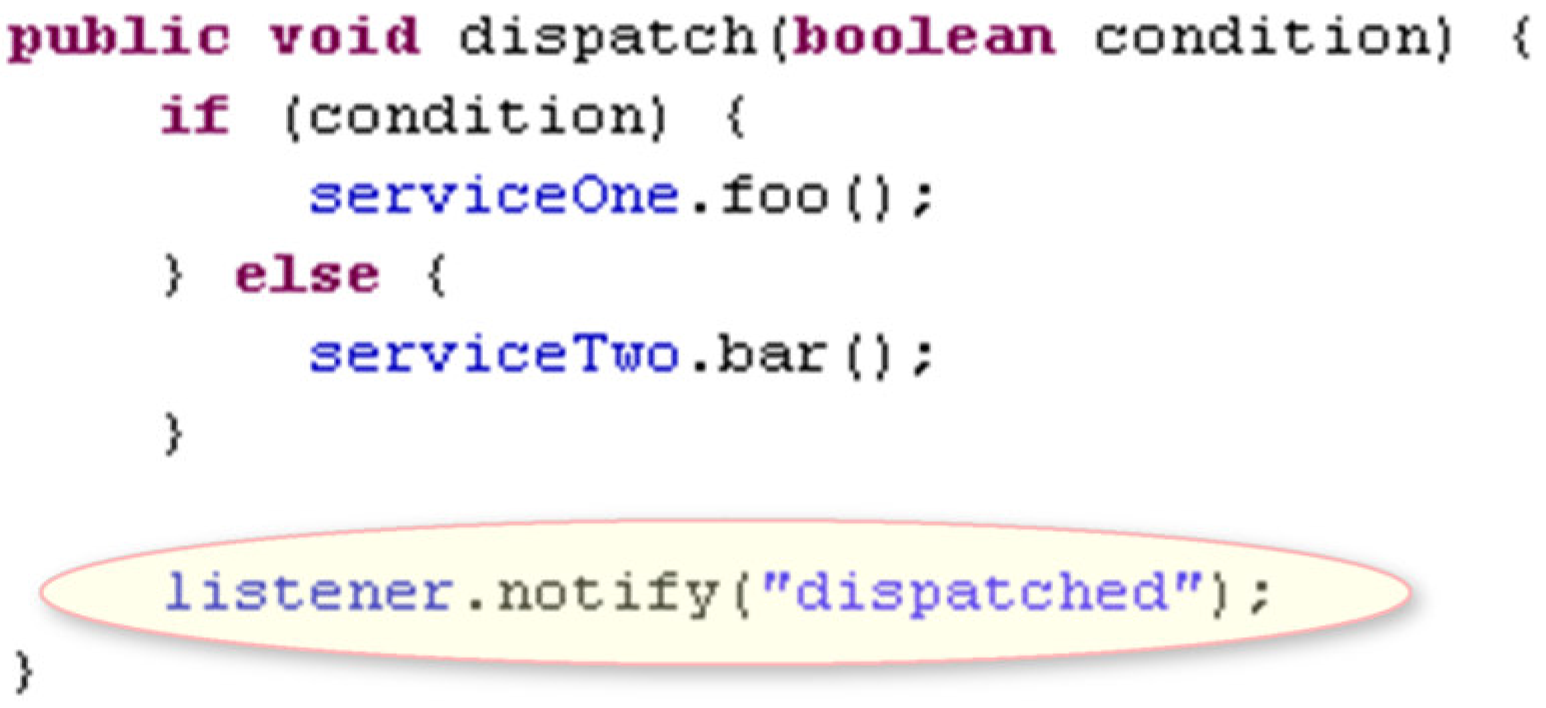


Finished after 0.047 seconds

Runs: 3/3 Errors: 0 Failures: 1

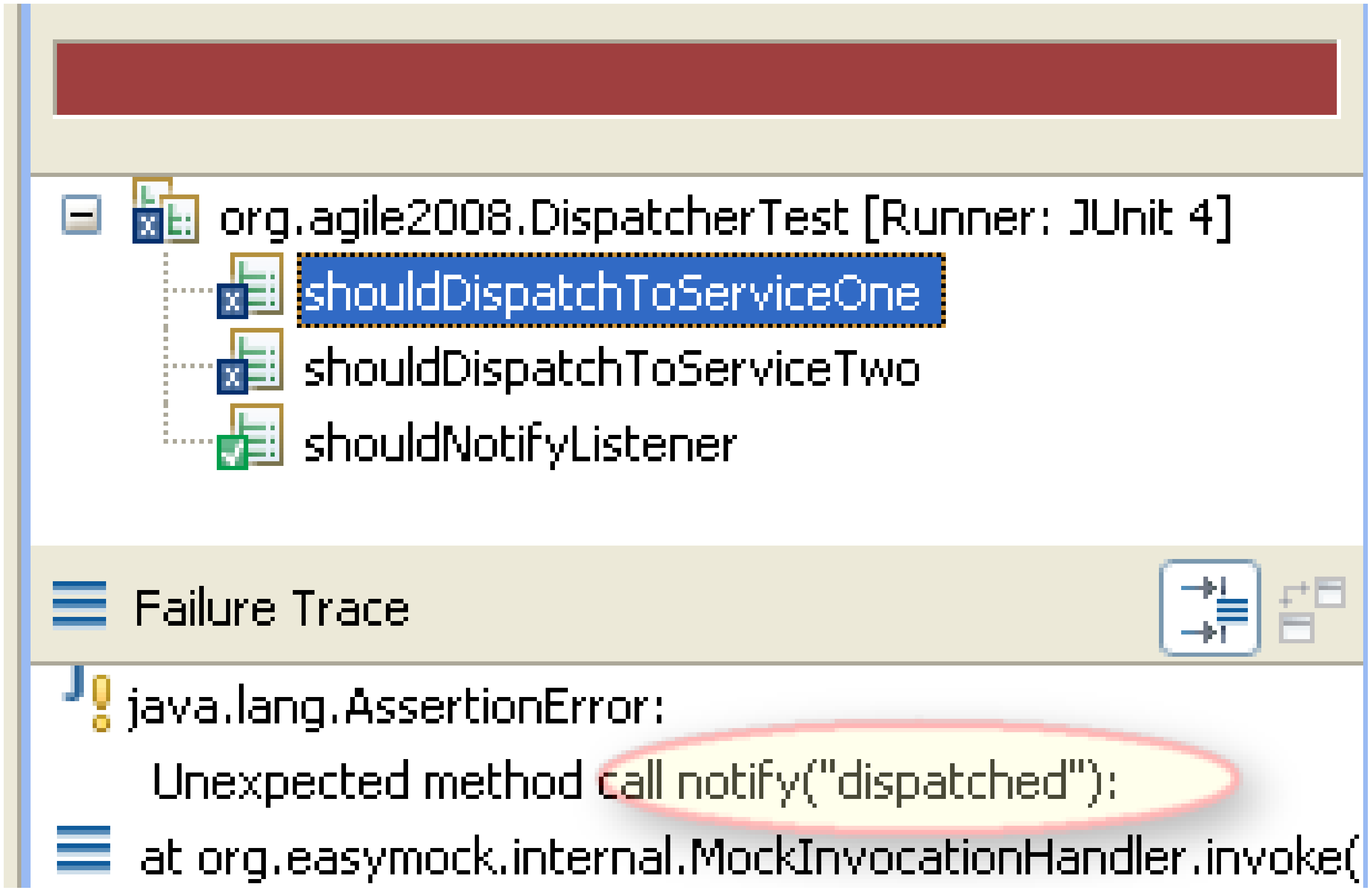
org.agile2008.DispatcherTest [Runner: JUnit 4]
shouldDispatchToServiceOne
shouldDispatchToServiceTwo
shouldNotifyListener

TDD-ing a new feature (code)



```java
public void dispatch(boolean condition) {
    if (condition) {
        serviceOne.foo();
    } else {
        serviceTwo.bar();
    }

    listener.notify("dispatched");
}
```

Whoah? Red bar again?



org.agile2008.DispatcherTest [Runner: JUnit 4]
shouldDispatchToServiceOne
shouldDispatchToServiceTwo
shouldNotifyListener

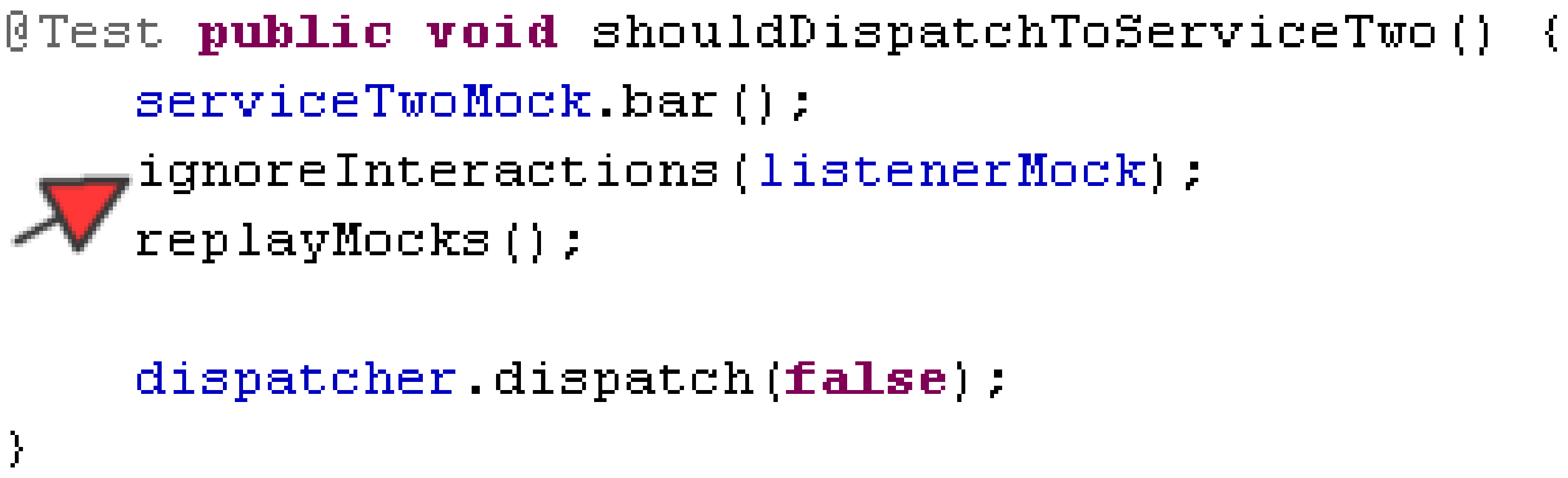
Failure Trace

```
java.lang.AssertionError:
  Unexpected method call notify("dispatched"):
at org.easymock.internal.MockInvocationHandler.invoke(
```

Why would one give up on mocking?

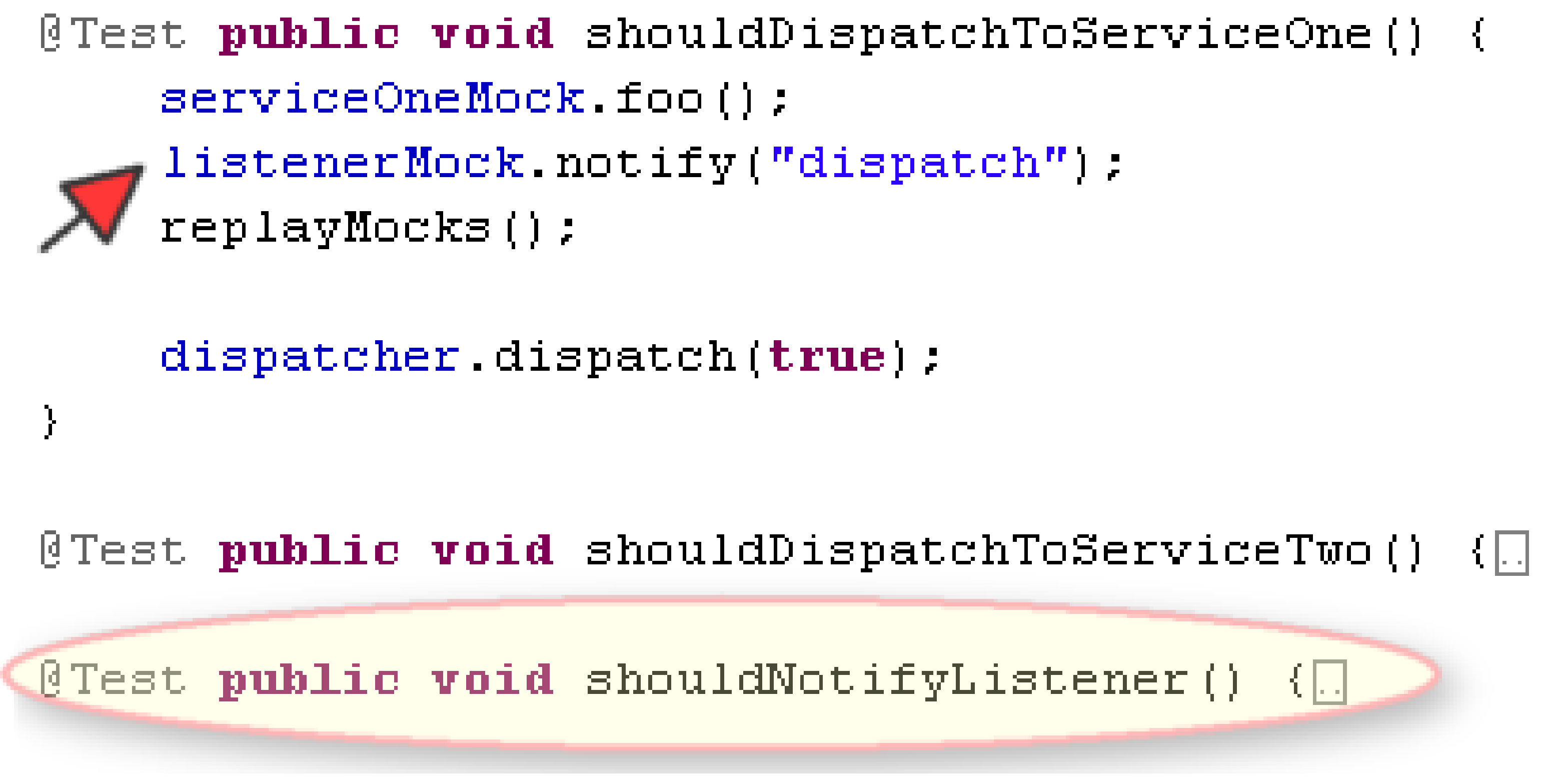
- because I have to fix tests even when the code is not broken:
  - may increase noise
  - may lead to overspecification

Fixing by ignoring interactions



```
@Test public void shouldDispatchToServiceTwo() {
    serviceTwoMock.bar();
    ignoreInteractions(listenerMock);
    replayMocks();

    dispatcher.dispatch(false);
}
```

Fixing by adding required expectation



```
@Test public void shouldDispatchToServiceOne() {
    serviceOneMock.foo();
    listenerMock.notify("dispatch");
    replayMocks();

    dispatcher.dispatch(true);
}

@Test public void shouldDispatchToServiceTwo() {..

@Test public void shouldNotifyListener() {..
```

Why would one give up on mocking?

- What if hand mocks were better?

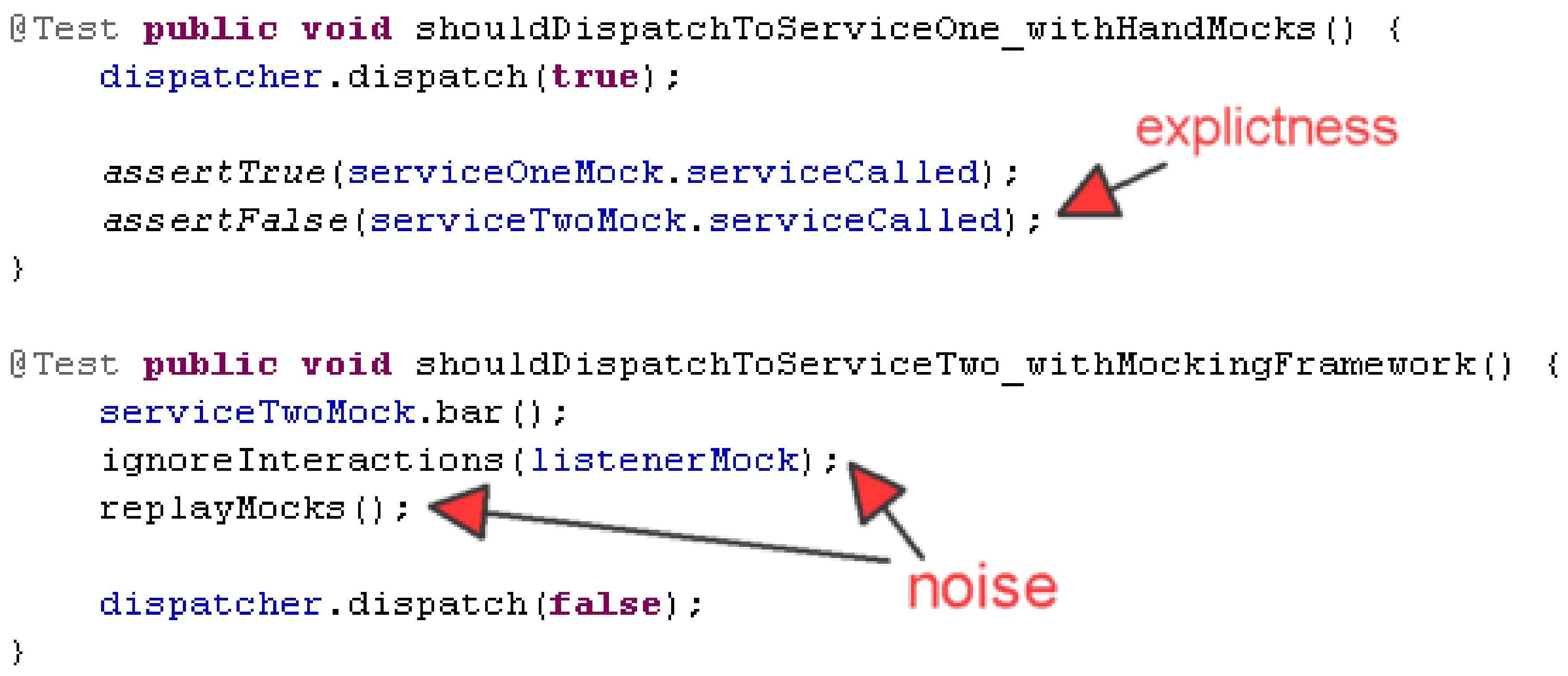
Remember the code?

```
public void dispatch(boolean condition) {
    if (condition) {
        serviceOne.foo();
    } else {
        serviceTwo.bar();
    }

    listener.notify("dispatched");
}
```

Let’s try some hand written mocks

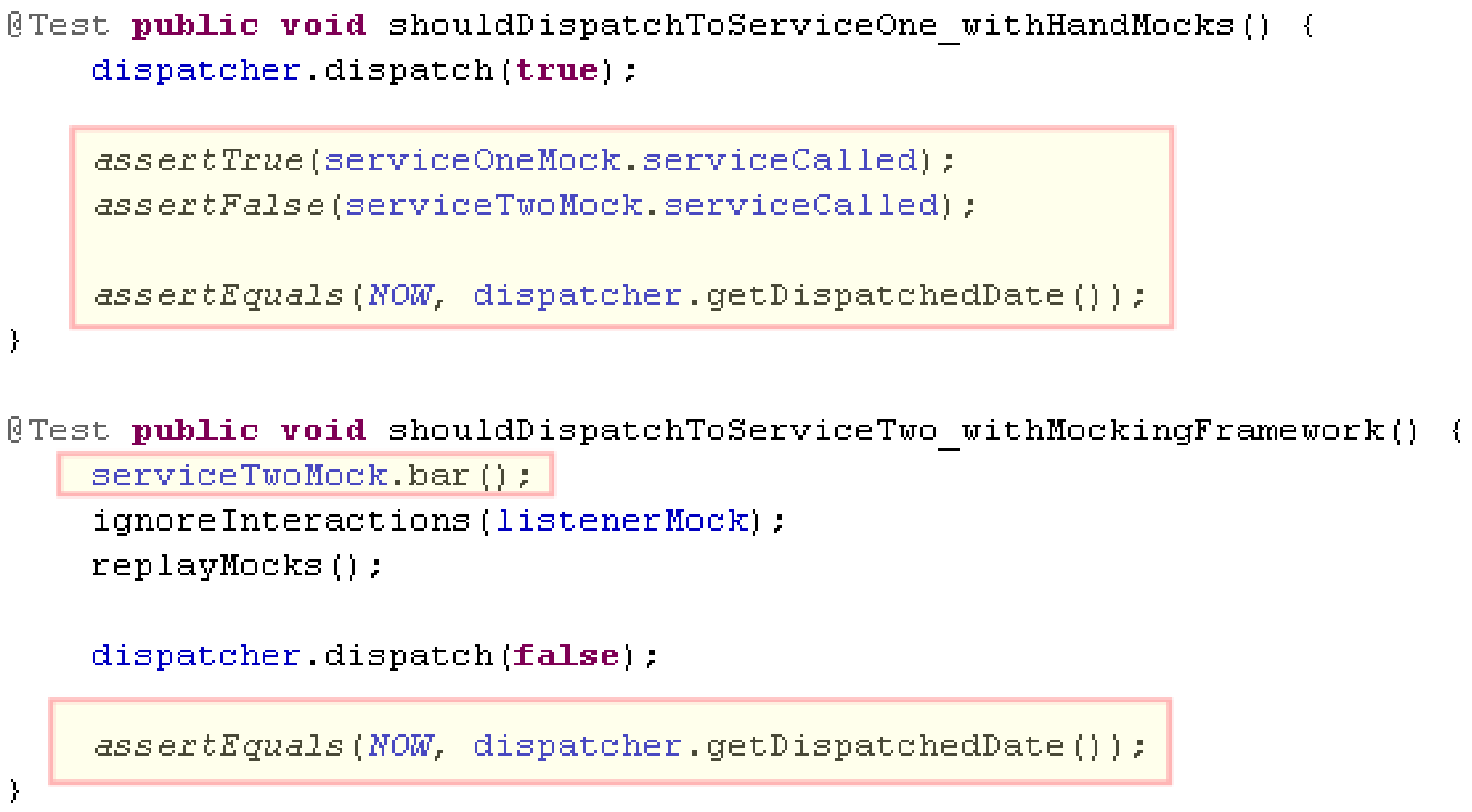
```
public class ListenerMock implements Listener {

    String notifiedWith;

    @Override public void notify(String notification) {
        this.notifiedWith = notification;
    }
}

public class ServiceTwoMock implements ServiceTwo {

    boolean serviceCalled;

    @Override public void bar() {
        serviceCalled = true;
    }
}
```

By hand or with the framework: the essence



```
@Test public void shouldDispatchToServiceOne_withHandMocks() {
    dispatcher.dispatch(true);

    assertTrue(serviceOneMock.serviceCalled);
    assertFalse(serviceTwoMock.serviceCalled);
}

@Test public void shouldDispatchToServiceTwo_withMockingFramework() {
    serviceTwoMock.bar();
    ignoreInteractions(listenerMock);
    replayMocks();

    dispatcher.dispatch(false);
}
```

By hand or with the framework: expectations



```
@Test public void shouldDispatchToServiceOne_withHandMocks() {
    dispatcher.dispatch(true);

    assertTrue(serviceOneMock.serviceCalled);
    assertFalse(serviceTwoMock.serviceCalled);

    assertEquals(NOW, dispatcher.getDispatchedDate());
}

@Test public void shouldDispatchToServiceTwo_withMockingFramework() {
    serviceTwoMock.bar();
    ignoreInteractions(listenerMock);
    replayMocks();

    dispatcher.dispatch(false);

    assertEquals(NOW, dispatcher.getDispatchedDate());
}
```

Complete test

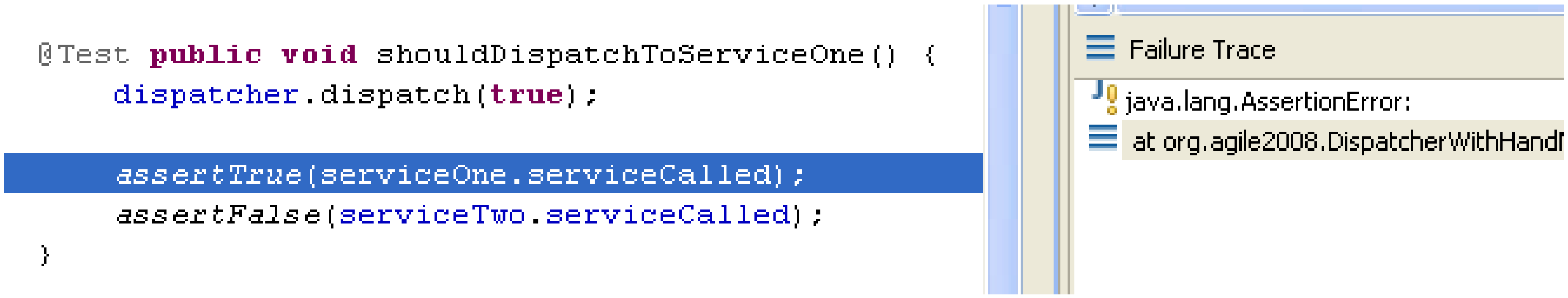
```
@Test public void shouldDispatchToServiceOne() {
    dispatcher.dispatch(true);

    assertTrue(serviceOneMock.serviceCalled);
    assertFalse(serviceTwoMock.serviceCalled);
}

@Test public void shouldDispatchToServiceTwo() {
    dispatcher.dispatch(false);

    assertTrue(serviceTwoMock.serviceCalled);
    assertFalse(serviceOneMock.serviceCalled);
}

@Test public void shouldNotifyListener() {
    dispatcher.dispatch(false);
    assertEquals("dispatched", listenerMock.notifiedWith);
}
```

Why would one give up on mocking?

- Let’s look at the point of failure

Point of failure and hand mocks

Hand mocks show useful stack trace pointing to exact line of code



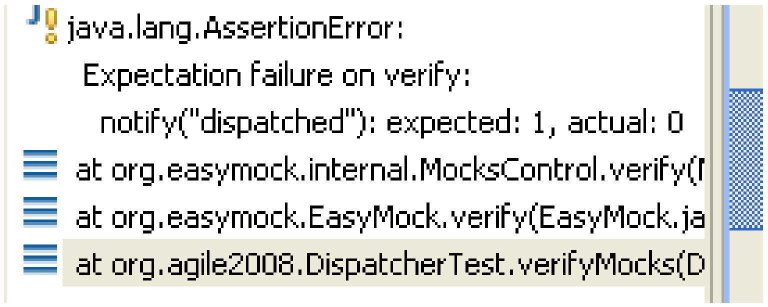
```
@Test public void shouldDispatchToServiceOne() {
    dispatcher.dispatch(true);

    assertTrue(serviceOne.serviceCalled);
    assertFalse(serviceTwo.serviceCalled);
}
```

Failure Trace

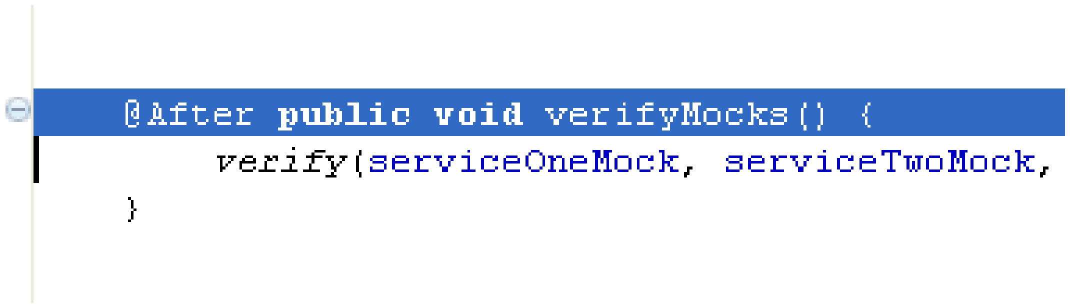
```
java.lang.AssertionError:
at org.agile2008.DispatcherWithHandI
```

When the framework fails on verify()

The exception message which tries to be readable.



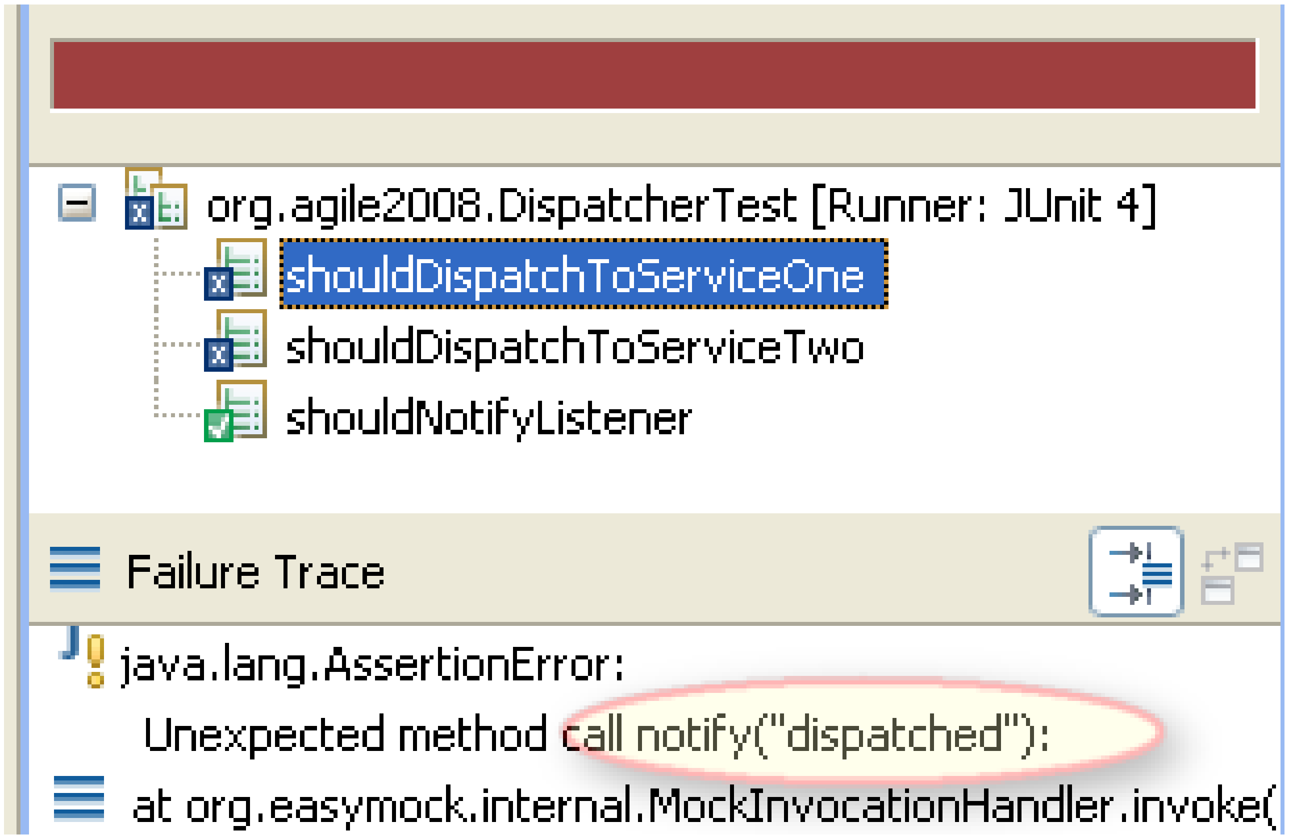
```
java.lang.AssertionError:
  Expectation failure on verify:
    notify("dispatched"): expected: 1, actual: 0
at org.easymock.internal.MocksControl.verify(M
at org.easymock.EasyMock.verify(EasyMock.ja
at org.agile2008.DispatcherTest.verifyMocks(D
```



```
@After public void verifyMocks() {
    verify(serviceOneMock, serviceTwoMock,
}
```

When the framework fails with „Unexpected Interaction!”

Helpful but...



```
org.agile2008.DispatcherTest [Runner: JUnit 4]
    shouldDispatchToServiceOne
    shouldDispatchToServiceTwo
    shouldNotifyListener
```

Failure Trace

```
java.lang.AssertionError:
  Unexpected method call notify("dispatched"):
at org.easymock.internal.MockInvocationHandler.invoke(
```

Ok, now I understand why one would give up on mocking.

- because aggressive validation makes the tests brittle ☹
- because I have to fix tests even when the code is not broken
  - but it can increase noise
  - or lead to overspecification
- because hand-mocks can be considered better:
  - less noisy
  - more natural
  - with better(?) point of failure

Are hand mocks a better option, then?

- Err... not really... hand mocks have different issues.
- Hand mocks bad, mocking framework bad what should I do now?

A taste of Mockito, a Test Spy framework

```
@Test public void shouldDispatchToServiceOne() {
    dispatcher.dispatch(true);

    verify(serviceOneMock).foo();
    verify(serviceTwoMock, never()).bar();
}

@Test public void shouldDispatchToServiceTwo() {
    dispatcher.dispatch(false);

    verify(serviceOneMock, never()).foo();
    verify(serviceTwoMock).bar();
}

@Test public void shouldNotifyListener() {
    dispatcher.dispatch(false);

    verify(listenerMock).notify("dispatched");
}
```

A taste of hand mocks, no framework at all

```
@Test public void shouldDispatchToServiceOne() {
    dispatcher.dispatch(true);

    assertTrue(serviceOneMock.serviceCalled);
    assertFalse(serviceTwoMock.serviceCalled);
}

@Test public void shouldDispatchToServiceTwo() {
    dispatcher.dispatch(false);

    assertTrue(serviceTwoMock.serviceCalled);
    assertFalse(serviceOneMock.serviceCalled);
}

@Test public void shouldNotifyListener() {
    dispatcher.dispatch(false);
    assertEquals("dispatched", listenerMock.notifiedWith);
}
```

Test Spy framework

- ~~because aggressive validation makes the tests brittle ☹~~
- ~~because I have to fix tests even when the code is not broken~~
  - ~~but it can increase noise~~
  - ~~or lead to overspecification~~
- ~~because hand-mocks can be considered better:~~
  - ~~less noisy~~
  - ~~more natural~~
  - ~~with better(?) point of failure~~

Languages, where are your Test Spy frameworks?

- You’ve got plenty of mocking frameworks
  - Java
  - C#
  - Ruby
  - Python
  - JavaScript
- But you’ve got so little Test Spy frameworks
  - Java
  - C#
  - Ruby
  - Python
  - JavaScript

This is what is trendy in the mocking world these days

- Better and better DSLs for describing expectations
- Partial mocking
- Mocking static methods
- Features that solve rare corner cases
- Etc.

Mock objects: the quest for quality

- Does application code quality vary when using different mock libraries (or hand mocks)?
- Does test code quality vary when using different mock libraries (or hand mocks)?
- Can I use different mock libraries in single project?

Mocking in Java

- jMock
- EasyMock
- Mockito

How to verify the method was called?

JMock:

```
context.checking(new Expectations() {{
    one(repository).deleteArticle(article);
}});
```

EasyMock:

```
repositoryMock.deleteArticle(article);
replay(repositoryMock);
```

Mockito:

```
verify(repository).deleteArticle(article);
```

How to tell a method to return a value?

JMock:

```
context.checking(new Expectations() {{
    one(repository).getArticle(headline);
    will(returnValue(article));
}});
```

EasyMock:

```
expect(repositoryMock.getArticle(headline)).andReturn(article);
replay(repositoryMock);
```

Mockito:

```
stub(repository.getArticle(headline)).toReturn(article);
```

How verify the method was not called

JMock:

```
never(repository).dontCallMe();
```

EasyMock:

(always implicit)

Mockito:

```
verify(repository, never()).dontCallMe();
```

Mockito separates stubbing from verification

```
//given
stub(repository.getArticle(headline)).toReturn(article);

//when
manager.deleteByHeadline(headline);

//then
verify(repository).deleteArticle(article);
```

Classic mocking doesn’t separate stubbing from verification

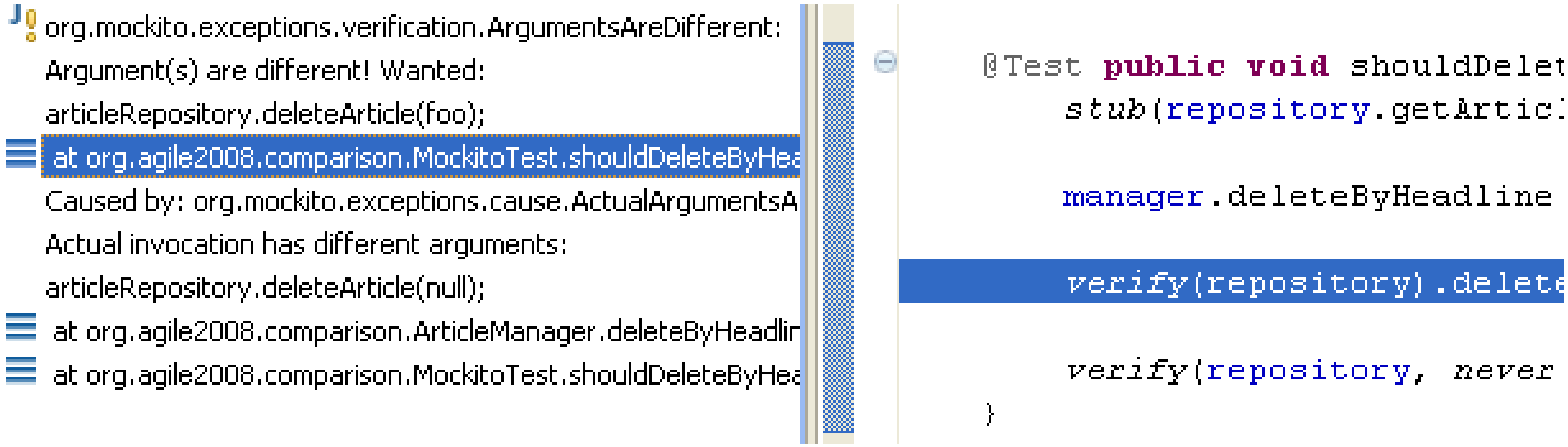
JMock:

```
context.checking(new Expectations() {{
    one(repository).getArticle(headline);
    will(returnValue(article));
    one(repository).deleteArticle(article);
}});

manager.deleteByHeadline(headline);
```

EasyMock:

```
expect(repositoryMock.getArticle(headline)).andReturn(article);
repositoryMock.deleteArticle(article);

replay(repositoryMock);

manager.deleteByHeadline(headline);
```

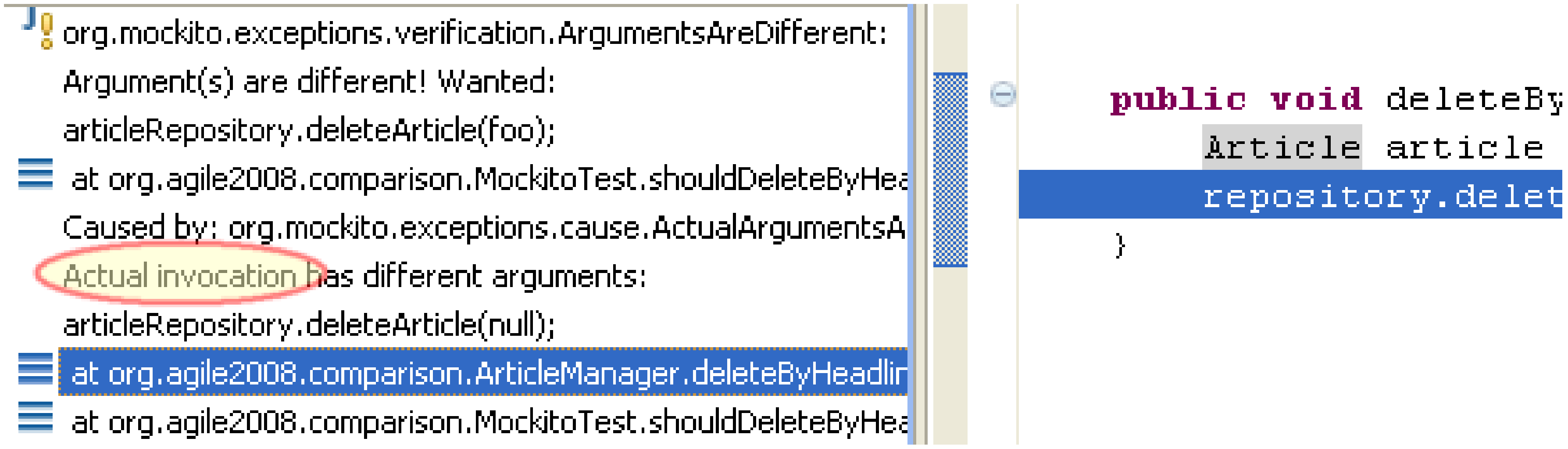
Mockito knows developers read stack trace



```
org.mockito.exceptions.verification.ArgumentsAreDifferent:
Argument(s) are different! Wanted:
articleRepository.deleteArticle(foo);
at org.agile2008.comparison.MockitoTest.shouldDeleteByHea
Caused by: org.mockito.exceptions.cause.ActualArgumentsA
Actual invocation has different arguments:
articleRepository.deleteArticle(null);
at org.agile2008.comparison.ArticleManager.deleteByHeadlir
at org.agile2008.comparison.MockitoTest.shouldDeleteByHea
```

```
@Test public void shouldDelet
    stub(repository.getArtic

    manager.deleteByHeadline

    verify(repository).delete

    verify(repository, never
}
```

Mockito knows developers read stack trace



```
org.mockito.exceptions.verification.ArgumentsAreDifferent:
Argument(s) are different! Wanted:
articleRepository.deleteArticle(foo);
 at org.agile2008.comparison.MockitoTest.shouldDeleteByHea
Caused by: org.mockito.exceptions.cause.ActualArgumentsA
Actual invocation has different arguments:
articleRepository.deleteArticle(null);
 at org.agile2008.comparison.ArticleManager.deleteByHeadlin
 at org.agile2008.comparison.MockitoTest.shouldDeleteByHea
```

```
public void deleteBy
    Article article
    repository.delet
}
```

Mockito is a Test Spy framework

Classic mocking	Spying	Classic testing

expectThis()
expectThat()
run()
verify()
	run()
	verifyThis()
	verifyThat()
		run()
		assertThis()
		assertThat()

Mockito and classic testing are explicit

Classic mocking	**Classic testing and Mockito**
strict by default	loose by default

loose style requires explicit specification:
 	
ignoreInteractions(mock);
	strict style requires explicit specification:
	
	assertNotTrue(something);
	verify(mock, never()).method();

The current era in my project is Mockitozoic!

- jMockozoic ->
- EasyMockozoic ->
- HandMockozoic ->
- Mockitozoic

What’s next?

- jMockozoic ->
- EasyMockozoic ->
- HandMockozoic ->
- Mockitozoic ->
- ?

What I don’t like about Mockito

- a bit inconsistent API:
  - verify(**mock).method()**;
  - stub(**mock.method()**).toReturn(x);
- stubbing voids is different:
  - doThrow(ex).when(mock).method();
- may lead to overmocking because it’s too easy to mock ☺

What users like about Mockito?

- explicit API
- flexible verification
- separation of stubbing and verification
- @Mock annotation
- expectations after exercising

What are the plans for Mockito:

- maintain slip API to promote simple code
- change the stubbing api:
  - instead: stub(mock.getStuff()).toReturn(x);
  - do: when(mock.getStuff()).thenReturn(x);
- spread to other languages (python, c++, C#)

Regards

- jMock guys for inventing mock objects
- EasyMock guys for their innovative syntax
- Gerard Maszeros for sorting out mocking terminology
- Mockito users and contributors for their ideas