Mock Objects

About Mock Objects, a technique for improving the design of code within Test-Driven Development

24 April 2007

Test Smell: Mocking concrete classes

One approach to Interaction Testing is to mock concrete classes rather than interfaces. The technique is to inherit from the class you want to mock and override the methods that will be called within the test, either manually or with any of the mocking frameworks. I think it's a technique that should be used only when you *really* have no other options.

Here's an example of mocking by hand, the test verifies that the music centre starts the CD player at the requested time. Assume that setting the schedule on a `CdPlayer` object involves triggering some behaviour we don't want in the test, so we override the `scheduleToStartAt` and verify afterwards that we've called it with the right argument.

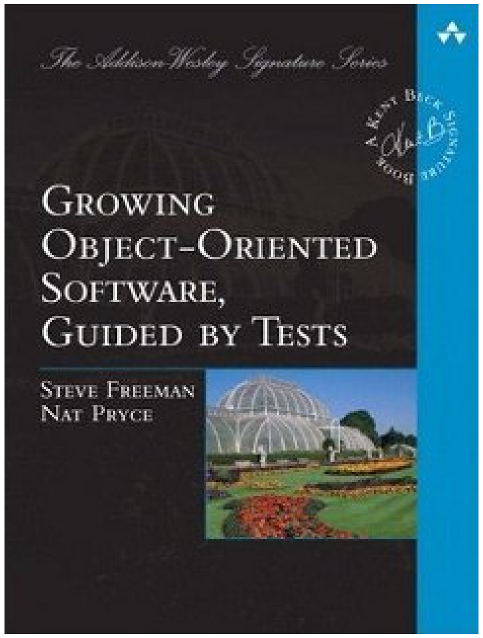
```
public class MusicCentreTest {
  @Test public void startsCdPlayerAtTimeRequested() {
    final MutableTime scheduledTime = new MutableTime();
    CdPlayer player = new CdPlayer() {
      @Override
      public void scheduleToStartAt(Time startTime) {
        scheduledTime.set(startTime);
      }
    }

    MusicCentre centre = new MusicCentre(player);
    centre.startMediaAt(LATER);

    assertEquals(LATER, scheduledTime.get());
  }
}
```

The problem with this approach is that it leaves the relationship between the objects implicit. I hope we've made clear by now that the intention of Test-*Driven* Development with Mock Objects is to discover relationships between objects. If I subclass, there's nothing in the domain code to make such a relationship visible, just methods on an object. This makes it harder to see if the service that supports

Our Book



Our Papers

[Endotesting (XP2000)](#)

[Evolving an Embdedded Domain Specific Language](#)

[Mock Roles, Not Objects](#)

There was an error in this gadget

There was an error in this gadget

this relationship might be relevant elsewhere and I'll have to do the analysis again next time I work with the class. To make the point, here's a possible implementation of `CdPlayer`:

```
public class CdPlayer {
  public void scheduleToStartAt(Time startTime) { …
  public void stop() { …
  public void gotoTrack(int trackNumber) { …
  public void spinUpDisk() { …
  public void eject() { …
}
```

It turns out that my `MusicCentre` object only uses the starting and stopping methods on the `CdPlayer`, the rest are used by some other part of the system. I'm over-specifying my `MediaCentre` by requiring it to talk to a `CdPlayer`, what it actually needs is a *`ScheduledDevice`*. Robert Martin made the point (back in 1996) in his Interface Segregation Principle that "Clients should not be forced to depend upon interfaces that they do not use", but that's exactly what we do when we mock a concrete class.

There's a more subtle but powerful reason for not mocking concrete classes. As part of the TDD with Mocks process, I have to think up names for the relationships I discover—in this example the `ScheduledDevice`. I find that this makes me think harder about the domain and teases out concepts that I might otherwise miss. Once something has a name, I can talk about it.

In case of emergency

There are a few occasions when I have to put up with this smell. The least unacceptable situation is where I'm working with legacy code that I control but can't change all at once. Alternatively, I might be working with third party code that I can't change. As I wrote before, it's usually better to write a veneer over an external library rather than mock it directly, but sometimes it's just too hard. Either way, these are unfortunate but necessary compromises that I would try to work my way out of as soon as possible. The longer I leave them in the code, the more likely it is that some brittleness in the design will cause me grief.

Above all, do not mock by overriding a class's internal features, which just locks down your test to the quirks of the current implementation. Override only visible methods. This rule also prohibits exposing internal methods just so you can override them in a test. If you can't get to the structure you need, then the tests are telling you that it's time to break up the class into smaller, composable features.

Posted by Steve Freeman at 15:29

Labels: design, listening to the tests, testability

14 comments:

Labels

- book (16)
- by the way (1)
- design (27)
- domain-specific-language (5)
- eclipse (1)
- explanation (3)
- growing object-oriented software (16)
- history (1)
- ide (2)
- jmock (7)
- junit4 (3)
- listening to the tests (12)
- mocks in action (5)
- news (20)
- news projects (1)
- pictures (1)
- process (1)
- publications (1)
- specification (1)
- talk (2)
- testability (14)
- tip (3)
- training (2)

Archive

- ► 2010 (1)
- ► 2009 (6)

Luismi said...

I'd like to know your recomendation when dealing with entities database persisted (using an ORM, for example) in a rich domain model. When testing an entity, I supposed you mock up every related entity, so applying the same rationale, do you recommend that every entity has its own interface?. And what if (very likely) one entity doesn't need the whole interface of every related entity?

I hope you get my point, my english isn't very good. What I'd like to know is how to apply this approach when there's several instances of each class/interface which state matters and that likely shall be persisted/depersisted.

25 April 2007 at 07:31

Steve Freeman said...

It all depends :)

So, what am I testing here? If I'm testing persistence, then I'd create a graph of objects, save into a DB, reload, and compare. If I'm testing behaviour then I want just the objects that are relevant to the test. If these are value objects, then there's probably nothing to mock, just use real ones. If there's behaviour then I might have collaborators which would need an interface.

Hmmm. We need more examples.

25 April 2007 at 10:20

Luismi Cavallé said...

An example: We have two entites, Customer and Order. We are testing the method "process" of the class Order. This method is expected to call the method "hasCredit" of its Customer and do different things depending on the value returned.

In this situation I would mock the Customer, but I don't see any value in using an interface that will be exact to the object itself, instead of mocking the object directly. I tend to think that interfaces are appropiate for services but not for entities. [when I talk about services or entities, i'm refering to the domain-driven design concepts]

Thank you for your answers.

25 April 2007 at 22:04

Steve Freeman said...

If Customer is just a blob of values and isInCredit() is just a getter, then there might not be much point in mocking.

What happens when the order is processed? Does the state of the Customer change?
26 April 2007 at 12:23

Luismi Cavallé said...

Ok! Oversimplified examples don't serve well to explain my point.

I'll try to find better, real-world, examples.

Thank you again, Steve

26 April 2007 at 17:03

Steve Freeman said...

Thanks for contributing. Always glad to discuss.

26 April 2007 at 17:27

Colin Jack said...

Interesting post.

I'm also interested in how to use interaction testing as a design approach specifically for a (DDD) domain model.

In addition to Lusimi's example the one from the BDD site:

http://behaviour-driven.org/BehaviourDrivenProgrammingExample

This shows (in DDD terms) a Service being tested and its dependencies being mocked.

Thats fine (though it only scratches the surface), and I'd probably test the service that way (probably with additional state tests).

However if we're using the tests as a design technique would we next define interface(s) for the roles the Account is playing here?

6 January 2008 at 19:00

Steve Freeman said...

From what I've seen of the example, the Account *is* a role. We can't see the implementation from this test. This might not be a great name for a role since it's a little bit general, but then this is only a small case. For example, for safety, we might have separate roles for deposit-taking and withdrawing, both implemented eventually by an Account. The Transfer object would only be allowed to see the appropriate role for each collaborating object.

6 January 2008 at 21:39

eric said...

Hi Steve. I'm currently working thru your paper about using mocks for interface discovery.

Perhaps you've addressed this somewhere else, but one reason I find myself wanting to mock concrete classes is to avoid having Responsibility and ResponsibilityImpl littered throughout the production code. Fowler calls this the unwanted Interface Implementation Pair.

These kinds of interfaces hide the cases where there are truly interfaces with real differing implementations, as opposed to simply a "Default/Impl" and a "Mock".

This tension is expressing itself throughout our codebase, and we have half of our people mocking concrete classes, and the other half with singly implemented interfaces.

Thoughts?

1 May 2010 at 22:05

Steve Freeman said...

@eric, the interface name should describe the *role* that the type plays (in the terms of the caller), the class name should tell us something about its implementation, rather than just having "Impl" tacked on the end. For example, Auditor might be implemented by LoggingAuditor or MessagingAuditor.

It might also be that sometimes you can use more real objects in your tests, if they're simple and just there to add structure. And we don't usually mock value objects.

An example would help, you could post to the mailing list associated with our book.

2 May 2010 at 01:15

Aaron said...

You make the case for interfaces by saying the benefit that interfaces provide in isolating a specific relationship between classes is that it allows that relationship to be easily identifiable if another class needs that same functionality identified by the interface (and presumably also that that functionality can be re-implemented by a different class later on).

You also state
>>> "Robert Martin made the point (back in 1996) in his Interface Segregation Principle that "Clients should not be forced to depend upon interfaces that they do not use", but that's exactly what we do when we mock a concrete class."

I think these are valid points but in many, dare i say most, cases if a class is adhering well to the Single Responsibility principle then the interface of the class will almost always be simple enough that the interface will always reflect the classes public methods one-to-one. Which unless that changes (and because of SRP mostly it does not) then in those cases both the above points seem rather mute to me.

I think the point the gentleman is making about people being frustrated with he sees calls the "unwanted Interface Implementation Pair" is because they are effectively just one-to-one mirrors of the class's methods.

Thoughts??
3 June 2010 at 04:42

Steve Freeman said...

My experience is that, for most of the sort of collaborators that I mock, their single responsibility is to bind together two concepts--for example a LoggingAuditor that provides a Log implementation of an Auditor. If I use the class, then I pull into the caller all sorts of Log-related dependencies that aren't relevant. Each time I do this, I increase the coupling across the codebase and make it a little harder to work with. It doesn't matter much for any one class, but the cumulative effect is significant.

It doesn't make a lot of sense in the abstract until you've experienced it.

3 June 2010 at 22:05

Carl-Erik said...

There is some confusion on what you are really saying here that has been brought up at StackExchange:
http://programmers.stackexchange.com/questions/176391/mocking-concrete-class-not-recommended

Care to comment there?

19 November 2012 at 13:55

Steve Freeman said...

@carl-erik. Thanks for the warning. Responded on the other site (as have other people)

19 November 2012 at 16:04

Post a Comment

Newer Post Home Older Post