The Curious Case of Tautological TDD

In 2014 I read an article with the provocative title "Mockists are Dead. Long live Classicists.". In it, the author frames mock objects as "dangerous" and warns against a mistake he calls "tautological TDD", which he describes as writing tests that repeat an idea in different words. This sounds risky to me, and something I'd probably want to warn against. I wondered how mock objects could create this problem, then I examined his example.[1]

The example centers around a `CarRepository`, which appears to implement the Domain-Driven Design Repository pattern in a straightforward way. The production code looks like this:

```
public class CarRepository {
  private ServiceHeaderFactory serviceHeaderFactory;
  private CarService carService;

  public CarRepository(ServiceHeaderFactory serviceHeaderFactory,
                       CarService carService) {
    this.serviceHeaderFactory = serviceHeaderFactory;
    this.carService = carService;
  }
  public Cars findAll() {
    ServiceHeader serviceHeader = serviceHeaderFactory.create();
    return carService.findAll(serviceHeader);
  }
}
```

The test looks like this:

```
@Test
public void shouldRetrieveCarsFromCarServiceUsingTheRightServiceHeader() throws E\
xception {
  // GIVEN
  ServiceHeader serviceHeader = new ServiceHeader();
  ServiceHeaderFactory serviceHeaderFactoryMock = mock(ServiceHeaderFactory.class\
);
  when(serviceHeaderFactoryMock.create()).thenReturn(serviceHeader);
  CarService carServiceMock = mock(CarService.class);
```

[1] Fabio Pereira, "TTDD - Tautological Test Driven Development (Anti Pattern)"

```
10    CarRepository carRepository = new CarRepository(serviceHeaderFactoryMock, carSe\
11  rviceMock);
12
13    // WHEN
14    carRepository.findAll();
15
16    // THEN
17    verify(carServiceMock).findAll(serviceHeader);
18  }
```

The author calls this test "tautological" because the checks appear only to duplicate the implementation. I agree: the **when** and **then** lines…

- `when(serviceHeaderFactoryMock.create()).thenReturn(serviceHeader)`
- `verify(carServiceMock).findAll(serviceHeader)`

…correspond perfectly to a couple of lines in the production code.

- `ServiceHeader serviceHeader = serviceHeaderFactory.create()`
- `return carService.findAll(serviceHeader)`

From this, the author reasons that the test provides no useful information, categorises it as redundant, labels it "tautological", and considers it dangerous. He writes:

> When we write tests this way, most of the time if the implementation changes, we end up changing the expectations of the test as well and yeah, the tests pass automagically. But without knowing much about its **behaviour**. These tests are a mirror of the implementation, therefore tautological.—Fabio Pereira

Another Interpretation

I understand the author's perspective, and I remember feeling skeptical about this myself. It felt strange seeing this correspondence, and it seemed like the same thing every time: the expectation at the end of my test corresponded to the last line of my production method, and the stubs corresponded to almost every other line of the method. I thought that this must fly in the face of *Don't Repeat Yourself*, *Once and Only Once*, and *The Four Elements of Simple Design*, and that made me nervous.

Let me offer another way to interpret the situation. Some objects interact very, very simply with their collaborators. Some objects act as simple mediators: …. It seems pretty reasonable to me that the checks for a mediator of only two collaborators with a single code path would look too simple. I don't think of this as a consequence of using mock objects, but rather a consequence of a very simple design. **I consider this a good thing**. Next month, when we discover that this mediator needs to do more, I suspect that I'll find it easy to understand and easy to add new behavior.

What Do We Really Have Here?

The author's example leaves me a bit confused. I don't quite understand the division of responsibility among the `CarRepository`, the `ServiceHeaderFactory` and the `CarService`. Since I can't use the author a definitive source of information, let me draw some conclusions from the code itself. It seems that `CarService` bears the responsibility for finding cars based on something called the *service header*. It seems that `CarRepository` wants to "find all the cars", but doesn't want to specify a service header, pushing that responsibility up to the caller. And at this point, I have no idea why `CarRepository` even exists.

Meet Bob.

Bob.

Bob has a *service header* (whatever that is) and wants to find all the corresponding cars, but he can't figure out how to make that happen.

Looking at the code available, Bob has two options:

```
new CarService(...).findAll(bobsServiceHeader)
```

```
new CarRepository(new ServiceHeaderFactory() {
  public ServiceHeader create() {
    return bobsServiceHeader;
  }
}).findAll();
```

I see a bewildered and bemused look on Bob's face. I think I understand why. I have no idea what value `CarRepository` brings to the design. So far, it turns a method parameter into an injectable constructor parameter. So far, it serves only to obscure the notion that one looks for cars relative (somehow) to a service header.

Put yourself in Bob's shoes. You have a service header. You want the corresponding cars. (I don't even need to know what "correspond" means yet.) You look for a library function to do this for you. What do you expect?

I know that I expect a function that maps a service header onto a set of cars. Something like

```
Collection<Car> findAllCarsBy(serviceHeader)
```

You know what I **don't** expect?! Something like

> Please put your service header in an envelope and write "Service Header Factory" on the outside of it (ignore the words on the envelope—Jake knows what they mean), then give that envelope to Jake over there. Jake's a good boy. Ask Jake for all the cars, at which point Jake will open the envelope, read the service header, and tell you all the cars that correspond. OK? Good boy, that Jake.

We'd better have a damn good reason for making Bob jump through all these hoops to find the cars that correspond to his service header.

If Bob could only use `CarService` directly, then he wouldn't have to deal with Jake at all. We could make this whole problem disappear by letting Bob use `CarService` directly. In this case, Bob would have the chance to write typical, straightforward collaboration tests:

- stub `CarService` to return convenient collections of `Cars` for his purposes, following the pattern of `0`, `1`, many, lots, oops.
- set an expectation on `CarService` to verify that his code asks for cars corresponding to the correct service header.

He might not need both of these kinds of tests, but he has the option, and I would certainly not label any of those tests "tautological".

Of course, if I simply defined the problem away, you'd want your money back, so I need to transport myself into a universe in which we have damn good reasons for making Bob jump through all these hoops to find the cars that correspond to his service header.

Some Damn Good Reasons…

Why might we need to expose the "find all cars for a service header" feature through `CarRepository` rather than `CarService`?

1. We need to adapt this feature to a frozen interface (like a framework) that disallows specifying the *service header* as a function parameter.
2. We want to shield clients from the difficulty of instantiating `CarService`.[2]
3. We want, for the purposes of this particular application, to interpret "all cars" to mean "all cars corresponding to a specific service header".

Since these all seem plausible, let me address them in turn.

`CarRepository` As Adapter

Thinking back to Design Patterns, we use the term *Adapter* to refer to an object with the responsibility of exporting another object (the delegate) using a more convenient interface. So maybe Fabio needs `CarRepository` to act as an Adapter for `CarService`. OK. Let's assume that. What now?

I have `CarService.findAll(serviceHeader)`, but my clients demand the interface `CarRepository.findAll()`. I have two choices: choose a default value for `serviceHeader` or make it a constructor parameter. I'll explore the first of these options below, because it equates to interpreting "all cars" to mean "all cars corresponding to a specific service header". Here, I'll assume that clients need to retain the option to pass the `serviceHeader` into the `CarRepository` for `findAll()` to use.

Let me repeat that.

Clients need to retain the option to pass the `serviceHeader` into the `CarRepository` for `findAll()` to use.

Guess what, folks: that describes exactly what this class needs to do. It describes **the essential interactions** between `CarRepository` and its collaborator `CarService`. `CarRepository` has the responsibility of giving the correct `serviceHeader` (and we haven't decided how it will get that yet) to `CarService.findAll()`. Now, when we articulate it like that, it sounds like we're assuming the implementation, which takes us back to the "tautological" objection that brought us all here right now.

I see it differently.

I have a `CarService` with a clear interface for mapping a service header to a collection of cars. I have an annoying client who insists on asking me for "all the cars" without specifying that service header. Unfortunately, I can't do that, so as a compromise, I tell the client that as long as it gives me the service header another way, then I'll have the information I need to find "all the cars". (Yeah…

[2] Fabio's article does not divulge how to instantiate `CarService`. It might involve deep magic.

it sounds stupid to me, too, so I hope this isn't how the real system in Fabio's example works. If it is, then I empathise with his situation.) Either way, **my job entails connecting the service header to the `CarService`, because my insane client insists on doing this the hard way**. We've all been there.

If my client would just give me the damn service header along with the request to "find all the cars"—you know, a function parameter—then we wouldn't need any of this craziness. Unfortunately, my client insists on doing things *its* way. Fine. It can have a `CarRepository`, but it has to provide the service header through the repository's constructor, and the repository will make sure that it uses that service header (and not some other one) to "find all the cars". Will that make you happy, client?!!?

Based on this, the test becomes clear to me. In words: the repository had better pass to `CarService.findAll()` whichever `serviceHeader` it received from its client, and not a different one; it had better return the resulting `Cars` unfiltered, back to the client.

I see two tests, although if you wanted to write them as a single test, I would cringe, but not veto the choice.

1. Stub `CarService.findAll(anything())` to return `arbitrarySetOfCars`. Now `CarRepository.findAll()` should be `arbitrarySetOfCars`.
2. When I call `new CarRepository(carService, myServiceHeader).findAll()`, someone should call `carService.findAll(myServiceHeader)`, matching the parameter exactly.

To avoid a deep philosophical argument (at least for now), I've combined these into a single test:

```
1  @Test
2  public void searchesForCarsWithTheRightServiceHeader() throws Exception {
3    final Cars cars = new Cars(new Car("irrelevant car name"));
4    final ServiceHeader serviceHeader = new ServiceHeader();
5
6    final CarService carService = mock(CarService.class);
7
8    // Implicitly verifies that findAll() receives the correct argument
9    when(carService.findAll(serviceHeader)).thenReturn(cars);
10
11   final CarRepository carRepository = new CarRepository(carService, serviceHeader\
12 );
13
14   assertSame(cars, carRepository.findAll());
15 }
```

> Just to simplify the discussion, I've assumed the best-case scenario of an immutable `Cars` object. I would hate to jump through the additional hoops necessary if someone had made `Cars` mutable. (Don't do that. Really.)

Far from "tautological", this test describes **the very purpose** of `CarRepository`: it relies on `CarService` to find the right cars, but it has to provide the right `serviceHeader`.

Uh… why not just write integrated tests? Fabio recommended this in his article. I don't feel the need to duplicate all the tests for `CarService` in order to test `CarRepository`. I also don't feel comfortable testing `CarService` only through `CarRepository`, especially given that `CarService` exports a much more convenient interface to its clients. If anything, I wouldn't bother testing `CarRepository` and put all my energy into testing `CarService`. Seriously.

Hiding The Ugliness of Instantiating `CarService`

Now imagine that whoever wrote `CarService` made it catastrophically difficult to instantiate. Nothing changes. In fact, all the more reason *not* to write tests that integrate `CarRepository` to `CarService`: the code to set up `CarService` would make us throw up in our mouths a little. This pushes me even more strongly to insist on using test doubles for `CarService` in order to bypass its painful constructor. I'd still write the same tests that I've described just a few paragraphs ago, and for the same reasons.

However, and much to my surprise, if we have a `CarService` that takes so much effort to instantiate, then we probably also have a `CarService` that takes too much effort to test. This nudges me in the direction of integrated tests for `CarRepository` **only because we probably have no tests for `CarService` itself**. In this case, we'd have no duplication. You could probably talk me into this, even without a bribe. I'd eventually want to do one of two things: either collapse `CarService` into `CarRepository` or redesign `CarService`'s constructor to suck less. In the first case, the integrated tests would no longer integrate things. In the second case, the integrated tests would (mostly) transform into `CarService` tests and I'd want to write the same `CarRepository` tests as I described just a few paragraphs ago, and for the same reasons.

Choosing a Service Header

If we don't need to adapt the sensible, simple, delightful `CarService` interface to connect to an insane, unreasonable, evil client that insists on `findAll()` with no parameters, then I can only think of one reason for `CarRepository` to exist: to decide on a service header. Maybe for this particular application, we always want cars corresponding to a given service header. (Notice, I still have no earthly idea what "service header" means, nor do I need to know.)

Hey! I just did it again. I described **the essential purpose** of `CarRepository`: to decide which service header to use to find all the cars. This tells me that the tests should focus on searching on "the right" service header. I see two choices: let clients pass the service header into `CarRepository` as a parameter, or force `CarRepository` to decide on the service header itself. Guess what? The first option takes us back to what I've already done here. (All roads seem to lead to one place.) The second option changes things just a little.

Let's assume that `CarRepository` encapsulates "the correct service header"—once again, *whatever that means*. How do I know that `CarRepository` has chosen wisely? I could ask it for its service

header, but I see no other value in making `CarRepository` do that, so I consider that my last resort.[3] With the pieces available to me, I can detect which service header `CarRepository` will use by… you guessed it… setting an expectation on the parameter it passes to `CarService.findAll()`. Once again, I've described **the essential interaction** between `CarRepository` and `CarService`: put "the right" service header in "the right place" at "the right time". Once again: same tests, same reasons, **except** perhaps a different emphasis, as reflected in subtle differences in naming.

```
1  @Test
2  public void searchesForCarsWithTheRightServiceHeader() throws Exception {
3    final Cars cars = new Cars(new Car("irrelevant car name"));
4    final ServiceHeader expectedServiceHeader = new ServiceHeader("my guess at the \
5  correct properties for service header");
6
7    final CarService carService = mock(CarService.class);
8
9    // Implicitly verifies that findAll() receives the correct argument
10   when(carService.findAll(argThat(equalTo(expectedServiceHeader)))).thenReturn(ca\
11 rs);
12
13   // The CarRepository encapsulates the service header
14   final CarRepository carRepository = new CarRepository(carService);
15
16   assertSame(cars, carRepository.findAll());
17 }
```

Rather than playing "spot the differences", I'll enumerate them for you.

1. I renamed `serviceHeader` to `expectedServiceHeader` to emphasise checking that `CarRepository` gets the service header "right".
2. I changed `verify()` to check the parameter to `findAll()` with `equals()` instead of ==/same, because the instance of `ServiceHeader` I *want* to compare to remains trapped inside `CarRepository` and I have no way to get a handle to it. (More on that in a moment.)
3. I don't pass `serviceHeader` into the constructor for `CarRepository`, but instead leave a comment to describe *why* the test has to guess the service header. I've used a string to represent the service header, because Fabio's article didn't describe what makes service headers differ from one another. In the real code, we would put real service header attributes.

Something bothers me about this test. I can express my consternation both in terms of intuition and mechanics.[4] My intuition says, "I shouldn't have to jump through all those hoops to know which

[3] See? I don't always add code "just for testing".

[4] Read more about how intuition and mechanics interact in my article, "Becoming an Accomplished Software Designer".

service header to check for." I look at the code and see that the service header I expect comes out of nowhere: I don't see it anywhere in the input, nor anything that corresponds to it in the input, so I can't account for why to expect it in the output. It works like magic.

I do not like magic.

What to do? I would prefer it if I could say something like *pretend that the `CarRepository` chooses service header X; when it does, then I should expect parameter X when invoking `carService.findAll()`*. How do I implement this?

If this doesn't make you chuckle, then pay more attention.

Step 1. I don't want to change the interface of `CarRepository`, but I need a *seam* so that I can override the service header. I can use Subclass to Test to do this.

```
@Test
public void searchesForCarsWithTheRightServiceHeader() throws Exception {
  final Cars cars = new Cars(new Car("irrelevant car name"));
  final ServiceHeader anyServiceHeader = new ServiceHeader();

  final CarService carService = mock(CarService.class);

  // Implicitly verifies that findAll() receives the correct argument
  when(carService.findAll(anyServiceHeader)).thenReturn(cars);

  // The CarRepository encapsulates the service header
  final CarRepository carRepository = new CarRepository(carService) {
    @Override
    protected ServiceHeader theServiceHeader() {
      return anyServiceHeader;
    }
  };

  assertSame(cars, carRepository.findAll());
}
```

You'll notice a few changes.

1. I don't need to specify any special attributes in the service header any more, because I don't need it to match whatever `CarRepository` has chosen for itself, because I've overridden it anyway.
2. I can go back to verifying the service header parameter with ==/same, rather than resorting to `equals()`, because the test controls the instance of `ServiceHeader` I expect to pass to `CarService.findAll()`.

3. I've renamed `expectedServiceHeader` to `anyServiceHeader`, because now I don't care about the value of that instance, just as long as it *is* a service header. Even `null` would do, although that would carry other risks that I'd rather avoid.

I like this more, but not enough, because although my test now clearly describes the path from input to expected result, it now depends on knowing that `CarRepository` gets its service header from `theServiceHeader()`, rather than using the field directly. Oh no! Tautological TDD!

We can't leave it like this. We won't. Calm down, Bob.

Step 2. Let's Replace Inheritance with Delegation[5].

```
@Test
public void searchesForCarsWithTheRightServiceHeader() throws Exception {
  final Cars cars = new Cars(new Car("irrelevant car name"));
  final ServiceHeader anyServiceHeader = new ServiceHeader();

  final CarService carService = mock(CarService.class);

  // Implicitly verifies that findAll() receives the correct argument
  when(carService.findAll(anyServiceHeader)).thenReturn(cars);

  final CarRepository carRepository = new CarRepository(carService, anyServiceHea\
der);

  assertSame(cars, carRepository.findAll());
}
```

This looks awfully familiar.

Step 3. Profit.

Once again, all roads appear to lead to the same place. I don't know what else to say here. The code wants me to design it this way, clearly, so I oblige it.

Uh... `ServiceHeaderFactory`?

Right. You probably noticed that I've ignored `ServiceHeaderFactory` so far, and I hope that that bothered you a little. It bothered me. Fortunately, it doesn't matter, because `ServiceHeaderFactory` does nothing more than create a closure around `serviceHeader`, so *nothing essential changes* in any of the foregoing if you decide to replace `serviceHeader` with `(() => serviceHeader)()`, or, if you'd like some names with your block of code—

[5] *Refactoring*, page 352.

```
new ServiceHeaderFactory() {
  public ServiceHeader create() {
    return serviceHeader;
  }
}.create()
```

It merely replaces a value with calling a function that returns that value. Potato, potato. You won't hurt my feelings.

Tautological? No. Essential.

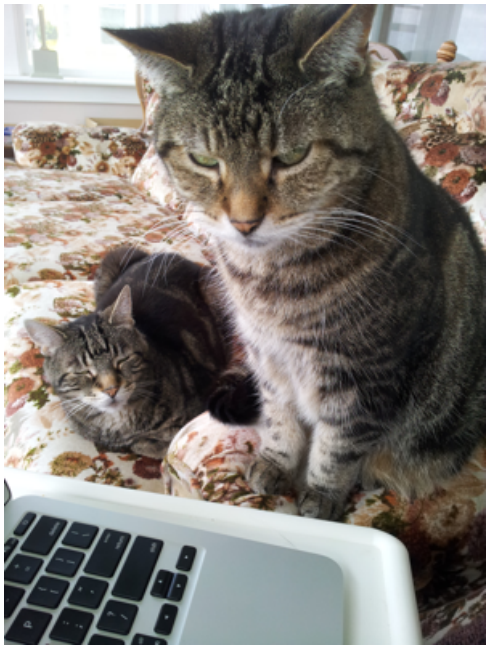
I think I've done a few things here. You will have to judge how well.

1. Shown how easily one can conflate "tautological testing" (a definite risk in programmer testing) with "testing essential interactions" (an effective technique in programmer testing that guides design).
2. Shown how clunky tests with mock objects can point to potential design flaws in production code. (I still think `CarRepository` has to go.)
3. Shown how sensible testing with mock objects and sensible modular design work hand-in-hand.

So there.

Epilogue

I totally lost count of the number of times I typed "Cat" and then replaced it with "Car" when I wrote this.



Cats.