Just Another Blog

... some random thoughts about simplicity and web development.

Sunday, October 30, 2005

Stubs or Mocks - State or Behavior - Testi ed!

Many discussions about my previous blog entry Stubs or Mocks - State or Behavior - Testing or Design, made me write some summary and also share my experience on that controversial but kind of important topic.

The main arguments for using state base rather than using more interactive based unit tests are:

1. **Most of the times the unit tests don't catch all problems.**

Some extracts:

- "...*things often go wrong where one part of a system communicates with another (e.g. business logic connects to a database). The mocked out functions are usually called correctly, but something else is going wrong (e.g. permissions have not been set on the database).*" (from comments)

- *"Why we should use mocks when we could test for free all underlying layers?"* (other developers conversations).

- *"In essence state-based tests are not just unit tests, but also mini-integration tests. As a result many people like the fact that client tests may catch errors that the main tests for an object may have missed, particularly probing areas where classes interact. Interaction tests lose that quality. In addition you also run the risk that*

Blogged by

Igor Stoyanov
Home
ThoughtWorks

expectations on interaction-based tests can be contradictory, resulting in unit tests that run green but mask inherent errors."(Fowler).

2) **Coupling test to implementation.**
3) **Coupling to the implementation also interferes with refactoring, since implementation changes are much more likely to break tests than with state-based testing.**

The first argument is very debatable since there are so many different opinions on the definition of what is unit test. I completely agree with the definition of "unit test" from Wikipedia:

"Limitations
Unit-testing will not catch every error in the program. By definition, it only tests the functionality of the units themselves. Therefore, it will not catch integration errors, performance problems and any other system-wide issues. In addition, it may not be easy to anticipate all special cases of input the program unit under study may receive in reality. Unit testing is only effective if it is used in conjunction with other software testing activities.
It is unrealistic to test all possible input combinations for any non-trivial piece of software. A unit test can only show the presence of errors; it cannot show the absence of errors.
Separation of Interface from Implementation
Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is classes that depend on a database: in order to test the class, the tester finds herself writing code that interacts with the database. This is a mistake, because a unit test should never go outside of its own class boundary. As a result, the software developer abstracts an interface around the database connection, and then implements that interface with their own mock object. This results in loosely coupled code, minimizing dependencies in the system." (Wikipedia)

For the rest of the arguments I will try to show the evolution of the style of unit testing in my current project and all problems related to that.

(All resemblance to real people or code is absolutely on purpose!)

Phase 1: Mostly state based testing – not using mocks or stubs.

We didn't think to much about our tests in the beginning. The tests were mainly state based without any consideration for behavior driven design, mocking or stubbing some parts of the system or for the time of our build.

Consequences:

1. **Since we didn't mock or stub the database access, our test became a little slow for the current code base.**

The rational of not using stubs or mocks was that by using Hibernate, we can use level 1/level 2 Hibernate caches, which would mean that we would not go to the db very often. Even if we query the db directly, we shouldn't worry about speed with today's state of computers and databases.

So, a little slowness in our tests did make us think about some kind of mocking or subbing but since we didn't have any other problems, we didn't change anything.

1. **Our tests were strongly dependent on the test data in the database.**

Since, we were doing mostly state base testing without using any mocks or stubs, our tests strongly relied on test data in the db. The same was true for the unit test in layers not immediately related to the database. Tests from many different layers started failing when something is wrong with the db or there were slight modifications in the test data. I guess that test in isolation principle didn't stand any more, and we had hard time discovering where was the underling problem?

An example would be when we retrieve some

reference data and verify the size of the retrieved collection. Once we check this size in DAO layer, then we check the same in the service layer since the service layer call dao object. Finally, we check for the same size in the view controller class where we construct the drop down component for the UI. You can imagine how many test will fail if we add one more reference data entry in the underling database. This was just because we tried to do state based testing instead of more interactive one.

Actions taken:

Since we started having so many problems after simple changes to the test data in the db and tests became a little slow, we decided to use stubs or/and mocks but only for the database access layer. So, we had persistence tests that tested the database access and our DAO. We used mocks to verify whether a service has called a dao. Also, we used stubs when we didn't care if the object has been persisted or not. When I say stubs, this is a little conditional. Sometimes, we used static stubs, but most of the time we used mocks as stubs - without verifying the behavior after using it.

Phase 2: Mostly state based testing – with using mocks or stubs only for database access in the layers above it.

Consequences:

1. OO design became a "little" too corrupted.

1.1 The state of the objects was exposed just to be tested without any other need in the application.

We had the following code in our application:

```
public class Name {

//public for testability
public Name(){
}

public Name(String firstName, String lastName){
}
```

.....

//public for testability
public void setLastName(String lastName){
.....
}

//public for testability
public Long getId(){ }
}

Of course, this was the immediate consequence of the **state base testing.** When we created the test first, we didn't think what BEHAVIOR we need, rather we thought of how to verify that the state of the object was set correctly. *(We should think very carefully whether we are doing OOP, if we need to create/expose some getter/setter or any kind of internal state for that matter just to test if something went well)* It turns out, that this is common pattern on many other projects, so not many people were concerned about it.
(*And yes, Hibernate can work not only with public but also with private/protected constructors, getters and setters*)

1.2. Instead of building rich Domain Model **with objects that collaborate among each other, we end up having** Anemic Domain Model .

Since testing was very coupled with the state implementation of the tested classes, we started *"treating objects like new age data structures: Ask an object for information about its state, process that, make decisions based on that, then do something to/with the object. This is a classic approach from the old days or procedural programming. It is NOT OO."* (Astel)

"By pulling all the behavior out into services, controllers and so on, essentially end up with Transaction Scripts*, and thus lose the advantages that the domain model can bring."* (Fowler)

Some of the developers (very few) start asking questions about the consequences of not having true Domain model and what we could've changed to fix it. We came to the conclusion that more behavior driven

design through interaction testing could've prevent that:

"One of the hardest things for people to understand in OO design is the "Tell Don't Ask" principle*, which encourages you to tell an object to do something rather than rip data out of an object to do it in client code. Interaction testers say that using interaction testing helps promote this and avoid the getter confetti that pervades too much of code these days."* (Fowler)

However, this problem didn't appeal to the rest of the developers as an important problem.

2. Based on the complicated business logic, too many bugs were introduced because of not properly collaborating and encapsulated object in our model.

As the business logic growth in our application, we had too many defects and the implementation of the story cards was apparently harder and timelier to implement than before.
This was as a direct consequence from the Transaction Script implementation of our business model. Yet, no any actions were taken.

3. Problems with chaining of methods.

"Interaction-based testers do talk more about avoiding 'train wrecks' - method chains of style of `getThis().getThat().getTheOther()`*. Avoiding method chains is also known as following the Law of Demeter. While method chains are a smell, the opposite problem of middle men objects bloated with forwarding methods is also a smell. (I've always felt I'd be more comfortable with the Law of Demeter if it were called the Suggestion of Demeter.)"*(Fowler)

I guess we had much more problems with chaining of methods than forwarding them. For our web application, we used web framework (Tapestry), which binds the object to the view components in configuration file. As a consequence of the state base testing, we had a lot of "train wrecks". Unfortunately, our IDEs (Eclipse, IntelliJ) couldn't successfully refactor those "train wrecks" in those configuration files. One can imagine the consequences of that in a web base

application.

1. **Test in isolation.**

"If you introduce a bug to a system with interaction testing, it will usually cause only tests whose primary object contains the bug to fail. With a state-based approach, however, any tests of client objects can also fail, which leads to failures where the buggy object is used as a secondary object. As a result a failure in a highly used object causes a ripple of failing tests all across the system." (Fowler)

Not only that failing tests rippled through the whole application, but we had hangover cards because of problems with refactoring or JUST FIXING NOT PROPERLY WORKING METHODS. Since, we used state based testing style, we used to verify the consequence of some behavior rather verify the actual behavior.

Example of this would be when we have an object that keeps some state. Let's say that keeping the state in the proper form is major part of an application. Many operations lead to changes in the current state of this particular object. When we use state base testing, we don't very that we change/revert/switch/override the state but rather we verify what is the state after performing some of these operation.

As it usually happens, some of those methods weren't implemented as the business wanted. After fixing them to work properly, the ripple effect of failing test all over the application was kind of awaking call.

Actions taken:
All these problems made our team think more in terms of Domain Driven Design. We started having domain discussions. Also, we concentrate more on the collaboration and behavior of the objects in our Domain Model. We went back to the essential principles of OOP and start thinking more about behavior not so much about state.

Phase 3: Shifting from state based testing to more interactive style of testing.

Consequences:

1. **Complex Fixture Setups**

"*With state-based testing, you have to create all the objects that are going to be involved in responding to the stimulus. While the example only had a couple of objects, real tests often involve a large amount of secondary objects. Usually these objects are created and torn down with each run of the tests. Interaction-based tests, however, only need to create the primary object and mocks for its immediate neighbors. This can avoid some of the involved work in building up complex fixtures.*" (Fowler)

As a consequence of building more sophisticated domain model and more complicated business logic, we started having a lot of complicated fixture setups. Of course we had "Object Moder", although we called it "**Domain Fountain**" as more appropriate name in our test suites. However, we still had very complex setups since most of them were very concrete to a particular test.

Action taken:

As a result of that, we started substituting state based with interactive based testing. We set up and verified mocks all over the place. We used mocks almost everywhere when the set up was more complicated. Also, in this phase we liked the interactive based testing so much so that some developers refactored all previous tests that used dynamic mocks as stubs from not calling verify to call verify of the mock expectation. We clearly moved to the next phase.

Phase 4: Accent on the interactive style of testing. Some initial steps towards Behavior Driven Design.

Consequences:

1. Coupling tests to implementation.

"This coupling leads to a couple of concerns. The most important one is the effect on Test Driven Development. With interaction-based testing, writing the test makes you think about the implementation of the behavior - indeed interaction testers see this as an advantage. State-based testers however think that it's important to only think about what happens from the external

interface and to leave all consideration of implementation until after you're done writing the test."(Fowler).

As Fowler pointed out, I think that this is an advantage. Since unit test is the clearest form of white/"window -(Rod Johnson)" box testing, I see interaction-based testing as the right thing to do. Also, in "pure" unit tests you should not be dependent on the implementation of the secondary objects as this is the case in state base testing.

2. Using Mocks instead of Stubs.

This was one of the most painful mistakes that we did. We used mocks when we want to verify some expectation. However, we used mocks even when we didn't care whether a particular method of a secondary object had been call. What is more, we used mocks instead of stubs in more than 80% of the cases.

"Coupling to the implementation also interferes with refactoring, since implementation changes are much more likely to break tests than with state-based testing." (Fowler).

As a direct consequence of that after every refactoring, we had very painful test fixing sessions.

Phase 5: Behavior Driven Design with very careful consideration on Interactive based testing.

1. Use Mocks only when we verify whether the behavior of the mock object has been called. All other case, use stubs.

In this phase, we use mocks only when we really need to verify the expectation. In all other cases, we use static stubs. In scenario like that, we don't have problem of failing test when the implementation is fine, but some of mock expectations are not met. Using interactive based testing with more consideration will bring all the benefits of that kind of testing but will safe us a lot of troubles with refactoring.

2. Behavior Driven Design.

Since, we saw a huge damage on our OO design with using state base testing, we try to avoid this type of testing. Try having "one assert statement per test" (Astel).

Anyway, this is much better explained in Dave Astel's article about BDD.

Actions Taken:
I think that I would've been satisfied on this level. Unfortunately, our project ran out of time and we never really got to this last phase. … well, probably in the next project we can skip the first couple of phases.

References Used:

1. Fowler, Martin – Mocks aren't Subs - http://www.martinfowler.com/articles/mocksArentStubs.html#CouplingTestsToImplementations

2. Astels, Dave – Behavior Driven Development - http://daveastels.com/files/sdbp2005/BDD_Intro.pdf
3. Wikipedia – Unit Tests - http://en.wikipedia.org/wiki/Unit_testing

|| Igor, Sunday, October 30, 2005

3 Comments:

Igor, this stuff rings very true to me. I've gone through a similar sequence of learning, and so far I'm at 'phase 4', using mocks even when they are probably over-coupling, resulting in brittle code. I wish you had more information on 'phase 5'. I'd really like to hear more of your experiences in that area.

Rob Harwood, at November 03, 2005 12:47 PM

Thanks, for the interest Rob.

I will definitely share my experience for phase 5. Actually, our project didn't finish before phase 5. We had about a month in this phase. We were pretty happy in this phase and didn't have any substantial problems. We definitely eliminated the problem after refactoring with unnecessary failing tests that use mocks. Also, using BDD was very beneficial for better OOD. However, the time wasn't enough to be able to make some generalizations about this phase. I couple of things that could be noted are:
- If TDD seems unnatural in the beginning to most of the developers, BDD is even harder to be apprehend.

> Since BDD make you much more aware about your design and what you expose, some developer doesn't feel that you have to go to all these troubles in order to test behavior of an object rather than expose its state. We surely have some reluctance from some of the developers on the team for practicing BDD. However, if most of the developers on a team are convinced in the benefits of using BDD, the result seams to be very impressive. Of course, as everything else, you should do it in the right way in order to have good results.
> - Using stubs could go in the other direction again. So, when we actually need to use mock in order to verify some kind of behavior, we could instead use static stubs. I haven't seen such a tendency yet but could be a possible outcome.
>
> The time wasn't enough to be able to do good evaluation on phase 5. However, our new project will be to integrate the application with the rest of the enterprise system. During that time, I will be able to evaluated more fully phase 5 and will share the results.
>
> Thanks.

Igor, at November 04, 2005 9:39 AM

> Hi, I am writing a framework based on BDD concepts. I try to think about 'intents' rather than tests: Intent framework

Chiaroscuro, at December 27, 2005 4:19 PM

Add a comment

Links to this post:

Unit-Test and Mocking

Create a Link