CS 528 : Object Oriented Programming and Design **Summer 2014**

Problem Set 7

Dan McQuillan *Handed In: July 5, 2014*

Part 0 Click the links above and read the original blog post, and the response from Kent Beck. Then, answer the following briefly (2-3 sentences).

(a) What are the major complaints that David brings against TDD?

That TDD was hurting the architecture/design of his applications. TDD will lead to overly complex applications and unnecessary abstraction. He also states that we should be focusing more on the design of an application rather than the structure of our unit tests.

(b) What does he propose as being the solution?

His proposed solution is that we should put less emphasis on unit tests and more emphasis on system tests. He proposes that we should move away from test first and move more in the direction of testing the responses after the fact.

(c) Pick two of the things Kent Beck would miss without using TDD. Can you accomplish those things without the use of TDD?

i. Logic errors
This could be accomplished through rigorous testing of the service, but it would be easier to plan out these utilities via unit testing and the test first pattern.

ii. Anxiety
This can be accomplished also through rigorous e2e tests, which can be very time consuming.

Part 1 (a) What is a Mock Object? Give an example of the use for a mock.

A mock object can be used to simulate an example instantiation of an object/class. An example usage could be when interacting with a ReST service. For example if you have a user authentication service and you would like to test responses from the service to make sure that the correct flow is taken for a response from the service.

(b) At one point, Kent says the following:

Even if I don't know how to implement something I can almost always figure out how to write a test for it. And if I can't figure out how to write a test, I have no business writing it in the first place.

Do you agree with this sentiment? Why or why not?

I don't fully agree with this sentiment. However, I do agree with Kent's opinion of you should have no business writing it yet. I say yet because I believe it's more of an issue of you not understanding the problem at hand. Not knowing how to write the unit tests is just an example of a causality of not understanding the problem.

(c) What is the "red/green/refactor" loop?

Red is after writing the tests they will failure. Then green is working on it until it works. Once it's working it may not be in a "perfect" state so you should then refactor to clean up the code to make it more readable/reusable. It is noted as a loop since this process should be repeated until the desired results have been reached.

(d) Which way of arriving at a test suite is better, in your opinion: "going through the tests" (test-first) or "going to the tests" (test-after)? Does it matter at all? Does this depend on the situation? (If so, in what way/ways?)

I think it depends on the situation and the task at hand. If for example you're writing a service that acts as a utility and you fully know what needs to be implemented and all the use cases. Then, test first may be the better option. However, if you're working on a task that is more dependent on a third party, such as an api, where all the use cases are not clear. Then, a mix of the two might be better. Test first to create the necessary set up and helpers and test after to do integration and performance testing.

(e) In what cases is TDD not (easily) applicable? Give at least one example. What can you do in these cases?

An example where TDD is not easily applicable would be when using live data. In these cases integration, service, and usability testing can be helpful. You can also use mock objects in those cases although all the cases may not be caught and tested.

(f) In what ways can an overzealous demand for isolation in unit testing create problems? Is this inherent in unit-testing or TDD?

By having a demand for isolation you are making sure that each component works when isolated from the rest of the components. However you are not testing these components when they are communicating with each other which can cause some edge cases to arise. This is inherent mainly in unit testing as TDD will allows you to thoroughly analyze the problem before beginning on development. However, it can also appear in TDD in edge cases that a developer may not expect.

(g) Why should you have an automated unit test suite? Give at least two reasons.

As the code base grows on larger projects it helps to have to test suite to make sure changes to the code do not break the core functional aspects of the application. This can help by integrating the unit tests in the build of the application. Another reason is because it helps to document the functionality of the component you are unit testing.

Part 2 (a) What is an MVC application? Where did this design pattern originate?

A simple explanation of MVC is that you have a Model, View and Controller. The controller manipulates the model. The model updates the view. When the view changes it uses and interacts with the controller.

From wikipedia:
MVC was one of the seminal insights in the early development of graphical user

interfaces, and one of the first approaches to describe and implement software constructs in terms of their responsibilities.

(b) Can an application have a worse design because of an attempt to make it more testable? Or is it the case that something that is more testable is always a better design?

I don't believe that an application can have a worse design if it is made more testable. I think that if it is made more testable(using a proper testing strategy) and it is broken up into lexical modules then the design will not suffer. However, if it is made such that it is broken up into overly simplified modules it can make the design of an application harder to understand.

(c) How does TDD influence the design of code? Give a positive and a negative example.

It can help a programmer to figure out a logical way to structure and separate your application into modules. A positive example could be when writing a larger application it forces you to think out the design of the application beforehand. However, a negative example is based on the same reason. Since you have to think out the design of the application, you will be breaking up your application into unit testable components. For example, if you are writing a simple Customer class you might have to then write an interface, a mock class, some IoC configuration and then tests for each.

(d) David correlates the size of a codebase with how easy it is to change it. Does this correlation ring true, in your experience?

I think that it depends on the situation. In some cases easy to understand and change code can be written in a few lines when the problem at hand is simple. However, for larger applications a smaller size of the codebase can lessen how orthogonal the application is.

(e) Compare the situation David brings up of "test-induced design damage" coming from a desire for isolation in TDD with the concept of "speculative generality". Are they similar? the same? why/why not?

They are in most cases the same. Speculative generality can be seen when the only callers of a method are test cases. Such methods could also be seen as test induced design damage. Test induced design damage also has the same result since in both cases the problem is over abstracted to allow for testing every aspect of the problem in isolation.

(f) What is cohesion? What is coupling? In what ways can more of one mean less of the other?

Coupling is how much a module relies on another module. Cohesion is how much parts of a module belong together. Being low coupling would mean that changing something major in one class should not affect the other. High coupling would make your code difficult to make changes as well as to maintain it, as classes are coupled closely together, making a change could mean an entire system revamp.

(g) Kent says,

Difficulty testing is a symptom of poor design.

Do you agree?

I agree with his statement since if a module is easy to test that means that it will usually have high cohesion. This is because the higher the cohesion the larger the isolation of the module. If a module is harder to test that could be a result of having higher coupling since it may be harder to test due to a need for mock objects to test it.

(h) Kent says he is optimistic in that he believes that there is always some design insight that exists that will result in a design that is simultaneously more easily testable as well as better structured.

Do you share his optimism?

I share his optimism for the hope that a design insight will eventually let a code base's design not suffer when making a code base more testable.

Part 3 (a) Martin identifies three important forms of feedback for software development. What were they, and how well is each one served by having a unit test suite?

i. Is the software doing something useful for the user
This works well since it will provide concrete examples of how the software is meant to be used by a given user. However, an aspect of this feedback is how to render output and thinking about your user's needs. This is not conducive to tests since it's more so what looks good. It also has a lot more manual tests such as usability testing to make sure the user base will, when given a page, know what their next logical step should be.

ii. Making sure you don't break anything
This one is test conducive since it will check your code base with a given test suite with each compile which will help to given feedback to the user if anything major has been broken.

iii. Is my codebase healthy
This one can be test conducive but it is more reliant on writing readable code and logical modularity of the codebase.

(b) How has Quality Assurance (QA) for programming changed over time? In particular, what role did TDD and self-testing code play in this change?

It has more so evolved to using TDD and gaining an aspect of overconfidence. This has created a thought that QA was not necessary and has led to some companies removing their QA process before the release of the software.

(c) What does David mean when he refers to "criticality"? How should the "criticality" of your software influence your testing?

The criticality refers to the importance of the software being built and what would be lost if it did not fully function. The criticality should influence test need for extensive testing. The higher the criticality the more focus should be placed on testing.

(d) Describe how it's possible for code with 100% test coverage with all tests passing can still have bugs.

When a software is pushed into the "wild" and the users start using it. A user can sometimes do things that no one could ever imagine or even plan for because you had thought that the flow was so ridiculous and no one would ever do that. It can often be hard to plan for those cases.

Part 4 (a) Kent mentions a good measure for understanding how important a given test is for a codebase. What is it? How could you determine its value for a given test? Using this measure, how could you numerically define "overtesting"?

He mentions the concept of delta coverage. If you have the exact same behavior in other places and multiple tests supply coverage for that case then the value for the new test is lessened. Using this measure if the there is no confidence gained from the value of delta coverage then the test should not be added.

(b) Describe Martin's trick used to identify if a given line of code is tested by the current test suite.

Remove the line of code and see if a test break.

(c) How is the ratio of functional code to test code influenced by coupling in the system?

The ratio is directly proportionate to the coupling in the system. If the coupling in the system goes up then so would the ration. Conversely, if the coupling goes down and the code base gets more cohesive then the ration should also go down.

(d) What are the symptoms of overtested code? undertested code?

You don't have enough tests if when you are changing the code you don't feel confident that it won't break because of the code which is considered under testing. Over testing is more focused on whether or not you're spending more time writing tests than you are writing production code.

(e) How does the "audience" of a codebase change its testing tradeoffs?

When handing off to a team of developers it causes the development team to become more paranoid that it is going to have significant issues. However, when handing the software off to a business team might result in having less test coverage since some developers may feel that they will be more confined to specific use cases. Because of this, the audience can have a direct correlation to the testing coverage of the codebase.

(f) Describe the difference between placing unit tests as the "authority" of the system differs from placing the functional code as the "authority" of the system. Which places a larger focus on refactoring?

When the tests are the authority of the system that means that when someone is looking at the codebase they should find the most clarity in what the system should do by looking at the tests. When the functional code is the authority, it means that you should look at the production code to interpolate what it should be doing. The functional code as the authority places a larger focus on refactoring.

(g) What is one situation where you would be more happy to lose the tests and keep the code? What is one situation where you would be more happy to lose the code and keep the tests? What is different about these two situations?

Lose tests:
If the tests are no longer sufficient the currently design of the system and are need of refactoring to create a better test of what the system should do.

Lose code:
If a system is being upgraded than the code is likely to be mostly deprecated so it would be better to remove the code.

What's different:
When losing the tests I believe it is more focused on how the system is designed and how it changes over time. Losing the code is more reliant on the design staying consistent over the life of the codebase.

(h) David thinks the testing battle has been won, and Martin isn't so convinced. Do you believe that we've "won" the battle to get people to unit test their code? Why or why not?

Yes, I believe that we've won the battle to get people to unit test their code. It is seen as good practice to unit test in order to gain confidence that a codebase is healthy. It also helps to explain to new developers what the code is meant to do if the tests are seen as a greater authority than the production code. However, although unit testing seems to have won the battle it does not directly follow that people will use it given a set of circumstances. Any number of excuses for a developer not to use unit testing in a project can be used. For example, one of the common excuses I've heard is that there is not enough time to create unit testable code and also the unit tests for it.