Google Testing Blog

How to get Started with TDD

Tuesday, November 17, 2009

By Miško Hevery

Best way to learn TDD is to have someone show you while pairing with you. Short of that, I have set up an eclipse project for you where you can give it a try:

1. `hg clone https://bitbucket.org/misko/misko-hevery-blog/`
2. Open project `blog/tdd/01_Calculator` in Eclipse.
3. It should be set up to run all tests every time you modify a file.
    - You may have to change the path to java if you are not an Mac OS.
    - Project -> Properties -> Builders -> Test -> Edit
    - Change location to your java
4. Right-click on Calculator.java -> Run As -> Java Application to run the calculator

Your mission is to make the calculator work using TDD. This is the simplest form of TDD where you don't have to mock classes or create complex interactions, so it should be a good start for beginners.

TDD means:

1. write a simple test, and assert something interesting in it
2. implement just enough to make that tests green (nothing more, or you will get ahead of your tests)
3. then write another test, rinse, and repeat.

I have already done all of the work of separating the behavior from the UI, so that the code is testable and properly Dependency Injected, so you don't have to worry about running into testability issues.

- Calculator.java: This is the main method and it is where all of the wiring is happening.
- CalculatorView.java: This is a view and we don't usually bother unit testing it has cyclomatic complexity of one, hence there is no logic. It either works or does not. Views are usually good candidates for end-to-end testing, which is not part of this exercise.
- CalculatorModel.java: is just a PoJo which marshals data from the Controller to the View, not much to test here.
- CalculatorController.java: is where all of your if statements will reside, and we need good tests for it.

I have started you off with first 'testItShouldInitializeToZero' test. Here are some ideas for next tests you may want to write.

- testItShouldConcatinateNumberPresses
- testItShouldSupportDecimalPoint
- testItShouldIgnoreSecondDecimalPoint
- testItShouldAddTwoIntegers

I would love to see what you will come up with and what your thoughts are, after you get the whole calculator working. I would also encourage you to post interesting corner case tests here for others to incorporate. If you want to share your code with others, I would be happy to post your solutions.

Good luck!

PS: I know it is trivial example, but you need to start someplace.

 10 comments 

Labels: Misko Hevery

Cost of Testing

Friday, October 02, 2009

By Miško Hevery

A lot of people have been asking me lately, what is the cost of testing, so I decided, that I will try to measure it, to dispel the myth that testing takes twice as long.

For the last two weeks I have been keeping track of the amount of time I spent writing tests versus the time writing production code. The number surprised even me, but after I thought about it, it makes a lot of sense. The magic number is about 10% of time spent on writing tests. Now before, you think I am nuts, let me back it up with some real numbers from a personal project I have been working on.

	Total	Production	Test	Ratio
Commits	1,347	1,347	1,347	
LOC	14,709	8,711	5,988	40.78%
JavaScript LOC	10,077	6,819	3,258	32.33%
Ruby LOC	4,632	1,892	2,740	59.15%
Lines/Commit	10.92	6.47	4.45	40.78%
Hours(estimate)	1,200	1,080	120	10.00%
Hours/Commit	0.89	0.80	0.09	
Mins/Commit	53	48	5	

Commits refers to the number of commits I have made to the repository. LOC is lines of code which is broken down by language. The ratio shows the typical breakdown between the production and test code when you test drive and it is about half, give or take a language. It is interesting to note that on average I commit about 11 lines out of which 6.5 are production and 4.5 are test. Now, keep in mind this is average, a lot of commits are large where you add a lot of code, but then there are a lot of commits where you are tweaking stuff, so the average is quite low.

The number of hours spent on the project is my best estimate, as I have not kept track of these numbers. Also, the 10% breakdown comes from keeping track of my coding habits for the last two weeks of coding. But, these are my best guesses.

Now when I test drive, I start with writing a test which usually takes me few minutes (about 5 minutes) to write. The test represents my scenario. I then start implementing the code to make the scenario pass, and the implementation usually takes me a lot longer (about 50 minutes). The ratio is highly asymmetrical! Why does it take me so much less time to write the scenario than it does to write the implementation given that they are about the same length? Well look at a typical test and implementation:

Here is a typical test for a feature:

```
ArrayTest.prototype.testFilter = function() {
 var items = ["MIsKO", {name:"john"}, ["mary"], 1234];
 assertEquals(4, items.filter("").length);
```

```
 assertEquals(4, items.filter(undefined).length);

 assertEquals(1, items.filter('iSk').length);
 assertEquals("MIsKO", items.filter('isk')[0]);

 assertEquals(1, items.filter('ohn').length);
 assertEquals(items[1], items.filter('ohn')[0]);

 assertEquals(1, items.filter('ar').length);
 assertEquals(items[2], items.filter('ar')[0]);

 assertEquals(1, items.filter('34').length);
 assertEquals(1234, items.filter('34')[0]);

 assertEquals(0, items.filter("I don't exist").length);
};

ArrayTest.prototype.testShouldNotFilterOnSystemData = function() {
 assertEquals("", "".charAt(0)); // assumption
 var items = [{$name:"misko"}];
 assertEquals(0, items.filter("misko").length);
};

ArrayTest.prototype.testFilterOnSpecificProperty = function() {
 var items = [{ignore:"a", name:"a"}, {ignore:"a", name:"abc"}];
 assertEquals(2, items.filter({}).length);

 assertEquals(2, items.filter({name:'a'}).length);

 assertEquals(1, items.filter({name:'b'}).length);
 assertEquals("abc", items.filter({name:'b'})[0].name);
};

ArrayTest.prototype.testFilterOnFunction = function() {
 var items = [{name:"a"}, {name:"abc", done:true}];
 assertEquals(1, items.filter(function(i){return i.done;}).length);
};

ArrayTest.prototype.testFilterIsAndFunction = function() {
 var items = [{first:"misko", last:"hevery"},
              {first:"mike", last:"smith"}];

 assertEquals(2, items.filter({first:'', last:''}).length);
 assertEquals(1, items.filter({first:'', last:'hevery'}).length);
 assertEquals(0, items.filter({first:'mike', last:'hevery'}).length);
 assertEquals(1, items.filter({first:'misko', last:'hevery'}).length);
 assertEquals(items[0], items.filter({first:'misko', last:'hevery'})[0]);
};

ArrayTest.prototype.testFilterNot = function() {
 var items = ["misko", "mike"];
```

```
 assertEquals(1, items.filter('!isk').length);
 assertEquals(items[1], items.filter('!isk')[0]);
};
```

Now here is code which implements this scenario tests above:

```
Array.prototype.filter = function(expression) {
 var predicates = [];
 predicates.check = function(value) {
   for (var j = 0; j < predicates.length; j++) {
      if(!predicates[j](value)) {
        return false;
      }
    }
    return true;
  };
  var getter = Scope.getter;
  var search = function(obj, text){
    if (text.charAt(0) === '!') {
      return !search(obj, text.substr(1));
    }
    switch (typeof obj) {
    case "bolean":
    case "number":
    case "string":
      return ('' + obj).toLowerCase().indexOf(text) > -1;
   case "object":
     for ( var objKey in obj) {
       if (objKey.charAt(0) !== '$' && search(obj[objKey], text)) {
         return true;
       }
     }
     return false;
   case "array":
     for ( var i = 0; i < obj.length; i++) {
       if (search(obj[i], text)) {
         return true;
       }
     }
     return false;
   default:
     return false;
   }
 };
 switch (typeof expression) {
   case "bolean":
   case "number":
   case "string":
     expression = {$:expression};
   case "object":
```

```
      for (var key in expression) {
        if (key == '$') {
          (function(){
            var text = (''+expression[key]).toLowerCase();
            if (!text) return;
            predicates.push(function(value) {
              return search(value, text);
            });
          })();
        } else {
          (function(){
            var path = key;
            var text = (''+expression[key]).toLowerCase();
            if (!text) return;
            predicates.push(function(value) {
              return search(getter(value, path), text);
            });
          })();
        }
      }
      break;
    case "function":
      predicates.push(expression);
      break;
    default:
      return this;
  }
  var filtered = [];
  for ( var j = 0; j < this.length; j++) {
    var value = this[j];
    if (predicates.check(value)) {
      filtered.push(value);
    }
  }
  return filtered;
};
```

Now, I think that if you look at these two chunks of code, it is easy to see that even though they are about the same length, one is much harder to write. The reason, why tests take so little time to write is that they are linear in nature. No loops, ifs or interdependencies with other tests. Production code is a different story, I have to create complex ifs, loops and have to make sure that the implementation works not just for one test, but all test. This is why it takes you so much longer to write production than test code. In this particular case, I remember rewriting this function three times, before I got it to work as expected. :-)

So a naive answer is that writing test carries a 10% tax. But, we pay taxes in order to get something in return. Here is what I get for 10% which pays me back:

- When I implement a feature I don't have to start up the whole application and click several pages until I get to page to verify that a feature works. In this case it means that I don't have to refreshing the browser, waiting for it to load a dataset and then typing some test data and manually asserting that I got what I expected. This is immediate payback in time saved!
- Regression is almost nil. Whenever you are adding new feature you are running the risk of breaking something other then what you are working on immediately (since you are not working on it you are not actively testing it). At least once a day I have a what the @#$% moment when a change suddenly breaks a test at the opposite end of the codebase which I did not expect, and I count my lucky stars. This is worth a lot of time spent when you discover that a feature you thought was working no longer is, and by this time you have forgotten how the feature is implemented.
- Cognitive load is greatly reduced since I don't have to keep all of the assumptions about the software in my head, this makes it really easy to switch tasks or to come back to a task after a meeting, good night sleep or a weekend.
- I can refactor the code at will, keeping it from becoming stagnant, and hard to understand. This is a huge problem on large projects, where the code works, but it is really ugly and everyone is afraid to touch it. This is worth money tomorrow to keep you going.

These benefits translate to real value today as well as tomorrow. I write tests, because the additional benefits I get more than offset the additional cost of 10%. Even if I don't include the long term benefits, the value I get from test today are well worth it. I am faster in developing code with test. How much, well that depends on the complexity of the code. The more complex the thing you are trying to build is (more ifs/loops/dependencies) the greater the benefit of tests are.

So now you understand my puzzled look when people ask me how much slower/costlier the development with tests is.

6 comments

Labels: Misko Hevery

Checked exceptions I love you, but you have to go

Wednesday, September 16, 2009

By Miško Hevery

Once upon a time Java created an experiment called checked-exceptions, you know, you have to declare exceptions or catch them. Since that time, no other language (I know of) has decided to copy this idea, but somehow the Java developers are in love with checked exceptions. Here, I am going to "try" to convince you that checked-exceptions, even though look like a good idea at first glance, are actually not a good idea at all:

Empirical Evidence

Let's start with an observation of your code base. Look through your code and tell me what percentage of catch blocks do rethrow or print error? My guess is that it is in high 90s. I would go as far as 98% of catch blocks are meaningless, since they just print an error or rethrow the exception which will later be printed as an error. The reason for this is very simple. Most exceptions such as FileNotFoundException, IOException, and so on are sign that we as developers have missed a corner case. The exceptions are used as away of informing us that we, as developers, have messed up. So if we did not have checked exceptions, the exception would be throw and the main method would print it and we would be done with it (optionally we would catch all exceptions in the main log them if we are a server).

Checked exceptions force me to write catch blocks which are meaningless: more code, harder to read, and higher chance that I will mess up the rethrow logic and eat the exception.

Lost in Noise

Now lets look at the 2-5% of the catch blocks which are not rethrow and real interesting logic happens there. Those interesting bits of useful and important information is lost in the noise, since my eye has been trained to skim over the catch blocks. I would much rather have code where a catch would indicate: "pay, attention! here, something interesting is happening!", rather than, "it is just a rethrow." Now, if we did not have checked exceptions, you would write your code without catch blocks, test your code (you do test right?) and realize that under some circumstances an exception is throw and deal with it by writing the catch block. In such a case forgetting to write a catch block is no different than forgetting to write an else block of the if statement. We don't have checked ifs and yet no one misses them, so why do we need to tell developers that FileNotFound can happen. What if the developer knows for a fact that it can not happen since he has just placed the file there, and so such an exception would mean that your filesystem has just disappeared! (and your application is not place to handle that.)

Checked exception make me skim the catch blocks as most are just rethrows, making it likely that you will miss something important.

Unreachable Code

I love to write tests first and implement as a consequence of tests. In such a situation you should always have 100% coverage since you are only writing what the tests are asking for. But you don't! It is less than 100% because checked exceptions force you to write catch blocks which are impossible to execute. Check this code out:

```
bytesToString(byte[] bytes) {
 ByteArrayOutputStream out = new ByteArrayOutputStream();
 try {
   out.write(bytes);
   out.close()
   return out.toSring();
 } catch (IOException e) {
   // This can never happen!
   // Should I rethrow? Eat it? Print Error?
 }
}
```

ByteArrayOutputStream will never throw IOException! You can look through its implementation and see that it is true! So why are you making me catch a phantom exception which can never happen and which I can not write a test for? As a result I cannot claim 100% coverage because of things outside my control.

Checked exceptions create dead code which will never execute.

Closures Don't Like You

Java does not have closures but it has visitor pattern. Let me explain with concrete example. I was creating a custom class loader and need to override load() method on MyClassLoader which throws ClassNotFoundException under some circumstances. I use ASM library which allows me to inspect Java bytecodes. The way ASM works is that it is a visitor pattern, I write visitors and as ASM parses the bytecodes it calls specific methods on my visitor implementation. One of my visitors, as it is examining bytecodes, decides that things are not right and needs to throw a ClassNotFondException which the class loader contract says it should throw. But now we have a problem. What we have on a stack is MyClassLoader -> ASMLibrary -> MyVisitor. MyVisitor wants to throw an exception which MyClassLoader expects but it can not since ClassNotFoundException is checked and ASMLibrary does not declare it (nor should it). So I have to throw RuntimeClassNotFoundException from MyVisitor which can pass through ASMLibrary which MyClassLoader can then catch and rethrow as ClassNotFoundException.

Checked exception get in the way of functional programing.

Lost Fidelity

Suppose java.sql package would be implemented with useful exception such as SqlDuplicateKeyExceptions and SqlForeignKeyViolationException and so on (we can wish) and suppose these exceptions are checked (which they are). We say that the SQL package has high fidelity of exception since each exception is to a very specific problem. Now lets say we have the same set up as before where there is some other layer between us and the SQL package, that layer can either redeclare all of the exceptions, or more likely throw its own. Let's look at an example, Hibernate is object-relational-database-mapper, which means it converts your SQL rows into java objects. So on the stack you have MyApplication -> Hibernate -> SQL. Here Hibernate is trying hard to hide the fact that you are talking to SQL so it throws HibernateExceptions instead of SQLExceptions. And here lies the problem. Your code knows that there is SQL under Hibernate and so it could have handled SqlDuplicateKeyException in some useful way, such as showing an error to the user, but Hibernate was forced to catch the exception and rethrow it as generic HibernateException. We have gone from high fidelitySqlDuplicateKeyException to low fidelity HibernateException. An so MyApplication can not do anything. Now Hibernate could have throw HibernateDuplicateKeyException but that means that Hibernate now has the same exception hierarchy as SQL and we are duplicating effort and repeating ourselves.

Rethrowing checked exceptions causes you to lose fidelity and hence makes it less likely that you could do something useful with the exception later on.

You can't do Anything Anyway

In most cases when exception is throw there is no recovery. We show a generic error to the user and log an exception so that we con file a bug and make sure that that exception will not happen again. Since 90+% of the exception are bugs in our code and all we do is log, why are we forced to rethrow it over and over again.

It is rare that anything useful can be done when checked exception happens, in most case we die with error! Therefor I want that to be the default behavior of my code with no additional typing.

How I deal with the code

Here is my strategy to deal with checked exceptions in java:

- Always catch all checked exceptions at source and rethrow them as LogRuntimeException.
    - LogRuntimeException is my runtime un-checked exception which says I don't care just log it.
    - Here I have lost Exception fidelity.
- All of my methods do not declare any exceptions

- As I discover that I need to deal with a specific exception I go back to the source where LogRuntimeException was thrown and I change it to <Specific>RuntimeException (This is rarer than you think)
    - I am restoring the exception fidelity only where needed.
- Net effect is that when you come across a try-catch clause you better pay attention as interesting things are happening there.
    - Very few try-catch calluses, code is much easier to read.
    - Very close to 100% test coverage as there is no dead code in my catch blocks.

34 comments

Labels: Java , Misko Hevery

It is not about writing tests, its about writing stories

Wednesday, September 02, 2009

by Miško Hevery

I would like to make an analogy between building software and building a car. I know it is imperfect one, as one is about design and the other is about manufacturing, but indulge me, the lessons are very similar.

A piece of software is like a car. Lets say you would like to test a car, which you are in the process of designing, would you test is by driving it around and making modifications to it, or would you prove your design by testing each component separately? I think that testing all of the corner cases by driving the car around is very difficult, yes if the car drives you know that a lot of things must work (engine, transmission, electronics, etc), but if it does not work you have no idea where to look. However, there are some things which you will have very hard time reproducing in this end-to-end test. For example, it will be very hard for you to see if the car will be able to start in the extreme cold of the north pole, or if the engine will not overheat going full throttle up a sand dune in Sahara. I propose we take the engine out and simulate the load on it in a laboratory.

We call driving car around an end-to-end test and testing the engine in isolation a unit-test. With unit tests it is much easier to simulate failures and corner cases in a much more controlled environment. We need both tests, but I feel that most developers can only imagine the end-to-end tests.

But lets see how we could use the tests to design a transmission. But first, little terminology change, lets not call them test, but instead call them stories. They are stories because that is what they tell you about your design. My first story is that:

- the transmission should allow the output shaft to be locked, move in same direction (D) as the input shaft, move in opposite (R) or move independently (N)

Given such a story I could easily create a test which would prove that the above story is true for any design submitted to me. What I would most likely get is a transmission which would only have a single gear in each direction. So lets write another story

- the transmission should allow the ratio between input and output shaft to be [-1, 0, 1, 2, 3, 4]

Again I can write a test for such a transmission but i have not specified how the forward gear should be chosen, so such a transmission would most likely be permanently stuck in 1st gear and limit my speed, it will also over-rev the engine.

- the transmission should start in 1st and than switch to higher gear before the engine reaches maximum revolutions.

This is better, but my transmission would most likely rev the engine to maximum before it would switch, and once it would switch to higher gear and I would slow down, it would not down-shift.

- the transmission should down shift whenever the engine RPM fall bellow 1000 RPMs

OK, now it is starting to drive like a car, but still the limits for shifting really are 1000-6000 RPMs which is not very fuel efficient way to drive your car.

- the transmission should up-shift whenever the estimated fuel consumption at a higher gear ration is better than the current one.

So now our engine will not rev any more but it will be a lazy car since once the transmission is in the fuel efficient mode it will not want to down-shift

- the transmission should down-shift whenever the gas pedal is depressed more than 50% and the RPM is lower than the engine's peak output RPM.

I am not a transmission designer, but I think this is a decent start.

Notice how I focused on the end result of the transmission rather than on testing specific internals of it. The transmission designer would have a lot of levy in choosing how it worked internally, Once we would have something and we would test it in the real world we could augment these list of stories with additional stories as we discovered additional properties which we would like the transmission to posses.

If we would decide to change the internal design of the transmission for whatever reason we would have these stories as guides to make sure that we did not forget about anything. The stories represent assumptions which need to be true at all times. Over the lifetime of the component we can collect hundreds of stories which represent equal number of assumption which is built into the system.

Now imagine that a new designer comes on board and makes a design change which he believes will improve the responsiveness of the transmission, he can do so because the existing stories are not restrictive in how, only it what the outcome should be. The stories save the designer from breaking an existing assumption which was already designed into the transmission.

Now lets contrast this with how we would test the transmission if it would already be build.

- test to make sure all of the gears work
- test to make sure that the engine is not allowed to over-rev

It is hard now to think about what other tests to write, since we are not using the tests to drive the design. Now, lets say that someone now insist that we get 100% coverage, we open the transmission up and we see all kinds of logic, and rules and we don't know why since we were not part of the design so we write a test

- at 3000 RPM input shaft, apply 100% throttle and assert that the transmission goes to 2nd gear.

Tests like that are not very useful when you want to change the design, since you are likely to break the test, without fully understanding why the test was testing that specific conditions, it is hard to know if anything was broken if the tests is red.. That is because the tests does not tell a story any more, it only asserts the current design. It is likely that such a test will be in the way when you will try to do design changes. The point I am trying to make is that there is huge difference between writing tests before or after. When we write tests before we are:

- creating a story which is forcing a particular design decision.

- tests are a collection of assumptions which needs to be true at all times.

when we write tests after the fact we:

- miss a lot of reasons why things are done in particular way even if we have 100% coverage
- test are often brittle because they are tied to particulars of the current implementation
- tests are just snapshots and don't tell a story of why the component does something, only that it does.

For this reason there are huge differences in quality when writing assumptions as stories before (which force design to emerge) or writing tests after which take a snapshot of a given design.

8 comments

Labels: Misko Hevery

How to think about OO

Friday, July 31, 2009

by Miško Hevery

Everyone seems to think that they are writing OO after all they are using OO languages such as Java, Python or Ruby. But if you exam the code it is often procedural in nature.

Static Methods

Static methods are procedural in nature and they have no place in OO world. I can already hear the screams, so let me explain why, but first we need to agree that global variables and state is evil. If you agree with previous statement than for a static method to do something interesting it needs to have some arguments, otherwise it will always return a constant. Call to a staticMethod() must always return the same thing, if there is no global state. (Time and random, has global state, so that does not count and object instantiation may have different instance but the object graph will be wired the same way.)

This means that for a static method to do something interesting it needs to have arguments. But in that case I will argue that the method simply belongs on one of

its arguments. Example: Math.abs(-3) should really be -3.abs(). Now that does not imply that -3 needs to be object, only that the compiler needs to do the magic on my behalf, which BTW, Ruby got right. If you have multiple arguments you should choose the argument with which method interacts the most.

But most justifications for static methods argue that they are "utility methods". Let's say that you want to have toCamelCase() method to convert string "my_workspace" to "myWorkspace". Most developers will solve this as StringUtil.toCamelCase("my_workspace"). But, again, I am going to argue that the method simply belongs to the String class and should be "my_workspace".toCamelCase(). But we can't extend the String class in Java, so we are stuck, but in many other OO languages you can add methods to existing classes.

In the end I am sometimes (handful of times per year) forced to write static methods **due to limitation of the language**. But that is a rare event since static methods are death to testability. What I do find, is that in most projects static methods are rampant.

Instance Methods

So you got rid of all of your static methods but your codes still is procedural. OO says that code and data live together. So when one looks at code one can judge how OO it is without understanding what the code does, simply by looking at the relationship of data and code.

```
class Database {
 // some fields declared here
 boolean isDirty(Cache cache, Object obj) {
   for (Object cachedObj : cache.getObjects) {
     if (cachedObj.equals(obj))
       return false;
   }
   return true;
 }
}
```

The problem here is the method may as well be static! It is in the wrong place, and you can tell this because it does not interact with any of the data in the Database, instead it interacts with the data in cache which it fetches by calling the getObjects() method. My guess is that this method belongs to one of its arguments most likely Cache. If you move it to Cache you well notice that the Cache will no longer need the getObjects() method since the for loop can access the internal state of the Cache directly. Hey, we simplified the code (moved one method, deleted one method) and we have made Demeter happy.

The funny thing about the getter methods is that it usually means that the code where the data is processed is outside of the class which has the data. In other words the code and data are not together.

```
class Authenticator {
 Ldap ldap;
 Cookie login(User user) {
   if (user.isSuperUser()) {
     if ( ldap.auth(user.getUser(),
            user.getPassword()) )
       return new Cookie(user.getActingAsUser());
   } else (user.isAgent) {
       return new Cookie(user.getActingAsUser());
   } else {
     if ( ldap.auth(user.getUser(),
            user.getPassword()) )
       return new Cookie(user.getUser());
   }
   return null;
 }
}
```

Now I don't know if this code is well written or not, but I do know that the login() method has a very high affinity to user. It interacts with the user a lot more than it interacts with its own state. Except it does not interact with user, it uses it as a dumb storage for data. Again, code lives with data is being violated. I believe that the method should be on the object with which it interacts the most, in this case on User. So lets have a look:

```
class User {
 String user;
 String password;
 boolean isAgent;
 boolean isSuperUser;
 String actingAsUser;

 Cookie login(Ldap ldap) {
   if (isSuperUser) {
     if ( ldap.auth(user, password) )
       return new Cookie(actingAsUser);
   } else (user.isAgent) {
       return new Cookie(actingAsUser);
   } else {
     if ( ldap.auth(user, password) )
       return new Cookie(user);
   }
   return null;
```

```
  }
}
```

Ok we are making progress, notice how the need for all of the getters has disappeared, (and in this simplified example the need for the Authenticator class disappears) but there is still something wrong. The ifs branch on internal state of the object. My guess is that this code-base is riddled with if (user.isSuperUser()). The issue is that if you add a new flag you have to remember to change all of the ifs which are dispersed all over the code-base. Whenever I see If or switch on a flag I can almost always know that polymorphism is in order.

```
class User {
 String user;
 String password;

 Cookie login(Ldap ldap) {
   if ( ldap.auth(user, password) )
     return new Cookie(user);
   return null;
 }
}

class SuperUser extends User {
 String actingAsUser;

 Cookie login(Ldap ldap) {
   if ( ldap.auth(user, password) )
     return new Cookie(actingAsUser);
   return null;
 }
}

class AgentUser extends User {
 String actingAsUser;

 Cookie login(Ldap ldap) {
   return new Cookie(actingAsUser);
 }
}
```

Now that we took advantage of polymorphism, each different kind of user knows how to log in and we can easily add new kind of user type to the system. Also notice how the user no longer has all of the flag fields which were controlling the ifs to give the user different behavior. The ifs and flags have disappeared.

Now this begs the question: should the User know about the Ldap? There are actually two questions in there. 1) should User have a field reference to Ldap? and 2) should User have compile time dependency on Ldap?

Should User have a field reference to Ldap? The answer is no, because you may want to serialize the user to database but you don't want to serialize the Ldap. See here.

Should User have compile time dependency on Ldap? This is more complicated, but in general the answer depends on weather or not you are planning on reusing the User on a different project, since compile time dependencies are transitive in strongly typed languages. My experience is that everyone always writes code that one day they will reuse it, but that day never comes, and when it does, usually the code is entangled in other ways anyway, so code reuse after the fact just does not happen. (developing a library is different since code reuse is an explicit goal.) My point is that a lot of people pay the price of "what if" but never get any benefit out of it. Therefore don't worry abut it and make the User depend on Ldap.

19 comments

Labels: Misko Hevery

Software Testing Categorization

Tuesday, July 14, 2009

by Miško Hevery

You hear people talking about small/medium/large/unit/integration/functional/scenario tests but do most of us really know what is meant by that? Here is how I think about tests.

Unit/Small

Lets start with unit test. The best definition I can find is that it is a test which runs super-fast (under 1 ms) and when it fails you don't need debugger to figure out what is wrong. Now this has some

The purpose of the unit-test is to check the conditional logic in your code, your 'ifs' and 'loops'. This is where the majority of your bugs come from (see theory of bugs). Which is why if you do no other testing, unit tests are the best bang for your buck! Unit tests, also make sure that you have testable code. If you have unit-testable code than all other testing levels will be testable as well.

A KeyedMultiStackTest.java is what I would consider great unit test example from Testability Explorer. Notice how each test tells a story. It is not testMethodA, testMethodB, etc, rather each test is a scenario. Notice how at the beginning the test are normal operations you would expect but as you get to the bottom of the

file the test become little stranger. It is because those are weird corner cases which I have discovered later. Now the funny thing about KeyedMultiStack.java is that I had to rewrite this class three times. Since I could not get it to work under all of the test cases. One of the test was always failing, until I realized that my algorithm was

Does each class need a unit test? A qualified no. Many classes get tested indirectly when testing something else. Usually simple value objects do not have tests of their own. But don't confuse not having tests and not having test coverage. All classes/methods should have test coverage. If you TDD, than this is automatic.

Medium/Functional

So you proved that each class works

Many of the classes are tested twice. Once directly throughout unit-test as described above, and once indirectly through the functional-tests. If you would remove the unit tests I would still have high confidence that the functional tests would catch most changes which would break things, but I would have no idea where to go to look for a fix, since the mistake can be in any class involved in the execution path. The no debugger needed rule is broken here. When a functional test fails, (and there are no unit tests failing) I am forced to take out my debugger. When I find the problem, I add a unit test

Now computing metrics is just a small part of what testability explorer does, (it also does reports, and suggestions) but those are not tested in this functional test (there are other functional tests for that). You can think of functional-tests as a set of related classes which form a cohesive functional unit for the overall application. Here are some of the functional areas in testability explorer: java byte-code parsing, java source parsing, c++ parsing, cost analysis, 3 different kinds of reports, and suggestion engine. All of these have unique set of related classes which work together and need to be tested together, but for the most part are independent.

Large/End-to-End/Scenario

We have proved that: 'ifs' and 'loops' work; and that the contracts are compatible, what else can we test? There is still one class of mistake we can make. You can wire the whole thing wrong. For example, passing in null instead of report, not configuring the location of the jar file for parsing, and so on. These are not logical bugs, but wiring bugs. Luckily, wiring bugs have this nice property that they fail consistently and usually spectacularly with an exception. Here is an example of end-to-end test: TestabilityRunnerTest.java. Notice how these tests exercises the whole application, and do not assert much. What is there to assert? We have already proven that everything works, we just want to make sure that it is wired properly.

8 comments

Labels: Misko Hevery

Yet Another JavaScript Testing Framework

Friday, May 22, 2009

by Miško Hevery & Jeremie Lenfant-engelmann

Did you notice that there are a lot of JavaScript testing frameworks out there? Why has the JavaScript community not consolidated on a single JavaScript framework the way Java has on JUnit. My feeling is that all of these frameworks are good at something but none solve the complete package. Here is what I want out of JavaScript unit-test framework:

> *I want to edit my JavaScript code in my favorite IDE, and when I hit Ctrl-S, I want all of the tests to execute across all browsers and return the results immediately.*

I don't know of any JavaScript framework out there which will let me do what I want. In order to achieve my goal I need a JavaScript framework with these three features:

Command Line Control

Most JavaScript test runners consist of JavaScript application which runs completely in the browser. What this means in practice is that I have to go to the browser and *refresh* the page to get my results. But browsers need an HTML file to display, which means that I have to write HTML file which loads all of my production code and my tests before I can run my tests. Now since browsers are sandboxes, the JavaScript tests runner can only report the result of the test run inside the browser window for human consumption only. This implies that 1) I cannot trigger running of the tests by hitting Ctrl-S in my IDE, I have to Alt-tab to Browser, hit refresh and Alt-tab back to the IDE and 2) I cannot display my test result in my IDE, the results are in the browser in human readable form only.

On my continuous build machine I need to be able to run the same tests and somehow get the failures out of the browser and on to the status page. Most JavaScript test runners have a very poor story here, which makes integrating them into a continuous build very difficult.

What we need, is the ability to control test execution from command line so that I can trigger it from my IDE, or my continuous build machine. And I need test failures

to be reported on the command line (not inside the browser where they are unreachable) so that I can display them in IDE or in continuous build status page.

Parallel Execution

Since most JavaScript test runners run fully in the browser I can only run my test on one browser at a time during my development process. In practice this means that you don't find out about failures in other browser until you have checked in the code to your continuous build machine (if you were able to set it up) and your code executes on all browsers. By that point you have completely forgotten about what you have written and debugging becomes a pain. When I run my test I want to run them on all browser platforms in parallel.

Instant Feedback in IDE

After I hit Ctrl-S on my IDE, my patience for test results is about two seconds before I start to get annoyed. What this means in practice is that you can not wait until the browser launches and runs the tests. The browser needs to be already running. Hitting refresh on your browser manually is very expensive since the browser needs to reload all of the JavaScript code an re-parse it. If you have one HTML file for each TestCase and you have hundred of these TestCases, The browser may be busy for several minutes until it reloads and re-parses the same exact production code once for each TestCase. There is no way you can fit that into the patience of average developer after hitting Ctrl-S.

Introducing JsTestDriver

Jeremie Lenfant-engelmann and I have set out to build a JavaScript test runner which solves exactly these issues so that Ctrl-S causes all of my JavaScript tests to execute in under a second on all browsers. Here is how Jeremie has made this seemingly impossible dream a reality. On startup JsTestDriver captures any number of browsers from any number of platforms and turns them into slaves. As slave the browser has your production code loaded along with all of your test code. As you edit your code and hit Ctrl-S the JsTestDriver reloads only the files which you have modified into the captured browsers slaves, this greatly reduces the amount of network traffic and amount of JavaScript re-parsing which the browser has to do and therefore greatly improves the test execution time. The JsTestDriver than runs all of your test in parallel on all captured browsers. Because JavaScript APIs are non-blocking it is almost impossible for your tests to run slow, since there is nothing to block on, no network traffic and no re-parsing of the JavaScript code. As a result JsTestDriver can easily run hundreds of TestCases per second. Once the tests execute the results are sent over the network to the command which executed the tests either on the command line ready to be show in you IDE or in your continuous build.

Demo

11 comments

Labels: JavaScript , Misko Hevery

Constructor Injection vs. Setter Injection

Thursday, February 19, 2009

by Miško Hevery

There seems to be two camps in dependency-injection: (1) The constructor-injection camp and (2) the setter-injection camp. Historically the setter-injection camp come from spring, whereas constructor-injection camp are from pico-container and GUICE. But lets leave the history behind and explore the differences in the strategies.

Setter-Injection

The basic-ideas is that you have a no argument-constructor which creates the object with "reasonable-defaults" . The user of the object can then call setters on

the object to override the collaborators of the object in order to wire the object graph together or to replace the key collaborators with test-doubles.

Constructor-Injection

The basic idea with constructor-injection is that the object has no defaults and instead you have a single constructor where all of the collaborators and values need to be supplied before you can instantiate the object.

At first it may seem that setter injection is preferred since you have no argument constructors which will make it easy for you to create the object in production and test. However, there is one non-obvious benefit with constructor injection, which in my opinion makes it a winner. Constructor-injection enforces the order of initialization and prevents circular dependencies. With setter-injection it is not clear in which order things need to be instantiated and when the wiring is done. In a typical application there may be hundreds of collaborators with at least as many setter calls to wire them together. It is easy to miss a few setter calls when wiring the application together. On the other hand constructor-injection automatically enforces the order and completeness of the instantiated. Furthermore, when the last object is instantiated the wiring phase of your application is completed. This further allows me to set the collaborators as final which makes the code easier to comprehend if you know a given field will not change during the lifetime of the application.

Let's look at an example as to how we would instantiate a CreditCardProcessor.

```
CreditCardProcessor processor = new CreditCardProcessor();
```

Great I have instantiated CreditCardProcessor, but is that enough? No, I somehow need to know to call, setOfflineQueue(). This information is not necessarily obvious.

```
OfflineQueue queue = new OfflineQueue();
CreditCardProcessor processor = new CreditCardProcessor();
processor.setOfflineQueue(queue);
```

Ok I have instantiated the OfflineQueue and remember to set the queue as a collaborator of the processor, but am I done? No, you need to set the database to both the queue and the processor.

```
Database db = new Database();
OfflineQueue queue = new OfflineQueue();
queue.setDatabase(db);
CreditCardProcessor processor = new CreditCardProcessor();
```

```
processor.setOfflineQueue(queue);
processor.setDatabase(db);
```

But wait, you are not done you need to set the Username, password and the URL on the database.

```
Database db = new Database();
db.setUsername("username");
db.setPassword("password");
db.setUrl("jdbc:....");
OfflineQueue queue = new OfflineQueue();
queue.setDatabase(db);
CreditCardProcessor processor = new CreditCardProcessor();
processor.setOfflineQueue(queue);
processor.setDatabase(db);
```

Ok, am I done now? I think so, but how do I know for sure? I know a framework will take care of it, but what if I am in a language where there is no framework, then what?

Ok, now let's see how much easier this will be in the constructor-injection. Lets instantiate CreditCardPrecossor.

```
CreditCardProcessor processor = new CreditCardProcessor(?queue?, ?db?);
```

Notice we are not done yet since CreditCardProcessor needs a queue and a database, so lets make those.

```
Database db = new Database("username", "password", "jdbc:....");
OfflineQueue queue = new OfflineQueue(db);
CreditCardProcessor processor = new CreditCardProcessor(queue, db);
```

Ok, every constructor parameter is accounted for, therefore we are done. No framework needed, to tell us that we are done. As an added bonus the code will not even compile if all of the constructor arguments are not satisfied. It is also not possible to instantiate things in the wrong order. You must instantiate Database before the OfflineQueue, since otherwise you could not make the compiler happy. I personally find the constructor-injection much easier to use and the code is much easier to read and understand.

Recently, I was building a Flex application and using the Model-View-Controller. Flex XML markup requires that components must have no argument constructors, therefore I was left with setter-injection as the only way to do dependency injection. After several views I was having hard time to keep all of the pieces wired

together properly, I was constantly forgetting to wire things together. This made the debugging hard since the application appeared to be wired together (as there are reasonable defaults for your collaborators) but the collaborators were of wrong instances and therefor the application was not behaving just right. To solve the issue, I was forced to abandon the Flex XML as a way to instantiate the application so that I can start using the constructor-injection and these issues went away.

17 comments

Labels: Misko Hevery

To Assert or Not To Assert

Monday, February 09, 2009

by Miško Hevery

Some of the strongest objections I get from people is on my stance on what I call "defensive programming". You know all those asserts you sprinkle your code with. I have a special hate relationship against null checking. But let me explain.

At first, people wrote code, and spend a lot of time debugging. Than someone came up with the idea of asserting that some set of things should never happen. Now there are two kinds of assertions, the ones where you assert that an object will never get into on inconsistent state and the ones where you assert that objects never gets passed a incorrect value. The most common of which is the null check.

Than some time later people started doing automated unit-testing, and a weird thing happened, those assertions are actually in the way of good unit testing, especially the null check on the arguments. Let me demonstrate with on example.

```
class House {
  Door door;
  Window window;
  Roof roof;
  Kitchen kitchen;
  LivingRoom livingRoom;
  BedRoom bedRoom;

  House(Door door, Window window,
            Roof roof, Kitchen kitchen,
            LivingRoom livingRoom,
            BedRoom bedRoom){
    this.door = Assert.notNull(door);
    this.window = Assert.notNull(window);
    this.roof = Assert.notNull(roof);
```

```
    this.kitchen = Assert.notNull(kitchen);
    this.livingRoom = Assert.notNull(livingRoom);
    this.bedRoom = Assert.notNull(bedRoom);
  }

  void secure() {
    door.lock();
    window.close();
  }
}
```

Now let's say that i want to test the secure() method. The secure method needs door and window. Therefore my ideal would look like this.

```
testSecureHouse() {
  Door door = new Door();
  Window window = new Window();
  House house = new House(door, window,
              null, null, null, null);

  house.secure();

  assertTrue(door.isLocked());
  assertTrue(window.isClosed());
}
```

Since the secure() method only needs to operate on door, and window, those are the only objects which I should have to create. For the rest of them I should be able to pass in null. null is a great way to tell the reader, "these are not the objects you are looking for". Compare the readability with this:

```
testSecureHouse() {
  Door door = new Door();
  Window window = new Window();
  House house = new House(door, window,
    new Roof(),
    new Kitchen(),
    new LivingRoom(),
    new BedRoom());

  house.secure();

  assertTrue(door.isLocked());
  assertTrue(window.isClosed());
}
```

If the test fails here you are now sure where to look for the problem since so many objects are involved. It is not clear from the test that that many of the collaborators are not needed.

However this test assumes that all of the collaborators have no argument constructors, which is most likely not the case. So if the Kitchen class needs dependencies in its constructor, we can only assume that the same person who put the asserts in the House also placed them in the Kitchen, LivingRoom, and BedRoom constructor as well. This means that we have to create instances of those to pass the null check, so our real test will look like this:

```
testSecureHouse() {
  Door door = new Door();
  Window window = new Window();
  House house = new House(door, window,
    new Roof(),
    new Kitchen(new Sink(new Pipes()),
           new Refrigerator()),
    new LivingRoom(new Table(), new TV(), new Sofa()),
    new BedRoom(new Bed(), new Closet()));

  house.secure();

  assertTrue(door.isLocked());
  assertTrue(window.isClosed());
}
```

Your asserts are forcing you to create so many objects which have nothing to do with the test and only confuse the reader and make the tests hard to write. Now I know that a house with a null roof, livingRoom, kitchen and bedRoom is an inconsistent object which would be an error in production, but I can write another test of my HouseFactory class which will assert that it will never happen.

Now there is a difference if the API is meant for my internal consumption or is part of an external API. For external API I will often times write tests to assert that appropriate error conditions are handled, but for the internal APIs my tests are sufficient.

I am not against asserts, I often use them in my code as well, but most of my asserts check the internal state of an object not wether or not I am passing in a null value. Checking for nulls usually goes against testability, and given a choice between well tested code and untested code with asserts, there is no debate for me which one I chose.

 27 comments

Labels: Misko Hevery

When to use Dependency Injection

Tuesday, January 20, 2009

by Miško Hevery

A great question from the reader...

> The only thing that does not fully convince me in your articles is usage of Guice. I'm currently unable to see clearly its advantages over plain factories, crafted by hand. Do you recommend using of Guice in every single case? I strongly suspect, there are cases, where hand-crafted factories make a better fit than Guice. Could you comment on that (possibly at your website)?

I think this is multi-part question:

1. Should I be using dependency-injection?
2. Should I be using manual dependency-injection or automatic dependency-injection framework?
3. Which automatic dependency-injection framework should I use?

Should I be using dependency-injection?

The answer to this question should be a resounding **yes**! We covered this many times how to think about the new-operator, singletons are liars, and of course the talk on dependency-injection.

Dependency injection is simply a good idea and it helps with: testability; maintenance; and bringing new people up to speed on new code-base. Dependency-injection helps you with writing good software whether it is a small project of one or large project with a team of collaborators.

Should I be using manual dependency-injection or automatic dependency-injection framework?

Whether or not to use a framework for dependency injection depends a lot on your preferences and the size of your project. You don't get any additional magical powers by using a framework. I personally like to use frameworks on medium to large projects but stick to manual DI with small projects. Here are some arguments both ways to help you make a decision.

In favor of manual DI:

- Simple: Nothing to learn, no dependencies.
- No reflection magic: In IDE it is easy to find out who calls the constructors.
- Even developers who do not understand DI can follow and contribute to projects.

In favor of automatic DI framework:

- Consistency: On a large team a lot can be said in doing things in consistent manner. Frameworks help a lot here.
- Declarative: The wiring, scopes and rules of instantiation are declarative. This makes it easier to understand how the application is wired together and easier to change.
- Less typing: No need to create the factory classes by hand.
- Helps with end-to-end tests: For end-to-end tests we often need to replace key components of the application with fake implementations, an automated framework can be of great help.

Which automatic dependency-injection framework should I use?

There are three main DI frameworks which I am aware off: GUICE, Pico Container and Spring.

I work for Google, I have used GUICE extensively therefor my default recommendation will be GUICE. :-) However I am going to attempt to be objective about the differences. Keep in mind that I have not actually used the other ones on real projects.

Spring was first. As a result it goes far beyond DI and has everything and kitchen sink integrated into it which is very impressive. The DI part of Spring has some differences worth pointing out. Unlike GUICE or Pico, Spring uses XML files for configuration. Both are declarative but GUICE is compiled and as a result GUICE can take advantage of compiler type safety and generics, which I think is a great plus for GUICE.

Historically, Spring started with setter injection. Pico introduced constructor injection. Today, all frameworks can do both setter and constructor injection, but the developers using these frameworks still have their preferences. GUICE and Pico strongly prefer constructor injection while Spring is in the setter injection camp. I prefer constructor injection but the reasons are better left for another post.

Personally, I think all of the three have been around for a while and have proven themselves extensively, so no matter which one you chose you will benefit greatly

from your decision. All three frameworks have been heavily influenced by each other and on a macro level are very similar.

Your milage may very.

13 comments

Labels: Misko Hevery

Interfacing with hard-to-test third-party code

Wednesday, January 07, 2009

by Miško Hevery

Shahar asks an excellent question about how to deal with frameworks which we use in our projects, but which were not written with testability in mind.

> Hi Misko, First I would like to thank you for the "Guide to Writing Testable Code", which really helped me to think about better ways to organize my code and architecture. Trying to apply the guide to the code I'm working on, I came up with some difficulties. Our code is based on external frameworks and libraries. Being dependent on external frameworks makes it harder to write tests, since test setup is much more complex. It's not just a single class we're using, but rather a whole bunch of classes, base classes, definitions and configuration files. Can you provide some tips about using external libraries or frameworks, in a manner that will allow easy testing of the code?
>
> -- Thanks, Shahar

There are two different kind of situations you can get yourself into:

1. Either your code calls a third-party library (such as you calling into LDAP authentication, or JDBC driver)
2. Or a third party library calls you and forces you to implement an interface or extend a base class (such as when using servlets).

Unless these APIs are written with testability in mind, they will hamper your ability to write tests.

Calling Third-Party Libraries

I always try to separate myself from third party library with a Facade and an Adapter. Facade is an interface which has a simplified view of the third-party API.

Let me give you an example. Have a look at `javax.naming.ldap`. It is a collection of several interfaces and classes, with a complex way in which you have to call them. If your code depends on this interface you will drown in mocking hell. Now I don't know why the API is so complex, but I do know that my application only needs a fraction of these calls. I also know that many of these calls are configuration specific and outside of bootstrapping code these APIs are cluttering what I have to mock out.

I start from the other end. I ask myself this question. 'What would an ideal API look like for my application?' The key here is 'my application' An application which only needs to authenticate will have a very different 'ideal API' than an application which needs to manage the LDAP. Because we are focusing on our application the resulting API is significantly simplified. It is very possible that for most applications the ideal interface may be something along these lines.

```
interface Authenticator {
 boolean authenticate(String username,
                      String password);
}
```

As you can see this interface is a lot simpler to mock and work with than the original one as a result it is a lot more testable. In essence the ideal interfaces are what separates the testable world from the legacy world.

Once we have an ideal interface all we have to do is implement the adapter which bridges our ideal interface with the actual one. This adapter may be a pain to test, but at least the pain is in a single location.

The benefit of this is that:

- We can easily implement an `InMemoryAuthenticator` for running our application in the QA environment.
- If the third-party APIs change than those changes only affect our adapter code.
- If we now have to authenticate against a Kerberos or Windows registry the implementation is straight forward.
- We are less likely to introduce a usage bug since calling the ideal API is simpler than calling the original API.

Plugging into an Existing Framework

Let's take servlets as an example of hard to test framework. Why are servlets hard to test?

- Servlets require a no argument constructor which prevents us from using dependency injection. See how to think about the new operator.
- Servlets pass around `HttpServletRequest` and `HttpServletResponse` which are very hard to instantiate or mock.

At a high level I use the same strategy of separating myself from the servlet APIs. I implement my actions in a separate class

```
class LoginPage {
 Authenticator authenticator;
 boolean success;
 String errorMessage;
 LoginPage(Authenticator authenticator) {
   this.authenticator = authenticator;
 }

 String execute(Map<String, String> parameters,
                String cookie) {
   // do some work
   success = ...;
   errorMessage = ...;
 }

 String render(Writer writer) {
   if (success)
     return "redirect URL";
   else
     writer.write(...);
 }
}
```

The code above is easy to test because:

- It does not inherit from any base class.
- Dependency injection allows us to inject mock authenticator (Unlike the no argument constructor in servlets).
- The work phase is separated from the rendering phase. It is really hard to assert anything useful on the Writer but we can assert on the state of the `LoginPage`, such as `success` and `errorMessage`.
- The input parameters to the `LoginPage` are very easy to instantiate. (`Map<String, String>`, `String` for cookie, or a `StringWriter` for the writer).

What we have achieved is that all of our application logic is in the `LoginPage` and all of the untestable mess is in the `LoginServlet` which acts like an adapter. We

can than test the `LoginPage` in depth. The `LoginSevlet` is not so simple, and in most cases I just don't bother testing it since there can only be wiring bug in that code. There should be no application logic in the `LoginServlet` since we have moved all of the application logic to `LoginPage`.

Let's look at the adapter class:

```
class LoginServlet extends HttpServlet {
 Provider<LoginPage> loginPageProvider;

 // no arg constructor required by
 // Servlet Framework
 LoginServlet() {
   this(Global.injector
          .getProvider(LoginPage.class));
 }

 // Dependency injected constructor used for testing
 LoginServlet(Provider<LoginPage> loginPageProvider) {
   this.loginPageProvider = loginPageProvider;
 }

 service(HttpServletRequest req,
         HttpServletResponse resp) {
   LoginPage page = loginPageProvider.get();
   page.execute(req.getParameterMap(),
        req.getCookies());
   String redirect = page.render(resp.getWriter())
   if (redirect != null)
     resp.sendRedirect(redirect);
 }
}
```

Notice the use of two constructors. One fully dependency injected and the other no argument. If I write a test I will use the dependency injected constructor which will than allow me to mock out all of my dependencies.

Also notice that the no argument constructor is forcing me to use global state, which is very bad, but in the case of servlets I have no choice. However, I make sure that only servlets access the global state and the rest of my application is unaware of this global variable and uses proper dependency injection techniques.

BTW there are many frameworks out there which sit on top of servlets and which provide you a very testable APIs. They all achieve this by separating you from the servlet implementation and from `HttpServletRequest` and `HttpServletResponse`. For example Waffle and WebWork

1 comment

Labels: Java , Misko Hevery

Static Methods are Death to Testability

Wednesday, December 17, 2008

by Miško Hevery

Recently many of you, after reading Guide to Testability, wrote to telling me there is nothing wrong with static methods. After all what can be easier to test than `Math.abs()`! And `Math.abs()` is static method! If `abs()` was on instance method, one would have to instantiate the object first, and that may prove to be a problem. (See how to think about the new operator, and class does real work)

The basic issue with static methods is they are procedural code. I have no idea how to unit-test procedural code. Unit-testing assumes that I can instantiate a piece of my application in isolation. During the instantiation I **wire** the dependencies with mocks/friendlies which replace the real dependencies. With procedural programing there is **nothing to "wire"** since there are no objects, the code and data are separate.

Here is another way of thinking about it. Unit-testing needs seams, seams is where we prevent the execution of normal code path and is how we achieve isolation of the class under test. seams work through polymorphism, we override/implement class/interface and than wire the class under test differently in order to take control of the execution flow. With static methods there is nothing to override. Yes, static methods are easy to call, but if the static method calls another static method there is no way to overrider the called method dependency.

Lets do a mental exercise. Suppose your application has nothing but static methods. (Yes, code like that is possible to write, it is called procedural programming.) Now imagine the call graph of that application. If you try to execute a leaf method, you will have no issue setting up its state, and asserting all of the corner cases. The reason is that a leaf method makes no further calls. As you move further away from the leaves and closer to the root `main()` method it will be harder and harder to set up the state in your test and harder to assert things. Many things will become impossible to assert. Your tests will get progressively larger. Once you reach the `main()` method you no longer have a unit-test (as your unit is the whole application) you now have a scenario test. Imagine that the application you are trying to test is a word processor. There is not much you can assert from the main method.

We have already covered that global state is bad and how it makes your application hard to understand. If your application has no global state than all of the input for your static method must come from its arguments. Chances are very good that you can move the method as an instance method to one of the method's arguments. (As in `method(a,b)` becomes `a.method(b)`.) Once you move it you realized that that is where the method should have been to begin with. The use of static methods becomes even worse problem when the static methods start accessing the global state of the application. What about methods which take no arguments? Well, either methodX() returns a constant in which case there is nothing to test; it accesses global state, which is bad; or it is a factory.

Sometimes a static methods is a factory for other objects. This further exuberates the testing problem. In tests we rely on the fact that we can wire objects differently replacing important dependencies with mocks. Once a `new` operator is called we can not override the method with a sub-class. A caller of such a static factory is permanently bound to the concrete classes which the static factory method produced. In other words the damage of the static method is far beyond the static method itself. Butting object graph wiring and construction code into static method is extra bad, since object graph wiring is how we isolate things for testing.

"So leaf methods are ok to be static but other methods should not be?" I like to go a step further and simply say, static methods are not OK. The issue is that a methods starts off being a leaf and over time more and more code is added to them and they lose their positions as a leafs. It is way to easy to turn a leaf method into none-leaf method, the other way around is not so easy. Therefore a static leaf method is a slippery slope which is waiting to grow and become a problem. Static methods are procedural! In OO language stick to OO. And as far as `Math.abs(-5)` goes, I think Java got it wrong. I really want to write `-5.abs()`. Ruby got that one right.

27 comments

Labels: Misko Hevery

Clean Code Talks - Inheritance, Polymorphism, & Testing

Monday, December 08, 2008

by Miško Hevery

Google Tech Talks

November 20, 2008

ABSTRACT

Is your code full of if statements? Switch statements? Do you have the same switch statement in various places? When you make changes do you find yourself making the same change to the same if/switch in several places? Did you ever forget one?

This talk will discuss approaches to using Object Oriented techniques to remove many of those conditionals. The result is cleaner, tighter, better designed code that's easier to test, understand and maintain.

Video

Slides

4 comments

Labels: Misko Hevery

Guide to Writing Testable Code

Wednesday, November 26, 2008

It is with great pleasure that I have been able to finally open-source the Guide to Writing Testable Code.

I am including the first page here for you, but do come and check it out in detail.

To keep our code at Google in the best possible shape we provided our software engineers with these constant reminders. Now, we are happy to share them with the world.

Many thanks to these folks for inspiration and hours of hard work getting this guide done:

- Jonathan Wolter
- Russ Ruffer
- Miško Hevery

Flaw #1: Constructor does Real Work

Warning Signs

- `new` keyword in a constructor or at field declaration
- Static method calls in a constructor or at field declaration
- Anything more than field assignment in constructors
- Object not fully initialized after the constructor finishes (watch out for `initialize` methods)
- Control flow (conditional or looping logic) in a constructor
- Code does complex object graph construction inside a constructor rather than using a factory or builder
- Adding or using an initialization block

Flaw #2: Digging into Collaborators

Warning Signs

- Objects are passed in but never used directly (only used to get access to other objects)
- Law of Demeter violation: method call chain walks an object graph with more than one dot (`.`)
- Suspicious names: `context`, `environment`, `principal`, `container`, or manager

Flaw #3: Brittle Global State & Singletons

Warning Signs

- Adding or using singletons
- Adding or using static fields or static methods
- Adding or using static initialization blocks
- Adding or using registries
- Adding or using service locators

Flaw #4: Class Does Too Much

Warning Signs

- Summing up what the class does includes the word “and”
- Class would be challenging for new team members to read and quickly “get it”
- Class has fields that are only used in some methods
- Class has static methods that only operate on parameters

5 comments

Labels: Misko Hevery

Clean Code Talks - Global State and Singletons

Friday, November 21, 2008

by Miško Hevery

Google Tech Talks

November 13, 2008

ABSTRACT

Clean Code Talk Series

Topic: Global State and Singletons

Speaker: Miško Hevery

Video

Slides

No comments

Labels: Misko Hevery

My Unified Theory of Bugs

Tuesday, November 18, 2008

by Miško Hevery

I think of bugs as being classified into three fundamental kinds of bugs.

- **Logical**: Logical bug is the most common and classical "bug." This is your "if"s, "loop"s, and other logic in your code. It is by far the most common kind of bug in an application. (*Think*: it does the wrong thing)
- **Wiring**: Wiring bug is when two different objects are miswired. For example wiring the first-name to the last-name field. It could also mean that the output of one object is not what the input of the next object expects. (*Think*: Data gets clobbered in process to where it is needed.)
- **Rendering**: Rendering bug is when the output (typical some UI or a report) does not look right. The key here is that it takes a human to determine what "right" is. (*Think*: it "looks" wrong)

NOTE: A word of caution. Some developers think that since they are building UI everything is a rendering bug! A rendering bug would be that the button text overlaps with the button border. If you click the button and the wrong thing happens than it is either because you wired it wrong (wiring problem) or your logic is wrong (a logical bug). Rendering bugs are rare.

Typical Application Distribution (without Testability in Mind)

The first thing to notice about these three bug types is that the probability is not evenly distributed. Not only is the probability not even, but the cost of finding and fixing them is different. (I am sure you know this from experience). My experience from building web-apps tells me that the Logical bugs are by far the most common, followed by wiring and finally rendering bugs.

Cost of Finding the Bug

Logical bugs are notoriously hard to find. This is because they only show up when the right set of input conditions are present and finding that magical set of inputs or reproducing it tends to be hard. On the other hand wiring bugs are much easier to spot since the wiring of the application is mostly fixed. So if you made a wiring error, it will show up every time you execute that code, for the most part independent of input conditions. Finally, the rendering bugs are the easiest. You simply look at the page and quickly spot that something "looks" off.

Cost of Fixing the Bug

Our experience also tells us how hard it is to fix things. A logical bug is hard to fix, since you need to understand all of the code paths before you know what is wrong and can create a solution. Once the solution is created, it is really hard to be sure that we did not break the existing functionality. Wiring problems are much simpler, since they either manifest themselves with an exception or data in wrong location. Finally rendering bugs are easy since you "look" at the page and immediately know what went wrong and how to fix it. The reason it is easy to fix is that we design our application knowing that rendering will be something which will be constantly changing.

	Logical	Wiring	Rendering
Probability of Occurrence	High	Medium	Low
Difficulty of Discovering	Difficult	Easy	Trivial
Cost of Fixing	High Cost	Medium	Low

How does testability change the distribution?

It turns out that testable code has effect on the distribution of the bugs. Testable code needs:

- Clear separation between classes (Testable Seams) --> clear separation between classes makes it less likely that a wiring problem is introduced. Also, less code per class lowers the probability of logical bug.
- Dependency Injection --> makes wiring explicit (unlike singletons, globals or service locators).
- Clear separation of Logic from Wiring --> by having wiring in a single place it is easier to verify.

The result of all of this is that the number of wiring bugs are significantly reduced. (So as a percentage we gain Logical Bugs. However total number of bugs is decreased.)

The interesting thing to notice is that you can get benefit from testable code without writing any tests. Testable code is better code! (When I hear people say that they sacrificed "good" code for testability, I know that they don't really understand testable-code.)

We Like Writing Unit-Tests

Unit-tests give you greatest bang for the buck. A unit test focuses on the most common bugs, hardest to track down and hardest to fix. And a unit-test forces you to write testable code which indirectly helps with wiring bugs. As a result when writing automated tests for your application we want to overwhelmingly focus on unit test. Unit-tests are tests which focus on the logic and focus on one class/method at a time.

- Unit-tests focus on the logical bugs. Unit tests focus on your "if"s and "loop"s, a Focused unit-test does not directly check the wiring. (and certainly not rendering)
- Unit-test are focused on a single CUT (class-under-test). This is important, since you want to make sure that unit-tests will not get in the way of future refactoring. Unit-tests should HELP refactoring not PREVENT refactorings. (Again, when I hear people say that tests prevent refactorings, I know that they have not understood what unit-tests are)
- Unit-tests do not directly prove that wiring is OK. They do so only indirectly by forcing you to write more testable code.
- Functional tests verify wiring, however there is a trade-off. You "may" have hard time refactoring if you have too many functional test OR, if you mix functional and logical tests.

Managing Your Bugs

I like to think of tests as bug management. (with the goal of bug free) Not all types of errors are equally likley, therefore I pick my battles of which tests I focus on. I find that I love unit-tests. But they need to be focused! Once a test starts testing a lot of classes in a single pass I may enjoy high coverage, but it is really hard to figure out what is going on when the test is red. It also may hinder refactorings. I tend to go very easy on Functional tests. A single test to prove that things are wired together is good enough to me.

I find that a lot of people claim that they write unit-tests, but upon closer inspection it is a mix of functional (wiring) and unit (logic) test. This happens becuase people wirte tests after code, and therefore the code is not testable. Hard to test code tends to create mockeries. (A mockery is a test which has lots of mocks, and mocks returning other mocks in order to execute the desired code) The result of a mockery is that you prove little. Your test is too high level to assert anything of interest on method level. These tests are too intimate with implementation (the intimace comes from too many mocked interactions) making any refactorings very painful.

 6 comments

Labels: Misko Hevery

Clean Code Talks - Dependency Injection

Tuesday, November 11, 2008

by Miško Hevery

Google Tech Talks

November 6, 2008

ABSTRACT

Clean Code Talk Series

Topic: Don't Look For Things!

Speaker: Miško Hevery

Video

Slides

3 comments

Labels: Misko Hevery

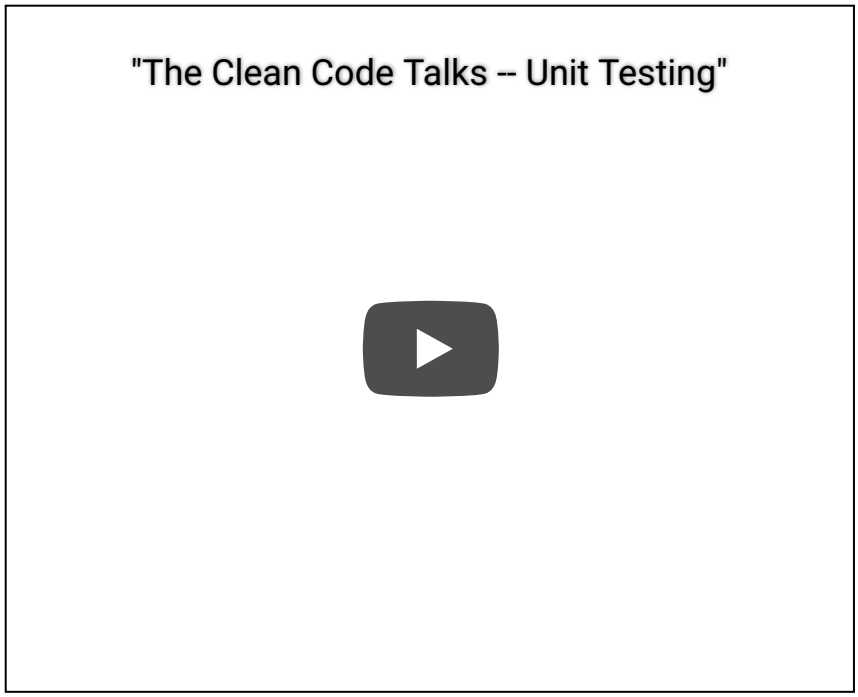
Clean Code Talks - Unit Testing

Wednesday, November 05, 2008

by Miško Hevery

Google Tech Talks October, 30 2008 ABSTRACT Clean Code Talks - Unit Testing Speaker: Misko Hevery

Video



Slides

12 comments

Labels: Misko Hevery

Testability Explorer: Measuring Testability

Monday, October 27, 2008

Testability Explorer: Using Byte-Code Analysis to Engineer Lasting Social Changes in an Organization's Software Development Process. (Or How to Get Developers to Write Testable Code)

Presented at 2008 OOPSLA by Miško Hevery a Best Practices Coach @ Google

Abstract

Testability Explorer is an open-source tool that identifies hard-to-test Java code. Testability Explorer provides a repeatable objective metric of "testability." This metric becomes a key component of engineering a social change within an organization of developers. The Testability Explorer report provides actionable information to developers which can be used as (1) measure of progress towards a goal and (2) a guide to refactoring towards a more testable code-base.

Keywords: unit-testing; testability; refactoring; byte-code analysis; social engineering.

1. Testability Explorer Overview

In order to unit-test a class, it is important that the class can be instantiated in isolation as part of a unit-test. The most common pitfalls of testing are (1) mixing object-graph instantiation with application-logic and (2) relying on global state. The Testability Explorer can point out both of these pitfalls.

1.1 Non-Mockable Cyclomatic Complexity

Cyclomatic complexity is a count of all possible paths through code-base. For example: a main method will have a large cyclomatic complexity since it is a sum of all of the conditionals in the application. To limit the size of the cyclomatic complexity in a test, a common practice is to replace the collaborators of class-under-test with mocks, stubs, or other test doubles.

Let's define "non-mockable cyclomatic complexity" as what is left when the class-under-test has all of its accessible collaborators replaced with mocks. A code-base where the responsibility of object-creation and application-logic is separated (using Dependency Injection) will have high degree of accessible collaborators; as a result most of its collaborators will easily be replaceable with mocks, leaving only the cyclomatic complexity of the class-under-test behind.

In applications, where the class-under-test is responsible for instantiating its own collaborators, these collaborators will not be accessible to the test and as a result will not be replaceable for mocks. (There is no place to inject test doubles.) In such classes the cyclomatic complexity will be the sum of the class-under-test and its non-mockable collaborators.

The higher the non-mockable cyclomatic complexity the harder it will be to write a unit-test. Each non-mockable conditional translates to a single unit of cost on the

Testability Explorer report. The cost of static class initialization and class construction is automatically included for each method, since a class needs to be instantiated before it can be exercised in a test.

1.2 Transitive Global-State

Good unit-tests can be run in parallel and in any order. To achieve this, the tests need to be well isolated. This implies that only the stimulus from the test has an effect on the code execution, in other words, there is no global-state.

Global-state has a transitive property. If a global variable refers to a class than all of the references of that class (and all of its references) are globally accessible as well. Each globally accessible variable, that is not final, results in a cost of ten units on the Testability Explorer.

2. Testability Explorer Report

A chain is only as strong as its weakest link. Therefore the cost of testing a class is equal to the cost of the class' costliest method. In the same spirit the application's overall testability is de-fined in terms of a few un-testable classes rather than a large number of testable ones. For this reason when computing the overall score of a project the un-testable classes are weighted heavier than the testable ones.

3. How to Interpret the Report

By default the classes are categorized into three categories: "Excellent" (green) for classes whose cost is below 50; "Good" (yellow) for classes whose cost is below 100; and "Needs work" (red) for all other classes. For convenience the data is presented as both a pie chart and histogram distribution and overall (weighted average) cost shown on a dial.

```
[-]ClassRepository [ 323 ]
 [-]ClassInfo getClass(String) [ 323 ]
   line 51:
```

```
    ClassInfo parseClass(InputStream) [318]
    InputStream inputStreamForClass(String) [2]
[-]ClassInfo parseClass(InputStream) [318]
  line 77: void accept(ClassVisitor, int) [302]
  line 75: ClassReader(InputStream) [15]
```

Clicking on the class ClassRepository allows one to drill down into the classes to get more information. For example the above report shows that ClassRepository has a high cost of 318 due to the parseClass(InputStream) method. Looking in closer we see that the cost comes from line 77 and an invocation of the accept() method.

```
73:ClassInfo parseClass(InputStream is) {
74:  try {
75:    ClassReader reader = new ClassReader(is);
76:    ClassBuilder v = new ClassBuilder (this);
77:    reader.accept(v, 0);
78:    return visitor.getClassInfo();
79:  } catch (IOException e) {
80:    throw new RuntimeException(e);
81:  }
82:}
```

As you can see from the code the ClassReader can never be replaced for a mock and as a result the cyclomatic complexity of the accept method can not be avoided in a test — resulting in a high testability cost. (Injecting the ClassReader would solve this problem and make the class more test-able.)

4. Social Engineering

In order to produce a lasting change in the behavior of developers it helps to have a measurable number to answer where the project is and where it should be. Such information can provide in-sight into whether or not the project is getting closer or farther from its testability goal.

People respond to what is measured. Integrating the Testability Explorer with the project's continuous build and publishing the report together with build artifacts communicate the importance of testability to the team. Publishing a graph of overall score over time allows the team to see changes on per check-in basis.

If Testability Explorer is used to identify the areas of code that need to be refactored, than compute the rate of improvement and project expected date of

refactoring completion and create a sense of competition among the team members.

It is even possible to set up a unit test for Testability Explorer that will only allow the class whose testability cost is better than some predetermined cost.

5 comments

Labels: Misko Hevery