TheBowlingGameKata [add child]

The Bowling Game Kata

Here is a kata for the Bowling Game problem. I have broken it down into the same tiny little steps that I do when I demonstrate it. However, as is usual for a kata, I have left out most of the explanatory comments.

A kata is meant to be memorized. Students of a kata study it as a *form*, not as a conclusion. It is not the conclusion of the kata that matters, it's the steps that lead to the conclusion. If you want to lean to think the way I think, to design the way I design, then you must learn to react to minutia the way I react. Following this form will help you to do that. As you learn the form, and repeat it, and repeat it, you will condition your mind and body to respond the way *I* respond to the minute factors that lead to design decisions.

Michael Feathers has long pondered the concept of "Design Sense". Good designers have a "sense" for design. They can convert a set of requirements into a design with little or not effort. It's as though their minds were wired to translate requirements to design. They can "feel" when a design is good or bad. They somehow intrinsically know which design alternatives to take at which point.

Perhaps the best way to acquire "Design Sense" is to find someone who has it, put your fingers on top of theirs, put your eyeballs right behind theirs, and follow along as they design something. Learning a kata may be one way of accomplishing this.

If you wish to try this style of learning, I suggest you proceed by memorizing it in short sections. Fully learn one section before adding the next. I have broken the kata up into five short sections. Learn each in order, and don't learn the next until you have mastered the previous. Move slowly and deliberately. DO NOT RUSH. A kata needs to seep into your bones, and this take time.

Here are the sections to memorize:

- The First Test
- The Second Test
- The Third Test
- The Fourth Test
- The Fifth Test

There is also a preamble section entitled "A Quick Design Session". This is part of the kata when demonstrating TDD to others, but is not part of the "Design Sense" of the kata itself.

!commentForm

Mon, 27 Jun 2005 04:15:22, Brian Farrar, This is Great! +1 Expand All | Collapse All

For how I learn, this teaching method is ideal. Concrete, repetitive practice, one tiny, easy, understandable step at a time. I am able to concentrate.

I did the First Test thirty times. It was tough. The first time I went through the kata, fully armed with the thought "Just do the First Test", I quickly found myself on the "Third Test" slide, it all just happened so fast. Even though I think I have a pretty good grip on it by now, I am sure there is still some illuminating revelation waiting if I do it another thirty times. I could go on more about how good doing this feels... but I'll spare it and instead torture with some humble notes I made from the ongoing experience:

- Deleting Code. The act of deleting the code at the end of every round was the most valuable thing I have learned so far from the First Test. Wow. Doing "all that work" and then just throwing it away, letting it go (it's really just a couple of lines of code, but still...). Practicing being able to evade my crying ego and just say goodbye to "my precious" was not only liberating but good practice at something so many of us find so hard to do.

- No Assert First. I am currently exploring the assert first technique. The First Test does not make use of it. I tried it a couple of times this way and I found it to be more awkward. It took a lot more steps and I got to the green bar without writing the for loop. I guess that's OK. It just felt weird. But maybe just because I had done it the other way so many times.

Wed, 29 Jun 2005 18:50:40, Shane Mingins, In Other Words? Expand All | Collapse All

Perhaps I am missing something in the analogy? Does this kata not equal Beck's "rules of simple code" which are:

1. Runs all the tests
2. Contains no duplication
3. Expresses all the important design intentions
4. Minimizes entities (e.g. classes and methods).

Cheers
Shane

Fri, 8 Jul 2005 06:07:14, Matteo Vaccari, going too fast? :) Expand All | Collapse All

I assume that when you do this kata, you are supposed to run the tests after completing each slide. Right?

Between slide 22 and 23, you are
- reimplementing the algorithm in Game, so that testAllOnes works
- factoring Game creation out from the two tests into setUp()

Do you actually do both things before running the tests? I tend to rewrite the algorithm and check the green bar before doing any refactoring.

Matteo

Tue, 12 Jul 2005 16:34:46, Uncle Bob, Going too fast. Expand All | Collapse All

Matteo,
I actually do run the tests more often than the Kata shows. Yes, indeed, I would have run the tests many times between the refactoring of the redundant game, and the new algorithm for making testAllOnes work.

Thu, 21 Jul 2005 12:44:30, Matteo Vaccari, tiny improvements Expand All | Collapse All

I find this kata is a great tool! When I do it I make too tiny improvements:
I think "frameIndex" is not a good name; it does not index frames, but rolls. I use "rollIndex"

Also, I don't like "currentRoll"; it has too much of an operational flavor. I like "rollCount" better, becouse it holds the count of how many rolls have been made.
At least, *I* think they are improvements ;) Thanks for sharing these things.

Wed, 10 Aug 2005 02:33:38, Tim, is there an error? Expand All | Collapse All

Thanks for posting this. It is very helpful in learning unit testing
It seems like each test should call setUp() but they never do.

- *JUnit automatically calls setUp() at the start of each test == UB.*

Thu, 27 Oct 2005 03:22:03, Raffaele Petracca, bowling example Expand All | Collapse All

Why implementation is completely different from design ?

- *Excellent question! It's because we could not see how simple the solution really was when we drew that design! == UB*

Wed, 16 Nov 2005 03:05:40, , Expand All | Collapse All

Fri, 20 Jan 2006 02:58:22, Jon Skeet, Slide 38 - new test? Expand All | Collapse All

(I don't know if anyone's still reading this to answer, but anyway...)

I'm slightly confused by slide 38. In the process of adding a simple way to make a test pass, you've thought of a reason why it wouldn't work. For slide 39, you've then backed out the test and the changes, then gone on to correct things in slide 40.

Now, I'd have thought it would be better to keep going with the "naive" way of getting the third test to pass, having written down on an index card (or other to-do list) that another test is needed. When the bar goes green, add another test which demonstrates the concern you've just thought of (eg a sequence of rolls of 3, 4, 6, 2, 0...) which would then go red. That would prevent anyone who didn't understand why the code needed to go in pairs from "simplifying" it to the broken code.

Now, I'm still pretty new to this whole area, so I'm sure it's me that's missed something rather than Uncle Bob. Can anyone enlighten me?

Wed, 3 May 2006 17:03:29, bowling, Expand All | Collapse All

Wed, 19 Jul 2006 14:00:32, Thomas Nilsson, From roll to frame Expand All | Collapse All

I found this kata-walkthrough excellent. It helped me a lot in getting a deeper understanding of TDD. However, I was also a bit puzzled by some of the steps, which I found a bit direct for me to understand the reasoning behind (the comment on slide 38 is one of the examples). Running through it a number of times I learned by my own thoughts, actions and mistakes what Uncle Bob actually meant.

As for slide 38, after a number of run throughs, I found that, although Uncle Bobs comment is true, I would actually rather take a different next step. First, after realizing the problem (that spares are triggered although the two rools are not in the same frame) I'd back out of the failing test. But then I would have created a test to exhibit this problem, e.g.

```
public void testTenPinsNotSpare() {
g.roll(0);
g.roll(7);
g.roll(3);
g.roll(2);
rollMany(16, 0);
assertEquals(12, g.score()));
}
```

This will exhibit the faulty behaiour and drive the fix that is performed in the subsequent slides (which Uncle Bob does without having a failing test to drive them... ;-)

Just my 0.02$.

/Thomas

Wed, 19 Jul 2006 17:54:47, David Chelimsky, re: From roll to frame Expand All | Collapse All

Thomas - there's a fair amount of refactoring one must do to get through that step. The problem w/ leaving the test failing is that you have to refactor (a bunch) in the red, rather than in the green - a TDD no-no.

One time I was going through this w/ some students and one indicated that he thought commenting that test is cheap. So we did this (not pretty - but viable).

```
//in GameTest.java
public void testAllFivesGameScoreShouldBe150() {
  rollMany(5, 21);
  assertEquals(150, game.score());
}

//in Game.java
public void roll(int pins) {
  if (pins == 5) score = 150;
  else score += pins;
}
```

This allowed us to keep that test in place and refactor in the green. In retrospect, I'm not sure that this is any less cheap than commenting the test because it really hides the fact that the implementation is not calculating the correct answer until you remove that line (if (pins == 5) score = 150)). But it did work, and did allow us to progress.

Thu, 20 Jul 2006 04:21:30, Thomas Nilsson, re: From roll to frame Expand All | Collapse All

Sorry, I was unclear. Of course I would have removed the failing test before refactoring. I just think that the refactoring steps at this point is to large. As we obviously have a discovered erroneous behaviour, I proposed a test that will be more precise in pinpointing which behaviour we are trying to fix. Of course we have to still comment out that test also to be able to refactor safely, but once refactored we can by uncommenting testTenNotSpare() ensure that the fault behaviour is removed.

My point just that the suggested step contains both refactoring to remove faulty behaviour and introduction of new functionality, which in my mind seems to be going a bit fast.

I suppose, the question is perhaps how the discovery of this faulty behaviour should be handled. Uncle Bob proposes (as I see it) to simply ignore that discovery and let the requirements of the next new functional step drive the refactoring. I propose to exhibit it in a test which will fail with the current implementation. (Again, it will have to be commented out to be able to do the refactoring.)

Also, I do not agree with your student in thinking that commenting out the test is cheap. Actually, that is one of the most important lessons for me in this kata, a simple way to back out to a green bar. It also makes it obvious for anyone that this test is a next step. I suppose that a good IDE with JUnit Test Case support (like Eclipse) could offer a simple "temporarily remove test" function. (Yes, I know about "Toogle Comment", but then you have to mark the whole test...)