Software Testing for Developers
By Udi Dahan – http://www.UdiDahan.com

In my first job as a developer, I remember spending most of my time getting the code to compile. Since I was writing in C++, this was fairly understandable. It didn't take me long to understand that code that compiles doesn't necessarily run. It just came to me from the blue one day – when my manager was yelling at me for screwing up the project schedule. "*Done* means that the code runs", he said, and that's when I started testing my code.

I'm pretty sure that around that time there were various smart people already doing unit testing and test-driven development, but I wasn't one of them. I just looked at how other developers tested their code and did the same. This was usually in the form of creating some kind of GUI, whose buttons called into the code that I wanted to test. I'd then step-through the code using the debugger, setting up "watches", looking at their values – the usual stuff. The quality of my work went right up.

Around the time that we were trying to stabilize the version, the testers came in. A hail of bugs started falling, and there was no end in sight. Management wasn't happy. After a couple of months of hellish bug fixing, we had another version ready for the testers. You can probably imagine that the hail kept falling. Another month went by, another version was given to the testers, and then the impossible happened. I got a bug assigned to me that I fixed 2 minor versions ago. I went to the tester who had opened the bug, Jay - a reasonable guy, and explained that there must have been some mistake – I had already fixed that bug. He was a bit older and wiser than me, and explained the way of the world.

I was shocked, could it be that one of my more recent changes had broken old code? Jay reminded me of the principle that guides testers: If it ain't tested, it don't work. This was quite a bit different from the "if it ain't broke, don't fix it" maxim that guided me as a developer. Of course the old code works – I mean, why shouldn't it? It worked before. When I hear these phrases today from our more junior developers, I'm reminded of times gone by. College just doesn't prepare you for the real world.

The real world doesn't care how many lines of code developers write. Productivity is measured in terms of features. That's *done* features. It turns out that a feature isn't done until a tester checks. The worst part is when I *finally* got a feature to work, and then one month later it stopped working and I had no idea why. Well, I knew why – sometime in that last month, I broke the feature, but I fiddled with a lot of code that month so I had no idea where to start looking. Wouldn't it be great, I wondered, if something could tell me when I broke something.

Several months later, I went to a conference where I met a bunch of smart people. One of them, Morris, was showing off some particularly clever code that he had written, and had this GUI that he used to run it. He told me that it

was JUnit, and that it wasn't directly running his code, but rather running the tests he had wrote. For a second, I was enthusiastic. This generic GUI could run any code I gave it – that's a lot of GUIs that I wouldn't have to write anymore. That enthusiasm dimmed when I realized that I still had to write the code that ran my code – the code behind the button in my old GUIs wouldn't go away. So what's the big deal, I wondered. Not having to put up a button isn't going to do much for my productivity, not in terms of lines of code or features.

"You're missing the point", Morris told me. "I'm writing *unit tests*."

My blank stare made it clear I wasn't getting the point.

"You know how you step through your code in the debugger to make sure it works OK?". I nodded. "Well, I'm just automating that", Morris explained. "Everything you check manually, I write code that checks for it. When you set up a watch on a variable, you do that so that you'll know if its value is right or not. I do the same thing by writing code like this:"

```
if (x != 5) fail test
```

"You're writing more code, and you want me to believe that's better?" Everybody knew that less code is better – less code to maintain, less bugs, this would only make things worse.

"Stay with me", Morris urged, "I'm getting to the point."
I shrugged. There were 10 more minutes until the next presentation and I had nothing better to do.

"It might take me longer to automate the tests you do manually, but then I can run them again and again instantly."

As skeptics go, I'm a tough nut to crack, but I was beginning to feel that there was something bigger going on.

"Let's imagine I need to fix a bug in some code I wrote last week." Morris began picking up steam. "So I fix the bug, write a test to make sure it's fixed, and run it. But I also run all the other tests that I'd written up to that point. That's when I find out that I broke something else, so I go fix that too. And then I'm done. No more regressions."

That sure rang a bell. I remembered that Jay told me that that "impossible" case was called a regression, because the system "regressed" back to a state before the bug had been fixed.

"You're telling me this technique will make it so you only need to fix bugs once? If it can really do that, we could seriously cut down the time it takes us to release." I was trying to imagine what life could be like – no going home late at night all the time, no more coming in on weekends. We'd be able to get more features *done* in the same amount of time.

“Why do you call it unit testing then?”, I asked. “It makes it sound like the point of all this is for developers to test. Nobody’s going to like that. Developers won’t like it because they want to write code, not to test it. Testers won’t like it because it’ll be like the developers are taking their jobs. Why don’t you call it, I dunno, *regression prevention*, or something like that?”

Morris thought *unit testing* was a perfectly good name that everybody already knew, so why change it. We went our separate ways after that, but that idea stuck. A regression prevention technique that could be automated – everybody could like that; developers, testers, even management. Management would be willing to try anything to get rid of all the time spent on bug regressions. So, when I got back after the conference and had to present to everyone what I learned and what they should know, I talked about it.

Since the project I was on then wasn’t in trouble, the project manager gave it a green light, and the company’s first regression prevention program was put into place. Sure it didn’t go smooth, and we spent a lot of time maintaining the “test code” that we wrote, but you should’ve seen our releases. Since every bug found got an automated test, we didn’t have any more regressions. Well, that’s not entirely true. At first, we didn’t always run all the tests before beginning a release, but we learned.

To make a long story short, quality went up, time-to-completion went down, and all us developers got to go home when everybody else did. I even got a 2% bonus for improving productivity, and a plaque. I was on top of the world. (Give me a break, I was young.)