It’s a fine line between chatting and pair programming

By Pascal Van Cauwenberghe
pvc at nayima.be
http://www.nayima.be

Introduction

I was in a meeting with S from sales & marketing, when she remarked, “You guys in software development don’t seem to have a lot of work to do.”

“What makes you say that?”, I asked.

“Well, this whole morning I’ve seen you do nothing but chat with M.!”, she replied.

It was true. As per the Extreme Programming (XP) canon, we worked in an open workspace. S. could see everything we did.

Of course, talking wasn’t the only thing we did. We also drank a lot of coffee. But S. already knew that: we had to pass by her desk on the way to the espresso machine.

Okay, XP recommends a “sustainable pace”, but this was ridiculous! What do developers do all day, besides chat and drink coffee?

The day before: a customer calls

The day before one of our customers had called me. He had found some failed transactions in the server logs of the application we had built for them. This application ran 24 hours a day, 7 days a week, with a few thousands of users per day. Once in a while, amongst the tens of thousands of transactions, some (minor) transaction failed.

I examined the logs and the code, tried to find some pattern, some clue to what was causing these rare, random failures. I couldn’t find any pattern or reason. At the end of the day I was no nearer to finding a cause or solution for this problem.

That morning: call in the cavalry

This morning, during the stand up meeting, I asked M. if she could help me to find and fix the failure. I was getting nowhere on my own. I needed some help.

We looked at the logs and code; we brainstormed and discussed possible causes. When we thought of some possible cause, we wrote an acceptance test to check if we could reproduce the problem.

Finally, after a lot of discussion and coffee, we had an acceptance test, which clearly exposed the failure. Whew! The most difficult part was done.

A few hours later: wrapping up

Now that we had a failing acceptance test, we drilled down to the unit where the failure was caused. We added a unit test to expose this failure. Now, it was easy to fix the code to pass the new unit test. After that, the system passed the new acceptance test for the failure.

We fired up the automatic build, to completely rebuild the system, run all the unit tests and execute all acceptance tests. We went for another coffee. When we came back we received a report saying the system was built successfully and all the tests passed. Our change hadn't created any regression failures. We released the code and created a new installer package. We installed the new package on our local staging server and tested the application. Everything worked.

We then called the customer, sent them the new package, helped them to install it and monitored the server logs remotely to see if the failure would still appear. We saw no trace of the failure, but decided to monitor the servers for a week. The customer was happy his problem was solved so quickly.

And then it was midday, time for lunch.

The happy end?

The failure did not reappear. The system has been working flawlessly ever since. The customer is very happy that his system is working reliably again. So, all's well that ends well?

Well yes, except for a few things. Why had we introduced this failure? Why hadn't we found the failure before shipping the software? This was the second time in two years that this customer had reported a problem with the system. Clearly, something was wrong with our software development process if we let two failures be discovered by a customer.

Why didn't we catch the failures? In both cases, the problem manifested itself as a rare, random failure of some transaction among the many thousands of transactions. Our unit tests and acceptance tests hadn't caught these failures. Maybe we should introduce some sort of stress testing, to find those failures that manifest themselves only once in a while. The acceptance test for this failure exposed the problem by stressing the application with a lot of transactions. Let's note that for our next project retrospective.

Why did we introduce those failures? The first case was a bug in a vendor's library. We thought we had put a workaround in our code, but apparently the workaround failed in some rare circumstances. The problem was solved when we got a fix from the vendor.

I don't usually try to find who introduced a failure, blaming people doesn't solve problems. But, in this case I didn't have to look far: I had introduced the problem while implementing a request from this customer. I had written unit tests, of course, but I hadn't tested this rare case.

I had written this piece of code on my own. This code was so simple I didn't need any help in programming it. I only needed help to debug it.

Oh wait….

Conclusion

It's a fine line between chatting and pair programming, if you don't look too closely.

Except for this customer. For them it's very clear. They just look at the results.