Jorge Payán
23570

Assignment 0

Not just code monkeys - Martin Fowler

Martin Fowler talks about many topics that are somewhat linked, but I would like to write my thoughts on them separately. He begins talking about his frustration with agile development: “The development team ends up being a passive recipient of stories”, “The developers basically see themselves as an engine to turn stories into actual code”. I agree on the point that stories should not be decided only by the business but also by the developers, as we should be more involved in deciding what to do, not only in the aspect of assigning stories, but also in deciding whether what we are doing is correct or not, because as Martin Fowler says, when we decide to write code that could be maleficent we are as responsible about it as the person who had the idea and assigned it. I don’t go on the morally correct side often, but when it comes to writing core that is directly going to affect somebody, such as quiet or almost hidden extra charges when purchasing something or planned obsolescence, it is required to question ourselves if we really want to be involved into doing so. Fowler also speaks about acquiring more knowledge about the domain that we work in, he mentions how there are people that don’t even seem to be interested in learning more about it. When we work with something we should always have extended knowledge about the area, so we can understand precisely how everything works and if we are doing our tasks the right way, thus delivering a product of quality. Going back into morals and personal satisfaction, asking the question “Is what I am doing going to improve someone's life?” should be a question we ask ourselves when we are seen as a prospect for a new job, I have changed my mind about this subject quite a few times, but I think I finally matured enough to see what it really means, besides it is more interesting and brings satisfaction when we work on something that will change the way we live in a positive way. Talking about segregation based on sex or ethnicity, I disagree with Fowler on the point that our area pushes them away. Engineering is open to everybody, and everybody has an opportunity to be a part of it. What the problem is, in my opinion, is that not everyone is raised to evolve an interest in it. Boys play with legos, cars, computers and develop a more natural attraction to engineering while girls are thought to be interested in other areas. Answering Fowler’s question on how to change this: it has to do with earlier education more than with the area itself, we have to show everyone the different paths they can take, so they can at least have basic knowledge on what it is that we really do.

Workflows of Refactoring - Martin Fowler

Refactoring our code is an essential part of software’s life cycle when it comes to adding new components to our system. It helps us keep our code clean and maintain the quality and integrity of it. I like the way Fowler separates refactoring in many workflows, it is a way of understanding the different ways of refactoring. TDD refactoring, when we preserve the code’s functionality but improve its internal structure; litter-pickup refactoring when we change code to make it cleaner, fix something that doesn't look right, and I personally like this one because it goes against the idea of "If it works, don't touch it", which often leads to a problem that sits in the system and will eventually fail as the system grows; comprehension refactoring, when we don't understand a piece of code, when you move the understanding of the code out of your head and placing it in the actual code, so everyone else can understanding. I personally think this workflow of refactoring should be something required, code should always be readable almost to the point of not needing comments. Preparatory refactoring, when we refactor something that might have been correct when it was written, but now it should be different. Things change a lot in this area, and they do it pretty quick, which demands for this workflow of refactoring, not everything that is right right now will be right tomorrow, it might need new functionality to get along with other components or to fit in with the bests practices of today. Planned refactoring, when we clean up our code as something planned in the beginning of the project. I share the idea of the code being cleaned should always be part of a project, but also as Fowler says, if you are doing it, it means that you are not really doing other kinds of refactor. Long term refactoring sounds to me like an unnecessary practice, I don't think code should be refactor in small steps every time you go back to a component; I know that gradual progress tends to be more efficient than other activities, but in this case I think components or functionalities should be worked on completely and left clean, and with this I don't mean to work on it only once. Again, refactor is an essential activity of software development. A problem left sitting in a system will lead to a problem that will be very hard to track in the future, we should always follow code standards that will allow everyone to work in the most comfortable way, because when we follow the best practices and make our code clean and readable, the performance of a team will boost to a point where we will be delivering our tasks at a faster rate.

Microservices - Martin Fowler
Monolith vs Microservice. An application that has many functionalities vs small processes, each providing a service. Whenever something new comes out, and people start paying attention to it, we tend to find every good reason to use it, but is it really all superior than the previous option? I don't really stand on either of the sides, with monolith and microservices, I stay neutral. There are just as many advantages as disadvantages when comparing these two. Microservices apports a more modular solution to a problem, easier to update for it can be updated individually while in a monolith system, we have to update everything at a time; it also adds in partial deployment because each service has about only one responsibility, but again, you can also deliver slices of a monolith system at a time; microservices can adapt easier to different situations, you don't have to replicate all your work in many computers, you can simply replicate the same service if it requires to take more load, but a microservice is not that consistent, they tend to fail so often that Fowler describes them as "Designed to fail". A Monolith application is more simple, all the internal components already communicate with each other in an easier way, where in microservices, you have to be careful when designing the interfaces. In these situations, there is not a preferred solution nor a solution that will replace the other one completely, it really comes to decide which one fits in better for your problem.

REST: I don't Think it Means What You Think it Does - Stefan Tilkov
I did not really have a formal introduction to REST until now, and that is mostly my fault for not reading more about what it is, rather than how to make a RESTful API, even though when I worked with it in the past. The theme of this video is the misconceptions of REST, which I sadly felt related with. I thought most part of REST was defining URIs that go along with the POST, GET, PUT, DELETE methods, what he calls "pretty URIs". I think Tilkov spends more time discussing what a REST API is not rather than what it is, but then again the title of the presentation infers that it would be this way. Overall, I think the parts of the video where he gets into explaining what it is, like when he talks about service interfaces, resource links, states, etc, are pretty well explained, even though most of the video is a bit redundant and not so practical, it's a more abstract explanation. Again, I don't think I have enough bases to criticise, either agreeing or disagreeing, the conference.