Homework 01

Lukas Blass - blassl

April 3, 2019

1 Collaboration

In the software development process I have always seen problems arise when multiple contributors work together on a project or a module thereof. I have experienced difficulties in this regard especially during projects at university, where one tends to start a project from scratch. I especially see two critical points that make this problem hard. The first consisting of specifying and distributing tasks among the contributors and the second being assumptions and definitions that change throughout the process. Let me further explain in the following paragraphs:

Before each member of the team can start developing, (at least) a subset of the modules has to be specified and tasks distributed among the developers. Coming up with the correct or ideal abstractions, encapsulations and APIs out of the blue is almost impossible. Nevertheless the process has to be started at some point and one proceeds to specify them to the best of ones knowledge. This process takes time, especially due to dependencies between different modules and the resulting need for the definition of interfaces. During the development process, the previously made assumptions then prove to be false, which can be both frustrating and time consuming. As APIs are adapted, this needs to be communicated to everyone using it. Often however, this communication fails to be optimal resulting in further frustration and loss of time due to now wrongly made assumptions by other contributors.

Of course these problems appear in an extreme form in the initial phase of a project. As projects become larger and modules well established, sudden changes won't be necessary (or possible?). This might also be a reason why more time is not dedicated to solving this problem: most developers are going to work on an existing large repository rather than starting a project. Furthermore, there is most likely not a real solution to the problem, since it is for example impossible to foresee the changes that will have to be made for a specific module.

2 When to refactor

Contrary to the previously described issue this problem is solely to be traced back to myself. During the development process I am over long stretches con-

cerned with achieving the desired functionalities and properties of my program in the sense of correctness. During these stretches I then tend to not focus strongly on design patterns, paradigms or cleanness, which can temporarily lead to confusing or unclean code.

Consequently, the question comes up how often the code should be refactored to match the aspired paradigms. Every time the code is uploaded to the repository? Only after the functionality has been achieved? Or try the best not to write messy software in the first place? Most likely there is no definite answer and actions should be taken depending on the specific case. Depending on who else is contributing to the software, how many components are manipulated simultaneously etc., different approaches might have to be taken. I for one regularly find myself cleaning up my own code asking myself why I didn't manage to write it down properly in the first place.

3 Design by Contract

I have been exposed to a very specific way of testing and verifying software at the start of my programming career. My first lecture on software development at university was held in the programming language Eiffel, an object-oriented programming language that's likely best known for its use of Design by Contract. Using Eiffel I have been taught that one should write pre- and postconditions (to check assumptions on function arguments and results) and class invariants (to test properties of objects). This methodology allows function inputs and outputs to be tested and certain assumptions to be asserted at all times. In other words, writing conditions right alongside the code is a way of testing the software.
However, this way of testing software, at least to my knowledge, has never really been adopted by programmers to the full extent. And that even though the programmer has the choice of excluding coded contracts at runtime, i.e. having the possibility to omit the tests e.g. at deployment.
I see a cause for the lacking success of this testing methodology in the fact that it mixes together the tests with the actual software. This downfall comes in two ways: firstly, the programmer is forced to think of conditions and contracts during the development process. In theory, this is a great argument for the Design by Contract methodology, since it makes the programmer more alert to unexpected behavior and edge cases. In practice however, programmers want to focus on producing code first and only later focus on testing. Secondly, large numbers of conditions in functions and class definitions reduce the cleanness and readability of the code. To implement meaningful assertions multiple lines of code have to be dedicated repeatedly, leading to an increase of total number of lines and a general reduction of conciseness. I find it highly interesting that the methodology is not applied more often, especially since there are great objective reasons to do so. It seems however that the subjective discomfort for the programmer prevails, which is why, I would speculate, not more developers write tests directly in their functions and classes.