Project Reflections - II1305

June 9, 2021

Philip Salqvist

phisal@kth.se
CINTE
1992-04-17

Contents

1 Introduction

This report is an elaboration of my individual project reflection from the course II1305. In the project reflection, a my focus was primarily on our job to write consistent, readable and maintainable code together as a group. The project group consisted of 8 developers, and many of us lacked experience to write code in a group of that large size. Furthermore, we will have to be a part of much larger teams in later parts of our education, and especially after graduation in the work place. During the project, I noticed that the size of the group made the internal communication among team members difficult, especially when it came to formatting the code. We lacked consistency, we duplicated each others work, and we didn't take the time to make the code readable and understandable for our future selves and the other group members. This created some particular problems. First, the inconsistency made it difficult to quickly scan a piece of code and to make sense of it, since you didn't have any sense of how the code was organized in the file when jumping between files written by different developers. Second, the duplication is a problem in that it makes us spend time writing code that has already been written. So it adds additional work that could have been avoided. Furthermore, it adds to the inconsistency, since one reusable function or component can only be designed in one way. But several functions or components that does the same thing, can be designed in several different ways. Finally, when it comes to the readability, the fact that we couldn't quickly understand what a function or component did, that was written by another developer, made us spend a lot of time understanding code, that could have been quickly explained in a comment. So, in this report I will walk through these three mistakes that we made in our project and make suggestions on how to avoid them. This is written partly, for myself to learn from these mistakes, but also for other students to not make the same mistakes when taking the II1305 course. To avoid my own biases in how code should be written properly, I will use the book Clean Code by Robert C. Martin as a reference.

2 Inconsistency

When talking about inconsistency among developers in a group, I think the focus has to be narrowed down in able to be some what specific. Because of this, I will merely talk about writing functions in this section. I will first and foremost discuss some ground rules to keep in mind when writing functions, and if there rules are discussed as a group before a given project, the code will in turn become more consistent as a consequence.

First of all, our group were very inconsistent in how large our functions were in terms of number of rows. Some members refactored their code a lot to keep functions smaller, while others wrote longer functions. When functions get to long, it could often be the case that it starts to do more things that was initially intended. Martin writes that "The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that." [1]. Martin goes on to say that functions shouldn't be 100 lines long, and that they hardly should be 20 lines [1]. Where the line is to be drawn on how long a function should be in terms of lines is debatable, but the fact that functions become less readable the longer they are is, at least according to me, not. So, if the group would discuss how long they think a function should be before considering to refactor it could at least gain a common understanding of function length among the group members and thereby increase consistency.

After Martin establishes the utility of writing small functions, he continues to discuss the fact that a function should only do one thing: "Functions should do one thing. They should do it well. They should do it only" [1]. This rule has a few utilities. First of all, it naturally makes the developer to shrink the size of a function. Also, I would say that forcing yourself to write functions that only does one thing would enable you to name it properly and clearly. This way, we make the code more readable to yourself and other developers in the future and thus makes the code more easy to understand. In our project the functions could sometimes become very long and that would

often be caused by the fact that the developer tried to create a lot of different functionalities within one single function. Instead of doing this, developers should keep in mind that if the function is doing multiple things, break it down into several smaller functions instead.

Finally, Martin discusses the fact that well written code should enable the reader to read it as a top down narrative. [1] He states that we should be able to read the code from top to bottom, and that the functions should descend in order of abstraction. [1] This means that we should move from a low level of detail to a high level of detail as we read from top to bottom. Making a rule like this explicit to the developers in a team, could increase the consistency substantially.

3 Duplication

Martin states in his book, Clean Code, that - "Duplication is the primary enemy of a well-designed system. It represents additional work, additional risk, and additional unnecessary complexity".[1] When conducting this project, the group noticed after about half way into it that a lot of duplication had been produced. There were two main reasons that this happened. For one, there was a lack of detailed discussion and planning initially. And second, there was a lack of communication about this fact during the project.

Let me explain by an example. The aim of the project was to produce an app. The app consisted of a number of pages, containing lists, where every item in the list was constructed of a specific list object that rendered some kind of data. The group divided itself into teams and created a page each. This resulted in a different list object for each of the different teams, that lacked consistency in design, both in terms of aesthetics and code. When this was discovered however, the group had to spend a lot of time to write reusable components, and pass arguments to these components to specify the content that they where supposed to hold. We then had one list item that could be reused and was consistent in it's design. And this is only one example. If

the team had discussed the components that could be reused, we would have saved a lot of time. There is no need to spend time reinventing the wheel over and over again. This is something that we all knew as programmers, but it was difficult to manage with a large amount of developers in one project.

We also have to consider the fact of maintainability. Duplicating code makes the program difficult to maintain. Martin raises this issue by explaining that if an algorithm is created four times in four different places, a change in the algorithm requires a four-fold modification [1]. It doesn't sound that bad, but at scale this can become problematic and it increases the risk of bugs in the code.

4 Readability

Another problem in our project, was that we often didn't pay enough attention to the readability of the code. Due to the time pressure we had put on our selves we didn't take the time to create high quality code, rather we focused on getting the code to work, and then often moved on to the next issue at hand. The problem with this was naturally that other people had to spend a lot of time understanding your code, and if the code isn't easy to understand future maintenance becomes slow and difficult, which increases the probability of making mistakes. A rule of thumb in scrum is in fact to not sacrifice quality for quantity, and unfortunately we did that in a lot of cases. So what can we do to make the code more readable? Martin discusses this at length [1]. In this report we will mainly discuss formatting, commenting and naming of variables.

One problem that we encountered was that our files became very vertically long. This was a problem, since it was difficult to get a quick overview of the code and what it was doing. Martin discusses this problem and concludes that a good aim is to create files that are between 200 and 500 lines long [1]. Of course this isn't always attainable, but discussing the upper limit of file length with team members and agree on a desirable aim could be ben-

eficial. The author continues to discuss something he calls The Newspaper Metaphor. A source file is compared to a newspaper in the case that a story in a newspaper starts with a headline that gives the reader a sense of what the story is about. The analogy here would be the name of the source file. The name should be sufficient to let the developer know if this is the module you are looking for. After the headline, usually their is some sort of high level summary of what the story is about. After this, the author often continues to go into a more detailed description. From a developers perspective, it would make sense to begin the file with high level concepts and algorithms that explain the functionality of the file, and to go into greater detail further down in the source file. [1] Martin continues to discuss Vertical Openness, Vertical Density and Vertical Distance. What he means by this, is that concepts that are separated should also be visually separated with a blank line. Concepts that are not separated should be associated with one another. And concepts that are related to one another should be vertically close to each other. Finally, Martin discusses variable declarations. Martin states, that variables should be declared as close to their usage as possible. Except for instance variables, that should be declared at the top of the class.

Moreover, my opinion is that the group didn't provide useful descriptive comments enough during the project. It is my opinion that a comment quite quickly can give you an overview of what the particular piece of code is doing, and if it is relevant to you or not. Although this was my initial assumption, Martin doesn't completely agree with this view [1]. The author rather writes that if you have to provide a comment, your initial reaction should be that you haven't made the code itself expressive enough. So you might wander why the author is so reluctant to use comments. Well, code usually change quite a bit during a project, as well as after a project is performed during maintenance for example. Comments then often have a habit of not being maintained, so the result is often that they are deceiving the developer rather than providing explanatory guidance. Comments aren't inherently bad, but if they could be avoided by creating code that is self explanatory in itself, that would be the desired option. [1] So, a way to make the code more self ex-

planatory is to provide meaningful names to variables, functions and classes. Let me provide an example from our project. Examine the code below:

```
const createTrnmnt = () => {
        // Validate rank interval
        if (toggle1 && toggle2 && rank1 <= rank2) {
            setErrorMsg('Invalid rank interval');
            return;
        };
```

Instead of providing a describing comment, we could create a function *validRankInterval*:

```
function validRankInterval(rank1, rank2, toggle1, toggle2) {
    return rank1 <= rank2 && toggle1 && toggle2;
};

const createTrnmnt = () => {
        if (!validRankInterval(rank1, rank2, toggle1, toggle2)) {
            setErrorMsg('Invalid rank interval');
            return;
        };
```

Martin explains the importance of well written names - "The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent." [1]. By using meaningful names, we can make the code more readable and thereby increase developers ability to effectively understand our code and avoiding unnecessary comments that won't be maintained properly in the future.

References

[1] Robert C. Martin *Clean Code*. Pearson, 2008.