Craftsmanship of Software Development

v. 1.0, 24-05-2019

(C) Przemysław Kruglej
2019

przemyslawkruglej.com
craftsmanshipof.software

przemyslaw.kruglej@gmail.com

Table of Contents

Craftsmanship of Software Development

As I've progressed in my software developer career, I've learned, often the hard way, that **doing things in certain ways was more productive, increased quality, and facilitated cooperation with my teammates**.

At some point I've started thinking about programming as a *craft*. Extremely hard to master, evolving quickly, and very demanding. Let me show you the definition of a *craftsman* from the online Cambridge English Dictionary:

Craftsman – a person who is skilled, especially someone who makes things by hand

https://dictionary.cambridge.org/us/dictionary/english/craftsman#dataset-cbed

I really like the sound of that. Aren't software developers the craftsmen of the 21st century?

This document is a conclusion of my findings related to working as a software developer. I believe that the ideas presented in this document will allow a developer to progress towards becoming a master craftsman in the land of software development. All topics presented here are technology-agnostic, meaning that it doesn't matter which programming language or framework is your domain.

I've divided the document into *Elements*, grouped by topics they relate to. I really liked the idea of *"Items"* in Joshua Bloch's astounding *Effective Java* book. The author clearly presented a way-to-go for various Java programming paradigms and problems in his *Items*. I came up with the notion of *Elements*, which are, in my opinion, a foundation for becoming a master craftsman. I myself am nowhere near to becoming one, yet, but I feel I'm on the right course.

All ideas presented here are just my thoughts. I'm basing them on experience gained on many projects I have been working on over the years – maybe you will agree with me on some of them, and disagree on other. I'd like to know what you think about them, and it would be great if you would share your thoughts – I'll happily receive any e-mails: przemyslaw.kruglej@gmail.com.

Elements of Quality

This group of elements relates to the quality of software developer's work.

Element 1 – Have Your Code Reviewed

How many times have you seen code that wasn't complying with your project's style guide (*Element 16 – Comply with Project's Code Style Guide*), didn't conform with the best practices, had bugs in plain sight, and just wasn't well written? All of those could have been spotted and fixed during a code review. So, why weren't they? The problem here is threefold.

Firstly, people fear criticism, and that's what often code review ends up being. On the other hand, well-done code review (*Element 14 – Put Effort in Code Reviews*) facilitates better code quality and lowers number of bugs, and in my opinion quality should be our driving force (*Element 2 – Make Quality Your Driving Force*).

If this is the case for you, don't be afraid of code reviews. Find people on your project that you know will provide constructive feedback and that will make effort to help you improve your code.

Secondly, some developers tend to overlook one simple truth: everyone makes mistakes (which every introduced bug confirms). Be humble – no matter how much experience you have, your code is not error-proof. We all make mistakes, and there is no shame in having somebody point out a problematic piece of our code **during a code review**.

Give others a chance to spot a bug in your code *before* it goes to production.

Lastly, some developers are just not used to code reviews, and often don't realize how much good it can bring them to have someone take a look at their code. Keep in mind that each code reviewer always has a different perspective, which includes different skills, experience, and project knowledge. They will see things that might be improved and/or that are vital to the project you are working on.

Make it a habit of having your code reviewed. It does pay off, and increases quality of the code you create.

To summarize, code review is a great tool that can help you increase quality of your code and minimize number of bugs in your code. You just need to give your teammates a chance to do it.

Element 2 – Make Quality Your Driving Force

When you start working on a task, what do you picture as the end result? Is it just a trio of: a commit, deployment, and time spent logged in an issue tracker?

Let me propose a different approach, where you care about the value of the delivered piece of software from the very beginning, and all you do, you do with one thing in mind.

Quality.

This might sound simply, but it takes time to get into the habit of this approach, which I call *Quality-Driven Development.*

Whenever you start working on a task, keep in mind that your approach is quality-driven. Your goal is to end up with a piece of software that you will be proud of. **Remember that your name will be next to the commits which introduce changes in the project.** And that alone, in my opinion, is a very powerful notion, worth stopping for a moment and thinking about. Consider that from the moment when you push your changes, everyone will be able to see them, and though you can't judge a book by its cover, the same cannot be said about source code and its author.

Therefore, **make quality your driving force.** Whenever you deliver something, ask yourself these simple questions:

- *"Is this the best I could have done?"* – Can you say that this piece of code couldn't have been done better, taking your skills and knowledge into consideration?
- *"Have I left nothing behind?"* – Are all of the cases, that you have thought of, taken into account?
- *"Could I have asked somebody to help me improve my code?"* – If you know there is somebody on your project that could help you improve your code, have you asked for their expertise?

Many of the other Elements in this document emphasise directly, or indirectly, *Quality-Driven Development*, and propose ways to support it, like *Element 3 – Care About Small Things* or *Element 9 – Think About Developers Who Will Maintain Your Code*.

Element 3 – Care About Small Things

Sometime you encounter a task (perhaps even yours) and you notice that it could have been done better.

- Maybe the author hasn't paid enough attention to the code formatting, or the code violates some of the best-practices used on the project, or the code has been written in a way that is hard to understand.
- It might be that the tests haven't been written or updated, or no attention was paid to what happened to the task after it has been merged (and something went wrong), or the commit message isn't descriptive.
- Perhaps some of the business cases were neglected by the developer, because he either thought they were close-to-nonexistent, or because they were not included in the business specification, and the developer didn't think about mentioning them to the business users.
- Maybe the related documentation hasn't been updated after the task had been done, or information that would help testers perform the test of the task hasn't been supplied.

The points listed above (and other, as the list could go on and on) would be regarded differently by each developer. Some would be negligible, whereas other would be requirements in the *definition of done*. Either way, they can all be viewed as details. And details make all the difference. **It is the small things that bring a developer's work to the next level.**

Don't leave anything behind. Whenever you feel like the task is ready, consider if there is anything that you can improve, or if there is any related action that you can do.

There is another important thing to mention here, and a separate *Element* is dedicated to it. People notice detailed work and quality code, and, therefore, its author, and it is a good feeling being regarded highly by your colleagues (*Element 4 – Be Reliable*).

So pay attention to the details. They all add up to the quality of what you deliver.

One thing, however, that working on the details requires, is time. It is often a trade off between how much time you have and how strongly you feel about delivering top quality. Keep in mind that making your tasks *pixel-perfect* will often take significantly more time. It might not always be the right solution and you will have to adapt (adaptation is the topic of the next set of Elements). Try finding a middle ground when needed.

Element 4 – Be Reliable

When you join a new project, you often find out that there are some people that are said to *know everything* related to the project and/or are very strong from the technical point of view, and you can always ask them for an advice, learn from them, and expect their work to be of finest quality.

You can be such a person. All it takes is the *Quality-Driven Development* approach, attention to details, technical expertise, and being helpful... Sounds like a lot, and it is. **You have to work for your reputation.** So, if you will care about your work, make quality your goal, and if you can act as a team player, you will become the 'guy (or girl) who can be counted upon'. It is a nice feeling to be respected and viewed this way, but it takes time, effort, and work.

Become a person that others can count on. Build your reputation, and you will be rewarded for it.

Also of importance, as mentioned above, is that not only quality of what you deliver matters here, but many aspects of teamwork as well, which is a topic covered in one of the following sets of Elements.

Elements of Versatility

Versatility for software developers means adapting quickly and developing a range of skills to help them keep up with changes, as well as being able to put themselves in the bigger picture.

Element 5 – Search for Answers

Often when you ask a teammate a question about a problem you are facing, they will just immediately type it into a search engine in their web browser. They don't know the answer, but they know where and how to look for it. A few seconds later they present to you a solution to your problem, taken from one of the first search results. You ask yourself: *why didn't I do that in the first place?*

Most of the questions have already been asked, and you will quickly find them on the Internet, along with proposed solutions. If this is what you do already, that's great, if not – give it a try. You will almost always find the solution in the very first few search results. It's just a matter of wording the query for the search engine in the right way, and it is something that takes practice, so don't be discouraged if the first few tries do not yield any relevant results.

You will often find what you are looking for on Stackoverflow, which has gained a lot of popularity in the last years. If you find an answer on Stackoverflow, consider upvoting it to show its author that he has helped yet another fellow developer.

At times, finding answers in the vast Internet might be hard, and it might require time and commitment. Keep in mind that searching for answers is a **skill to learn**, and a very important one. It is also rewarding – it will quicken your development process and make you less dependable on getting help from other developers. What is more, it saves time – both yours and your teammate's, whom you would ask the question. Of course, it doesn't mean that you shouldn't ask for help when you are stuck with a problem – *Element 10 – Value Time of Other People by Asking the Question the Right Way* proposes a way for you to *help other people help you*.

Element 6 – Acknowledge Testing as a Part of Your Job

Testing is vital in our line of work, and often there are dedicated testers on projects. They take care of testing once we assign a task to them that is, in our opinion, ready to go.

From time to time, though, tester from our team might be on holidays, get sick, or just have too many tasks on her or his plate to be able to finish all of the testing in time before the system goes live. It might also be a case that there just aren't any testers in your team.

Whatever the reason, you might be asked to help in testing. I often hear developers answer that *testing is not part of their job*. They sound like they feel offended by the notion of having to do some testing, like it is somehow going to diminish them, whereas, in fact, **testing is a part of every developer's daily work**. Whenever we write some code and produce a functionality, we should test our solution before it goes further down the workflow. I myself can't imagine claiming the task is ready, having not tested it myself, and asking tester from my team to take care of it.

Testing is rooted in the very concept of software development. If there is a chance to support the team by testing a couple of tasks in a time of need, why not do it? A little versatility goes a long way. Don't think about testing like something worse than writing the code, where testing is in fact a part of the development process.

Help your team by doing some testing in a time of need. Nothing wrong about it, and having to test some of the other developer's work might just reveal to you how hard testing can get. You can easily make it simpler for the testers if you provide information about the task you have done, covered by *Element 15 – Document Your Work and Provide Helping Information for Testers*.

Element 7 – Look at the Bigger Picture

Imagine: a developer comes to work and there he is, with his tasks and problems to solve. He writes code all day and happily goes back home. That picture, for many of the developers I have worked with so far, would be a *dream job*. Unfortunately for them, it is not how the developer's work looks like most of the time.

There are meetings, supporting colleagues, code reviews, work time tracking, more meetings, tedious bug fixing, tasks time-requirement evaluation, business' needs analysis, and so on. Our line of work consists of much more than just writing the code, and we need to be aware of that. The workflow of delivering value is complicated, and varies from project to project, and team to team. What is important here is **understanding that there is much more to software development than... software development.** Next few times when you'll be doing some task that is not related to writing source code, think how your work contributes to the overall process of software creation in the company that you are a part of.

Two complaints that I hear from teammates very often are: logging work time and having to estimate time-cost of tasks. It can be tedious, but it is often necessary from your employer's point of view. He is, after all, paying you for what you do, and most probably there's a budget to consider, as well as requirements from stakeholders. Thus, knowing what the team has been working on and how much time it took is vital to the company. Same goes for the task estimation – knowing the cost of implementation of different business needs allows a better understanding of what can be delivered and when. Basing on that, your employer can choose, with the time-cost in mind, what is the most important and thus what should be the point of focus for the team.

Whenever you join a team, become a member that contributes to the team's effort of delivering value. Take a look at the bigger picture, and understand the project's and team's workflow. Keep in mind that your work consists of many different tasks. Writing source code is an important one on that list, though not the only one.

Element 8 – Embrace the Challenge of Other Technologies

Our world of software development evolves quickly, and many new trends, languages, frameworks, and technologies pop up every year. Each of us specializes in what she/he does and/or likes best, and we tend to stick to those same technologies for quite a long time. Sometimes we might be asked to do a task not related to our specialization. What should we do? Decline or happily try out something new?

It depends on what we are asked to do, which technologies are involved, and how much time it will take, and, possibly, how often similar tasks will pop up in the future. We have to evaluate all of those and make a choice. However, consider that learning other technologies might be beneficial, and developers who are able to work with a broader range of languages and frameworks are valuable to employers.

I have worked with developers who were very unhappy when they were asked to do something that was just outside their comfort zone. No one was asking them to totally abandon their language of choice, yet still they complained about doing something new to them. Instead of looking at it as a challenge and possibility to learn something new, they regarded it as a waste of time and something that was out of scope of their responsibilities.

Just to be clear, **by no means am I talking about leaving your favorite language and framework**, and starting working with other technologies that you have no interest in. What I mean is to be versatile enough to deal with tasks that might not be covered by your specialization, from time to time. If there are too many of them, then it is a good idea to talk to your manager about it.

When asked to do some programming in a different language, or using a different technology or framework, try to look at it as both: a challenge and a possibility to learn something new. You might like it, and if not, it might be an addition on your curriculum vitae. Getting basics in other technologies might be a huge advantage on the market.

Elements of Teamwork

Elements in this chapter are about facilitating teamwork.

Element 9 – Think About Developers Who Will Maintain Your Code

Whenever you finish working on a piece of code and merge it to the main branch, it will be available for all other developers to see as long as the project will be maintained. **Your name will be next to your commits** and you can be sure that, sooner or later, somebody will be looking at your code. It would be beneficial if the code was clean and good as it speaks about your attitude, technical knowledge, and care for quality, as well as shows that you think about anyone who will be taking care of it in the future.

Because of bugs or requirements changes, someone will have to read your code, try to understand it, and change or fix it. Probably it will be you. How many times have you tried to understand other developer's code, and wished they thought about any future person who would have to maintain it? How many times the author of the enigmatic code was you?

Taking time and caring to write a piece of code you will be proud of will save you a lot of time in the future, and, what is probably more important, even more time of your fellow teammates.

It is a good habit of walking through your code once you are finished with it and trying to consider the following:

- Will I understand my code tomorrow, next week, in six months?
- Will the code be understandable by someone else?
- What questions might be asked about my code and what could I change to make the code give those answers by itself?

Remember – your code will be read and maintained by other people (and, probably, you too). Taking a few extra minutes to think about its clarity and understandability can save hours of future time of anybody who will be maintaining it. What is more, remember that your code speaks about yourself. Strive for quality and put effort to deliver a piece of source code that you will be proud to have your name next to.

Element 10 – Value Time of Other People by Asking the Question the Right Way

We, as software developers, encounter lots of interesting problems in our daily work. From time to time we get stuck with a problem, and we look for answers on the Internet, since most of the questions have already been asked (*Element 5 – Search for Answers*). However, from time to time we seek help among our teammates.

The questions I often hear are quick, without providing any context, and asked in an unstructured way, leaving the person being asked in a spot that makes it hard for them to provide help. This makes both people lose time, as the person who is being asked has to ask more questions to actually understand the problem.

Whenever you want to ask a question, it is a good thing to consider whether you would understand your own question and be able to help if someone else asked it, taking into account that you only got as much information as you decided to provide in the question.

Thinking upfront about what you want to ask saves time, because you put your problem in a structure of a question in such a way that will be understood by someone else. Problem understood is a problem more likely to be solved quickly.

What is more, if you prepare your question, and the first person that you ask will not know the answer, you will be able to ask another person the same, already prepared, question.

Lastly, there is the *rubber duck debugging (https://en.wikipedia.org/wiki/Rubber_duck_debugging)*. If you have not heard about it, it is a way of trying to tackle a problem by talking to a "rubber duck" about it. You are supposed to tell the "duck" what your problem is and describe it in a simple way. It turns out that this way, when you are thinking about the problem in terms of explaining it to somebody else, our brains work in a different way, which very often leads to finding a solution by ourselves. It actually works pretty often, so preparing a question has a benefit of a chance of granting you the solution even before you go and ask the question in the first place.

To sum up, prepare your question before asking. Provide all the necessary information in a concise way, structuring it, for example, in the following manner:

- Firstly, give a short and precise description of the problem in two to three sentences. Also mention what correct result you are expecting.
- Secondly, if you have tried using a piece of code, enclose it, as it might be useful. If you have tried a few possible solutions and none of them worked, that is also an information worth sharing.
- Next, if there were any errors, attach their codes and description – they always shed light on the problem.
- Lastly, depending on the type of the problem, you may also give an instruction how to reproduce the error you have encountered.

This is of course only a general structure that can serve as a starting point. The point here is to think about what you want ask, and how you can ask the question in such a way that will **help another person help *you***.

Element 11 – Be Helpful

As you gain technical experience and new skills, and expand your knowledge about the project you are working on, you will probably be a person that will be often asked for help or advice. You might prefer to just focus on your programming tasks, but remember: *a little kindness goes a long way.*

Younger developers will look up to you and ask for your help, and other teammates will ask for your advice when stuck with a problem. Take such opportunities to teach, share your knowledge, and give advice to other people on the project. Teaching and knowledge-sharing is a good thing for the overall quality of the project you are working.

Even when working with sombody with whom you are not on the same page most of the time, it is still better to have a professional relationship. Life on the project will just be easier.

To be clear, **I do not mean to be always available to everyone, answer all questions and forfeit your daily tasks**. What I mean, as in *Element 4 – Be Reliable*, is to be a person that is known to be helpful and passionate about passing their knowledge, and putting effort in supporting teammates.

Be helpful. Help your teammates, share your technical knowledge and project experience. By helping others, you help the team deliver better software.

Element 12 – Don't Blame – Solve Problems

An inherent part of our job is making mistakes. Bugs happen, mistakes are being made, stuff is being overlooked. No one is infallible, no matter how accurate or experienced. When something bad happens, the key point, in my opinion, is not to seek a person to be blamed for their mistake. There is just no point in putting the blame on one of your teammates, and even less in attacking them publicly. A better option is to solve the problem at hand. Talk to your teammate and try to find a solution.

From my experience, when there is a need to communicate to somebody that their mistake has caused some trouble, the best approach is to contact that person through a communicator that the team is using. You can calmly talk about the problem that you have found and discuss a way to handle it. People tend to get defensive when their mistake is pointet out, and this way it is easier for them to digest.

This approach has an additional advantage. Again, talking from experience, sometimes it happens that you mistakenly think that a certain person have made an error. When this happens, and you want to talk about it (and I've been on both sides), it is best not to accuse them of anything in front of other teammates. It leaves a bad taste (no matter if it was their fault or not), and doesn't make it easier working on the project. If it wasn't their fault in the end, you can still talk it through – maybe you'll be pointed in the right direction to solve the problem.

The message you could write, using your team's favorite communicator, could be something along these lines:

> "Hey, I see in your commits from last Friday that you made some changes
> in files X and Y. I think it might have caused some trouble. Following errors were found
> in production logs: <...>. Could you take a look at it?"

If the person isn't willing to cooperate to fix the problem (as they should per *Element 13 – If Things Go Wrong, Admit It*), then, at least, you have tried. Create a bug task in your team's issue tracker and inform your manager/team leader/product owner about it. Better yet, depending on the magnitude of the problem, if you have time and you know how, just try to fix it.

When a problem occurs, instead of blaming – look for solutions. Communicate the problem in a discrete way, and lend a helping hand, if possible. Helping, not blaming.

Element 13 – If Things Go Wrong, Admit It

As mentioned in the previous Element, mistakes happen. From time to time somebody will point it out to you that you have broke something. There are two very different ways to deal with this situation.

You may either go in the direction of denial, explaining yourself, blaming another teammate, or, worst of all, telling the person who approached you to just fix it by themselves, wanting to have nothing to do with it. This is not the way to go.

The way to go is to **take responsibility for your actions**. Something went wrong and it was possibly my fault? **Alright, my bad, thanks for letting me know.** Let me check it out and fix it. After all, we want to deliver quality (*Element 2 – Make Quality Your Driving Force*).

How your teammate will decide to communicate to you that you broke something is up to him or her. How you decide to react is entirely up to you. Whether or not their approach is similar to description from *Element 12 – Don't Blame – Solve Problems*, stay calm. The best thing you can do is assess the situation and consider if the problem was really caused by something you did. Discuss it with them (if they are also trying to find a solution instead of somebody to blame), ask where you can find the logs with errors (where applicable), go through your commits to analyze possible error in your code, etc. Act upon it – find the root of the problem and apply a solution. Don't deny it and don't tell your teammate that he should just fix it by himself. Discussion is always better than arguing.

Somtimes it will turn out that it wasn't your fault after all. What to do then? Just contact the person who found the problem and explain it to him/her. Discuss what to do next. You or the other person will probably want to talk to the person who might have introduced the bug. If that's not possible, or you can just fix it easily, then for all its worth you can just fix the problem yourself, or create a new task in your project's issue tracker.

If you break something and somebody points it out to you, do not deny it. Start thinking about how to solve the problem and act upon it. It might also happen that you will be the person who finds out that something you were working on went sideways. If this is the case, try to fix it, and if necessary, notify people who might need to know about the problem, like release manager or a product owner.

Element 14 – Put Effort in Code Reviews

Not by a coincidence the first Element of this document is *Element 1 – Have Your Code Reviewed* – it is to emphasise how important the code reviews are in my opinion.

No matter how small a change, it is always good to have somebody take a look at our code. If we can decrease the chance of introducing a bug and improve quality of our code by asking somebody to review it, then we should let them.

If you are the person doing the code review, then try to make the most of it. There are three good reasons to do code reviews:

- **Reducing the number of bugs** – when you look at somebody else's source code, you have your own perspective, experience, project knowledge, and you aren't biased towards thinking that the code works. A lot of bugs can be prevented from reaching production thanks to that.
- **Improving code** – you might see things that should/could be improved. Sometimes bigger and more important, sometimes smaller, but important as well (*Element 3 – Care About Small Things*). All fixes done during code review contribute to a better code quality. Just remember to be thorough and don't leave anything behind.
- **Passing knowledge and teaching** – code reviews facilitate flow of both technical and project knowledge. You can really help out your teammates by suggesting how they can improve their code. Code reviews are also a good way to pass project knowledge about important matters that new teammates might not take into account.

Keep in mind that how you do a code review is also very important. **Code review should be more like a conversation and tutoring rather than trying to show your teammate how badly he or she wrote his/her code.** Code review where you try to prove that the code is bad and you do not provide any constructive feedback will certainly make your teammate think twice before asking for a code review next time – and not just you, but anybody. And this is something that you really do not want. You should want your teammates to never skip code reviews – they are too important to not do them.

Just one more thing that is important – remember that your teammates do not always have to agree with your point of view and you have to take that into account. Provide your feedback, present your point of view and discuss with them potential changes, and consider their arguments.

To sum up, if somebody asks for a code review, either put effort in the code review and try to make the most of it, or tell them that you do not have time to do it. There is no middle ground. Thorough code reviews can really reduce the number of bugs and improve code quality. You can also greatly help teammates who are new on the project by pointing out important matters related to the project you are working on. Just rememeber that your teammates don't always have to agree with you!

Element 15 – Document Your Work and Provide Helping Information for Testers

Software developers often consider a task to be done when they are happy with the code they have written and pushed to the main branch. Often the task is then assigned to be tested by one of the team's testers. If you have ever done any testing of functionalities introduced by other developers (*Element 6 – Acknowledge Testing as a Part of Your Job*), then you may know just how hard testing can be. Especially when you are supposed to test a task with either one line of cryptic description or with highly technical/business-related description, neither of which make it easy to actually know what and how should be tested.

Tasks with well written description in issue trackers are, unfortunately, not very common. Often they are hastily created, and the task's creator often is not the one who dictates the task's definition.

Consider this: when you are given a task with poor description, do you have any idea how to approach it? Most likely you either have to talk to the author of the task, or find somebody that might have an idea how to do the task. It works the same for the testers – if the tasks are not descriptive, how will they now how to test it? You may either put them in the same situation as you were when you started working on the task, or you might do what the author of the task could have done to help you – provide appropriate description.

In the end, you want your teammate-tester to assure that your task is indeed well done and everything is working correctly, right? You can easily help him or her to do so by providing useful summary of:

- **What you have done to complete the task** – this might consist of a technical part that would be useful for other developers who might once stumble upon your task in the future when working on something similar, or investigating some related malfunction. Enclosing some screenshots (where applicable) that present changes in the system is also helpful – *a* screenshot *is worth a thousand words* (contemporarily speaking).
- **How to actually test your bug solution or new functionality** that you have developed – this is really, really helpful – anytime I started working on a new project and I provided information for the testers after I have finished a task, they were always appreciative and told me it was helpful. You may also add some test cases that you think are important and should be prioritized.

This is neither hard work, nor it is time consuming. It is, in my opinion, one of those small things that do make a difference (*Element 3 – Care About Small Things*). Consider writing information for the testers as a part of your *definition of done*. By *definition of done* I mean steps that you consider necessary to be taken to regard a task as finished.

Whenever you finish a task, spend a few minutes to write down what you have accomplished. If the task will be assigned to a tester, provide useful information in the comment that will help the tester do the testing. It is also a good thing to provide some test cases that you think are the most important to be tried. Making some screenshots, when applicable, also works greatly.

Element 16 – Comply with Project's Code Style Guide

Spaces or tabs? I bet you heard that one before. Every developer has a coding style of his or her own. You can write and format your source code in so many ways, and there are different convention you can use. Most of the time, you work with other developers on the project. It is unlikely that everyone will comply with the same conventions and formatting style.

But does style and formatting differences really matter?

They do. They actually matter very much. They are a part of the attention to details from *Element 3 – Care About Small Things* and they contribute to the overall quality of the code (*Element 2 – Make Quality Your Driving Force*).

To understand why, consider a source code written by one person. One day, another developer will open the file and add something to it. If he doesn't comply with the file's code style, part of the file will look differently. And now the crucial point in this story – a third developer has to change (or add) something in the file – he/she sees that some parts of the file are written and formatted differently than the other parts. "Which style should I use?" he/she might think. Since two different styles are already being used in that file, why not style and format the new source code in a way that is familiar to the third developer (which will probably differ from the styles of the first two developers). **As time goes by, the file will probably change many times, and it's style and formatting will be a contribution of many developers. As a result, it will not comply with *anyone's* code style guide.**

Why do we want to keep the source code formatting and style consistent across the project? Because it makes it easier to understand the code and maintain it. When the whole project looks the same, it is easier to get into a different part of the code, analyze, and change it. You get used to a certain way of expressing logic through source code and formatting it.

It might look like not much of a deal, but **it actually is a tough work to keep the style and formatting of the source code in check**. Keep in mind that the less you and other developers care about the code, the faster it will deteriorate, the harder it will be to understand and maintain it. And somebody will, rather sooner than later, have to read the code and maintain it (*Element 9 – Think About Developers Who Will Maintain Your Code*).

When you join a new project, you might consider doing the following:

- Ask if there is a style guide – perhaps a list of conventions and guides, and a formatter configuration file, are available on your project's wiki-like site?
- If there is no such guide, adhere to the style of the file you are changing – keep the source code conventions and formatting the same all over the file.
- Propose a style guide – ask your new teammates to work together on a code style guide.
- Refactor if necessary – if a file violates some conventions that are crucial to you, consider refactoring after discussing it with your team.
- When you create a new file, comply with the style of code seen elsewhere in the project.

To conclude, comply with the project's style guide to keep the code consistent, whether the style, used conventions, and formatting, are described in some project's document or you just have to check how it is done by looking at the already existing source code.

Element 17 – Read What You Wrote Before Hitting "Send"

Whether we like it or not, we spend part of our time sending e-mails, writing documentantion, description of tasks in issue trackers, comments etc. How much time we spend on doing that is up to us, and with some additional effort we can make life easier for all people who will be reading it. It is a part of the *Element 3 – Care About Small Things*, tough I do not consider it a small thing.

Keep in mind – **whenever you write something, it will be written once, but read many times.** Let's say you write an important e-mail and make three typos and one of the sentences is formed in a way that is hard to understand. If your text will be read by five people, then the typos will be seen fifteen times, five times each, and every person will have to invest some time to understand this one ambiguous sentence. The typos will not look well, and not everybody might understand the sentence in the same/correct way. Now consider that you might have avoided all of that if only you had took a moment to read what you had written.

It is a good habit to go through the document/e-mail/question/comment etc. that you have just written down and correct all the typos you have found and improve the text as much as you can (also check out *Element 10 – Value Time of Other People by Asking the Question the Right Way*). **You want it to be easily understood – after all, that's why you have written it in the first place.** This is even more important when you are writing an official e-mail to the client. Make sure it is spotless as you represent both: the company you are working for *and* yourself.

Before you hit that *Send* button, read what you have just written. You will most likely find a few typos, wrongly-placed words, and sentences that could be improved to be easier to understand. The better the quality of the text, the easier it will be for the readers to grasp its meaning. No matter what you write, if somebody else will read it, take a few moments to make it as good as possible. You might even thank yourself later for it, cause one of the readers... might be you!

Last but not least, this chapter is about importance of taking care of your source code repositories.

Element 18 – Care for the Code Repository

Taking care of a code repository? Is that really something of any importance? Well, it is, and by a great deal. It might be considered one of the small things I mentioned in *Element 3 – Care About Small Things*, but it is neither small nor a detail. It is more of a part of the *Element 2 – Make Quality Your Driving Force*.

Source code repository holds history of all changes introduced by developers that used to work on the project, and of those that still do. Once you push your changes to the main branch, your commits can be seen by other developers, and **will be available to all developers who will join the team in the future**.

Consider this – people around you will likely, at some point in time, go through your code. Your name will be next to your commits. Would you rather have them looking at neat, elegant commits with descriptive messages, or disorganized, jumbled commits with "fix" messages? **What and how you commit speaks about yourself** – taking a few minutes to organize your commits and coming up with descriptive messages goes a long way.

Also, think about *Element 9 – Think About Developers Who Will Maintain Your Code*. Whenever you commit your changes, consider whether the person that will be going through your code will have an easy time understanding why you had changed what you changed (more in *Element 21 – Break Changes Into Commits*). Here's a short checklist of things you can look at when working with your team's source code repository:

- Neatly organized commits – group related changes together, separate them from refactoring, split changes to show your progress. More about that in *Element 21 – Break Changes Into Commits*.
- Descriptive commit messages – take a moment to write a message that will explain why certain changes have been done. Avoid "fix" and "TASK-123" messages. More in *Element 20 – Write Descriptive Commit Messages*.
- Compliance with your team's:
  - branching strategy – part of how different versions of your project's software are handled, and how new features are introduced – it is really important to comply with those,
  - branch naming – teams often have a branch naming convetion – it's good to stick to it,
  - merge strategy – maybe your team likes to keep their change history simple, without too many merge commits, and developers are supposed to rebase often?

Care for the code repository. Consider it a part of your *definition of done* to neatly organize your commits for any developer that will one day go through your changes. After all, your commits carry changes that are important – otherwise you wouldn't make the them in the first place.

Element 19 – Master Your Version Control System

Applications with graphical interface, sitting on top of version control systems, are a nice addition to developer's daily work – they can really help out in browsing history of changes, staging files for committing, simplifying merging process etc. However, they will not subsitute powerful commands you can use from the command line, and using graphical interface for many operations is much slower than doing the same from the command line.

I am not saying they shouldn't be used, but I think that firstly one needs to understand the version control system he or she is using, and then try some GUIs and see how they can facilitate using the version control system even further. One thing that using GUIs without learning the ins and outs of a version control system is that you lose your control over what is happening (what commands the GUI is using under the hoods)?

There are four reasons why it is good to take time to know your version control system:

1. Better understanding of what is happening – when you know, to some extent, what is going on under the hoods, you have a better grip of what the version control system is doing for you. You are then able to better use the version control system's commands, as you know how they interact with each other and what you can achieve with them.
2. More comfort – from understanding comes comfort. You know what will happen if you use this command, followed by the next, and the next. Whatever you have to do, you are already thinking about the combination of commands to be used. Because you know the commands well, using them feels easy, and, what is even more important, you feel *safe* using them.
3. More control – understanding and comfort bring control. You know what the commands do and you feel comfortable using them. You can do much more from the command line than by using GUI. Combining a few commands allows you to do what you want with your source code.
4. Working faster – controlling your source code versions by getting a lot of work done with a few commands is way faster than most of the same actions achieved by clicking in GUI. You will be able to do much more version-controlling in less time.

Master your version control system to fully utilize its power. You will gain more control over the version control process, understand it better, and feel comfortable doing it. You will also save a lot of time as doing most of the things from the command line is just faster than clicking it in a GUI.

Element 20 – Write Descriptive Commit Messages

Commit messages are the first thing you see when you browse through the history of changes of a repository. Unfortunately, too often do we see the history looking like this:

```
fix refactor
refactor
TASK-789
fix
TASK-42
```

Such commit messages are not helpful at all – they don't tell you much. You have to go through the actual changes to get an idea of what was actually changed. What is worse is the fact that **it would only take a moment to think about a commit message that explains the commit's purpose.** It's one of the small things that do matter (*Element 3 – Care About Small Things*).

Coming up with a short, yet descriptive, commit message is not an easy feat, but it also doesn't take that much of our time. Consider that you will write the commit's message once, but it will be read many times by other developers – would you rather have them reading *"Add displaying recent search entries"* or *"TASK-1401"* (*Element 9 – Think About Developers Who Will Maintain Your Code*)?

Commit message should be a summary of the introduced changes. There are many ways to format your commit messages:

- How long should they be?
- How should they be structured?
- Should there be additional information provided beyond the initial brief description?

Different teams have different approaches for writing commit messages – it is one of the things that you can discuss with your teammates to find the solution that works best for you (*Element 16 – Comply with Project's Code Style Guide*).

Whenever you commit your changes, take a moment to think about a brief, but descriptive, commit message. **Keep in mind that the commit will probably stay in the repository's history forever, and your name will be next to it.** Descriptive commit messages help you and other developers go through the history of changes and easily understand their purpose.

Element 21 – Break Changes Into Commits

This last Element was a subject of many discussions with my teammates over the years. Many of them did not agree with my point of view. I wonder what you will think about it?

I consider splitting changes into separate commits a good practice (as a part of *Element 3 – Care About Small Things*), where each commit, with a descriptive commit message (*Element 20 – Write Descriptive Commit Messages*), is a chapter of a story. Going through those *chapters* should allow the reader to understand why and how the application has changed. An important assumption here is that each of the introduced commits does not break the application (i. e. the tests pass).

There are two benefits to this approach:

1. Firstly, the person making the code review (*Element 1 – Have Your Code Reviewed*) will have an easier time going through the changes analyzing if they are correct, comply with the project's best practices (*Element 16 – Comply with Project's Code Style Guide*) and achieve what they were supposed to.
2. Secondly, it will be clearer (*Element 9 – Think About Developers Who Will Maintain Your Code*) for the people going through your changes in the future to see what, why, and how had changed, since the commits will be describing it part by part.

There are two good candidates for splitting commits:

- working on a large functionality – you can plan your work, deliver it piece by piece in separate commits with related changes,
- refactoring – it often encompasses many files and a brings lots of changes – if you can split those changes into separate commits to show consequent iterations of refactoring, you can help your code reviewers analyze the refactor process.

As I have mentioned, many of my teammates have disagreed with me on this approach. I've often heard that the commits should always be squashed into one commit. Although I disagree with this practice, there are some cons of commits-splitting I propose in this Element that might be worth talking about:

- It takes time to structure your commits this way – it does, but if you agree with me, it is a time worth taking.
- It is not always easy to have separate commits for related changes – sometimes it might be challenging to separate commits without breaking the tests, or there are just too many changes to be able to split them in a good manner. If that's the case, next time consider how you can approach a task to be able to prepare separate commits with related changes.
- Changes in next commits can override changes from previous commits – it is often difficult to take everything into account and sometimes we will change our code in newer commits, overriding our work from previous commits – a reviewer going through those first commits might spend time analyzing code that is no longer there. In this situation, you can move the change (where applicable) to one of the previous commits to tackle this problem.
- There are too many commits – if there are many commits, then if they are neatly orgnized to contain related changes, it shouldn't be a problem in the long run.
- Looking at all combined changes from some task requires using a command that squashes history log – that's true, but it's a matter of using a command that can do that.

Whether you agree with me or not, consider this: when you submit your code to code review (*Element 1 – Have Your Code Reviewed*), will reviewers have an easy time analysing your changes and will they be able to tell whether your code doesn't break anything and comply with the best practices? Perhaps splitting changes into smaller chunks would make it easier for them. Also consider that neatly organized commits might help anyone in the future, who would be going through the history of changes, to understand them.