Łukasz Wróbel

Memoirs of a Software Team Leader

Memoirs of a Software Team Leader

Łukasz Wróbel

This book is for sale at
http://leanpub.com/memoirs-of-a-software-team-leader

This version was published on 2014-09-14

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Łukasz Wróbel by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought an eBook ”Memoirs of a #Software #Teamleader” #MemoirsTeamLeader

The suggested hashtag for this book is #MemoirsTeamLeader.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#MemoirsTeamLeader

Contents

1. Leadership

Regardless of whether you create software or serve meals, it takes more than domain knowledge to become a team leader. The defining trait of a leader is his responsibility not only for himself, but also for the rest of the team. It requires specific personality attributes to be a good leader, though I believe most of them can be developed. The main limitation is your will to improve.

In a nutshell

If I was about to describe what it takes to be a good leader using as few words as possible, I would say:

- know your stuff;
- know your staff;
- don't be a jerk;
- don't be weak.

Being given more words, I would probably enumerate the crucial personality traits and say that it's good to be conscientious, reliable, fair, capable, imaginative, observant, precise, confident, keen (without exaggeration, though), encouraging and helpful. The truth is, I don't believe such a list could help anyone to become a better leader. This is why I'm going to focus on the crucial aspects, mainly because everything else is just a detail that you may start to think of and work on somewhere in the future.

Know your stuff

Knowing the things that you and your team work on is essential. While a CEO or even CTO is responsible for setting the goals ("Increase the number of photos being uploaded per second" or "Reduce response time by 15%"), a team leader is more or less directly responsible for the **execution**. Not only you need knowledge to make decisions, but you also have to serve as a mediator between the management and the team.

From my experience, being good at technology is the most sustainable way of gaining respect among the team members. I admit you can achieve some results quickly and without too much effort, for instance by intimidating people. However, apart from being totally wrong, this strategy won't work on a long-term basis.

It is desired to have a good overview and to be flexible, which means that:

- You're not tied to any specific language or technology. Even though I haven't been using any functional language in my professional career, I borrow some concepts from them at times, which leads to writing less imperative and in consequence less error-prone code. It's also true that there are no perfect libraries and frameworks around, every single one has its limitations or drawbacks. But if you're familiar with a few, you'll know which one to use in a specific case and where to borrow good ideas from.
- You know not only IT, but also computer science. Working knowledge on algorithms and data structures quite often turns out to be useful as it allows to save

resources and distinguish between doable and not-doable things.

I remember a guy trying to write a BBCode parser only using regular expressions. If only he knew it's not a regular language, and as such can't be described with a regular expression, he would probably opt for a stack-based algorithm. I won't forget him trying to predict all possible nestings; if I was his team leader, I wouldn't let it all happen.

- You've worked in a few companies.

  Working in one company for a long time may build a false sense of confidence. When facing a new problem, though, one may instantly realize he's defenseless. Don't get me wrong, I'm not encouraging you to become a serial job hopper, I'm rather trying to say you can't let yourself become overconfident. I also believe that experience gathering and constant learning is the only way to develop a healthy sense of confidence.

  Working in different environments lets you gain a wide perspective. Sometimes you work for an internal customer, and sometimes for an external one. You work with talented or lousy developers. Sometimes you're free to choose any technology you want, at other times you have to follow strict guidelines. All of these experiences make you more likely to say "I've seen that already, I know what to do!" whenever a problem appears. Remember that the best measure of intelligence is the ability to **adapt** to new situations; having a proven record of successful adaptations is a good sign.

Knowing is half the battle and sharing is the other half. Your team will make no use of your knowledge if you opt for

being selfish. Talk to your developers, show them how to solve problems more effectively and if they make mistakes, tell them not only how to fix the code, but also **why** the solution you proposed is better.

If you won't be willing to help, sooner or later your teammates will stop asking you questions. This will cause development to take longer because instead of actually working together, you'll be endlessly exchanging code review comments.

Know your staff

If you want to be more than just a person who shows up and gives orders, you must get to know your team. Sometimes the relationships stay at work, but quite often things get more personal. The latter is more likely if there are not too many differences between you and your teammates, especially in the terms of age. The truth is, in most IT companies there are no strict rules set when it comes to e.g. calling people by first name. And if you feel there is too much tension, you can always try going out for a dinner or having a beer with your team after work.

I encourage you to avoid treating each other formally, as lowering communication barriers makes people more likely to speak about their problems. It's much better to prevent people from deciding to quit than to find out what's been troubling them when it's too late.

You may like your co-workers or not, but you need at least to know them. Knowing developers' abilities and limitations, combined with knowing what they're interested in, will make you plan and assign tasks more sensibly. For example, I personally used to struggle with front-end related tasks, while

working on storing and processing data has always seemed to absorb me. I've always emphasized it, so that everyone around knew what type of tasks I should be assigned to.

It's not always possible to make everyone happy with their tasks, though it's good to be aware of teammates' preferences.

Don't be a jerk

One of the ways of getting to know the real nature of people is giving **power** in their hands. For some, even very little power is enough to corrupt them. As opposed to that, some of the most influential people can resist overusing their power.

If you're given any power, including the power derived from being a team leader, you should use it for other people's and your own good. How can having power be beneficial? Let's consider collaborative decision making, which quite often leads to endless discussions. Having someone designated to moderate them makes it easier to avoid getting stuck, which is highly beneficial for the company.

In software development, misusing power as the leader is often connected to forcing people to do something they totally don't agree with. If they come up with a better solution, the leader should talk it over with them and either decide to implement the idea or find its weaknesses and abandon it. He should also remember that the lack of a broader perspective is a common cause of misunderstanding. Due to a larger experience and being in touch with the management, the leader has better background than developers. This is why he should explain why certain decisions have been made instead of presenting his team with a fait accompli.

Being a leader is particularly about setting a good example. Have you promised to do something and forgotten about it? Be careful, because even if you haven't noticed, your teammates are watching you closely. If they see you don't keep your promises, not only will they lose some of the respect for your person, but they'll also be more likely to behave like you did. Think twice before doing something nasty, as it may go back to you one day.

We're all humans and we have our flaws. The long list includes more or less unexpected mood changes, which affect not only the person experiencing them, but also everyone around. If you work in a team, you should learn to isolate your emotions and don't let the mood overwhelm you. In extreme cases it would be better for you to stay at home, however, most mood changes can be quite easily coped with. Learning how to do this is particularly important for a leader, as his work quality affects the entire team. If you're feeling blue or experience a bad hair day, better let everyone around know about it and don't talk to people more than necessary. This way they'll know what to expect and won't be bothering you with questions like "What's wrong with you?" all day. On the other hand, if you're feeling extraordinarily well, you should use it as an advantage and try to tackle something that requires a lot of self-determination, something that you've been putting forward since a long time.

Not being a jerk may be interpreted as trying to be a person everyone would like to work with. However, this advice may do more harm than good when taken literally. Instead of trying to be amicable all the time, at least try not to initiate conflicts if you don't expect them to be beneficial. Not to mention that the definition of "the person one would like to work with" may vary and that a leader who follows it can't

be usually counted on in a crisis situation.

To me, not being a jerk also involves accepting constructive criticism. It's not always easy, especially if you're new to the team. You'd like to build your authority first and it's quite natural then to be afraid of admitting you make mistakes as doing it may undermine your position. However, as the leader you need to find a balance between strength and your erroneous human nature. A rule of thumb is that revealing your flaws remains unharmful as long as you achieve successes. To me, it allows to build a more healthy relationship than just being bossy.

Try to adapt to your team's needs and create an efficient work environment, but don't forget that he who pleased everybody died before he was born.

Don't be weak

This advice may seem to be at variance with the previous one ("don't be a jerk"). Sometimes it really is, as taking care of the team in its entirety is different than taking care of each member separately.

Besides personal issues, the main argument against being weak as the leader is that every team needs decisions in order to keep going. If everyone has its own vision, the team is unlikely to achieve its goal, because it's unable to achieve *any* goal. It's a great idea to keep everyone involved in the decision making process, though there are times when someone (that is, the leader) has to cut the discussion and make a decision by himself.

If your influence on the team isn't strong enough, you may quickly lose all the respect you've gained. If your voice can't

be heard or can be altered without providing arguments, you're not the leader anymore. "Live and let live" is a good principle, but up to some point only.

Not being weak also means removing obstacles instead of waiting for them to disappear. If you know that a problem will solve itself, spending any time on sorting it out will be a waste of time. However, if something's surely going to prevent your team from functioning properly, you should act without any delays. The longer you wait, the more likely you are to forget or suffer from lack of time and end up being caught unprepared.

Hands-on or hands-off?

Whether you're a hands-on or hands-off leader depends on the company and on the leadership style you prefer. In some companies, not writing any code and "only" being a leader is perceived as a waste of money. In other companies there's no allowance for the leader to do anything that might involve getting his hands dirty.

As the leader you're obligated to secure efficient work environment for your team in the first place. Only if you've done everything you could to achieve it, you can start to think about writing code. If you prefer working on tasks than being a leader, be cautious as you may get so attracted that you won't notice serious problems your team suffers from.

Switching from hands-on to hands-off mode and vice versa can be quite refreshing, though. If you're not used to write code and you try again after a long break, you have a chance to encounter problems developers have to face every day.

Maybe there's something wrong with the software environment, which you didn't even know about? Maybe one of your developers will be thankful for a chance to review the code you've written, thereby getting revenge on you? Getting hands dirty at least from time to time definitely helps to stay in touch with reality.

Delegate!

As the leader, you have the power to delegate duties to your teammates. You should use it, what's more, you can probably delegate more than you believe.

There are people who are strongly convinced they're the only ones in the entire universe capable of doing things right. They often end up working overtime, as one person can handle strictly limited number of tasks. Being ambitious is valuable, but every leader has to learn to distinguish between ambition and self-righteousness. If you suffer from this kind of affliction, you need a treatment and I think I can help.

First, pick up a task that fairs the best in the terms of delegation and bow it out. If choosing a delegable task is difficult, you can employ a heuristic and choose the smallest one. Try it once or twice and you'll see that not only you'll finish more things on time, thereby improving whole team's efficiency, but you'll also build up your co-workers by making them more empowered. Not to mention that more knowledge is being spread among team members then, which proves to be life-saving in emergency situations.

Work delegation is a part of a wider problem, namely, planning your work as a leader. This reminds me of a simple, yet powerful technique. It's so general, it can be partially applied

in other domains including your private life. It goes like this: if it seems you should do something, answer these questions first:

1. Does it have to be done at all?
   If not—forget about it.
2. Does it have to be done by yourself?
   If not—delegate it.
3. Can it wait?
   If so—make sure it's going to be done by fitting it into your schedule or adding it to your "actual to-do list".
4. Do it.

You may begin every work day by filtering your tasks through the list above and taking a look at your calendar or the list of previously approved tasks. You'll see for yourself how it helps to declutter your life and in consequence makes you more effective and satisfied.

Getting real

That's one of the most useful, but at the same time one of the least obvious things you should learn. It's important because as the leader you're able to make decisions by yourself. What's more, you're probably convinced that the decisions you've made so far are right, unless reality has already verified those claims.

What I want to encourage you to is "getting real", especially in the domain of decision making. It goes like this: if you're about to make a decision, ask yourself a few questions: "Is this thing I'm about to do *actually* required?" or "Does it solve

the very problem or another one instead?" and such. Let me provide you with a few example situations, so that you can get a better understanding:

- The button your team has added recently doesn't work in the X version of the Y web browser.
  The first question that probably comes into your mind is "How to fix it?". I suggest asking yourself a different question instead: "Is the X version of the Y web browser used by a significant amount of users?" It may turn out that only less than 1% of users haven't updated their browser yet and there's no point in spending time on fixing the bug. What you can do instead is provide a message like "Your web browser is quite old, please consider upgrading it to the latest version: [link]. If you don't upgrade it, some parts of the website may not function properly."
  Providing such a message is much easier and less time-consuming than struggling to fix things preserving compatibility with newer, thereby more important browsers.
- That new feature would require much storage space if it became popular.
  You may start wondering how many new disk arrays you will need. However, to me the question is: "Are you sure that this feature will become popular?" The answer is yes when, for instance, the new functionality will be bound to already existing and widely used one. But if you can't be certain of the new feature's popularity, you also shouldn't assume that buying new hardware is a must.
  In the beginning, try to use some adaptive and scalable solution like cloud storage or LVM to handle growth of

disk space requirements. Consider placing an order for hardware only when you're sure you're actually going to need it.

- You think that the new feature can be handled using hardware you already own, because there's enough of it to cope with regular amount of traffic.
  It's not the average load that will cause problems, it's the peak load. If your application is being heavily used on the weekends, don't do calculations by looking at the load on Wednesdays. Taking this into account, the question should be: "Are you able to survive the peak load?"
- One of your developers proposes to use a template system to produce HTML output.
  The question you should ask is "Why should we use it?" or "How can we benefit from this system?" There's a chance that the answer will sound like "Everyone uses it, so why shouldn't we?"
  I'm writing this because I've been using template systems without even thinking whether they actually help me or not. The truth is, most of them were adding significant overhead and offering very little in exchange. These days, depending on the technology, either I don't use any template system or pick one that actually facilitates development.
- When developing new features, your team accidentally reveals serious errors in an old functionality.
  Everyone involved (including the customer) will probably ask you to fix those errors. My opinion is, if for a long time no one has even noticed something has been wrong, then maybe that error is not that troublesome as everyone wants you to think?

Going step further, perhaps the whole functionality is no longer needed? Instead of fixing the newly discovered bugs, it may be better to get rid of unused code. It takes some time, but it will decrease maintenance costs in the future.

- You're afraid that having to sort user messages by date is going to have a huge performance impact.
  Let's assume that the API which messages are being taken from doesn't support ordering by date. Perceiving this as a potential problem, you're trying to discourage the customer from introducing the sorting functionality.
  But wait, doesn't it look like an example of a premature optimization? It may turn out that users have an average message count of 100, which means you can download all the messages and sort them "manually" before displaying to the user. And if anyone has a way more messages than a hundred, you can return the messages in default order.

As an exercise, try to think of something you're working on and ask yourself whether you're doing it the right way, the "real" way. Perhaps you shouldn't be doing it at all?

One of the most spectacular ways of using the "getting real" technique is interrupting heated debates. Imagine yourself taking part in a meeting where everyone is keenly discussing something, going into more and more details every minute. In the beginning you even join the discussion and provide some arguments, but then you start to feel that instead of getting closer to finding the solution, everyone is making up new problems. You also realize that the problem everyone was trying to solve in the beginning is not *that* important

and, what's more, no one has even noticed it. Then you ask "Why are we talking about all this in the first place?" It's unlikely that someone will provide a meaningful answer. After pushing the reset button this way, the thinking process can be started over, hopefully bringing better results this time.

"Getting real" means that whatever you do, you must do it for a reason. When making a decision, try to imagine a smart and questioning adversary:

- "Do you actually need it?"
- "Are you doing it because it's reasonable or only because you're a crowd follower?"
- "Is it really going to be used this way?"
- "Doesn't it create a vulnerability?"

If you answer these questions in advance, not only you'll make better decisions, but also become better at explaining reasons why certain decisions were made. Mainly because you'll have good reasons.

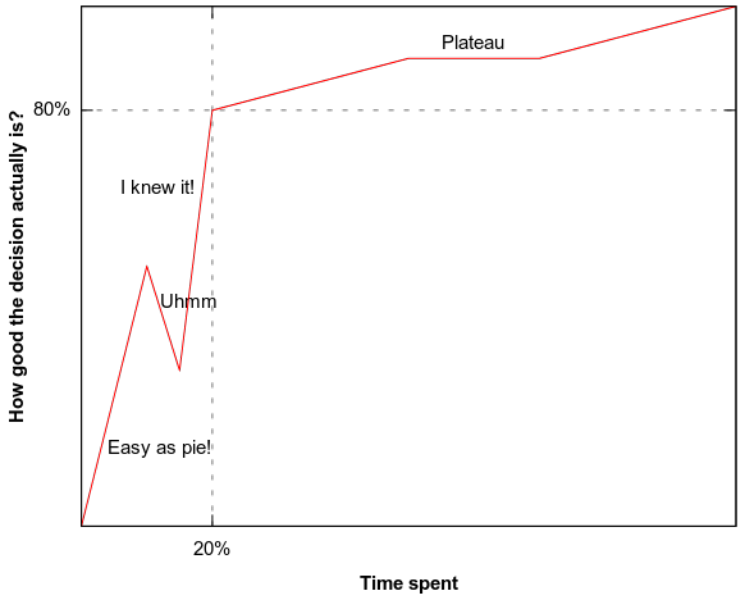
Decision making

Except for rationality, good decisions are those made quickly.

In the software development process, questions like "Should I use this method or rather that one?" are very common. Some of them are easily answered and don't require leader's involvement, though there are some who require strategic or "political" thinking. And this is where you come in.

Besides applying the "getting real" technique described above, you should remember not to try to make **perfect** decisions.

I'm saying this because I've learned that any decision is better than no decision. Pareto principle may be turned into good use here: there's a chance that by spending only 20% amount of the time required to make a perfect decision, you'll make a decision that will be 80% as good.



Pareto principle applied to decision making

The image above shows the Pareto principle applied to decision making. Of course, you shouldn't treat it literally as not every time things look that simple, though (here I apply the principle again!) the chart should cover about 80% of possible scenarios.

Let me describe the chart itself:

1. Usually, in the beginning there's a lot of optimism ("Easy as pie!").

2. However, on the second thought ("Uhmm") things doesn't look that simple anymore.
3. Then you come up with a better idea ("I knew it!"). At this point, you've reached 80% of the best possible effect.
4. You can spend some more time on thinking, though you won't achieve significantly better results. Probably you won't discover any surprises, but on the other hand the new ideas will be no better or even worse than the previous ones. Congratulations, you've reached the point of stagnation ("Plateau")! Was spending 80% of the time this way a good idea?

If you're not sure about something or simply want to compare everyone's opinion, organize a quick meeting. It doesn't have to involve booking a meeting room—a hallway meeting or talking in your room will be just fine. Keep in mind that the cozier it gets, the longer such a meeting will take. If people don't feel comfortable, they'll be more likely to stay focused in order to finish the meeting as soon as possible.

Such a meeting can be **time-boxed**, which means that if no brilliant conclusion has been reached, you have to pick the most promising one. It's very similar to the way that daily meetings (which I'll talk about later) should be organized.

Let me share with you another thought concerning collaborative decision making. Winston Churchill said that the best argument against democracy is a five-minute conversation with the average voter. One of the differences between governing a country and managing your development team is that you have direct influence on people. You can hire those who are good or at least seem promising. You can talk to them

in person if there are any problems. Finally, you can give them walking papers if they can't act as team players (at least, you have something to say about it).

This all means that you probably work with the right people. If so, Churchill's argument doesn't suit your situation, does it? But there's another problem: clever people usually have clever ideas and each of them may have ideas of his own. This is why from time to time you have to put yourself in King's Solomon's shoes and resolve a dispute.

Resolving conflicts

If there's more than just one man around, a conflict may happen. As the leader you should embrace this fact and learn how to benefit from it.

In general, there are two kinds of conflicts, I would call them **unjustified** an **justified**. The former are considered harmful since there's no logic behind them. They happen when one person doesn't like the other and tends to automatically dislike everything that person says or does. No one will benefit from such conflicts and the only way to prevent them is to talk in person to the people involved. They may not like each other, but they shouldn't let it influence their work. At least, this is how mature people should behave.

There are times when specific unjustified conflicts seem unsolvable. Then you can either try to designate an external mediator or if it doesn't help, consider dismissing the trouble-causing person. Just make sure you've done all the reasonable things you could to avoid it.

Justified conflicts, as opposed to unjustified ones, have factual causes and will bring improvements if used well. Since you

can tell what the fuss is about, you can also do something about it. Organize a short, in-place meeting or talk this over in person and do exactly the same things as in the case of a "lessons learned" session: make people stop complaining, work out a solution, implement it and verify whether things have actually changed some time later. Fixing a deadline is a good idea, as it makes people more likely to exchange aggression and criticism for positive energy. After all, they were listened to and they both gave a promise and were given one. If they really want to make things better, they will keep their promise. You have to be a man of your word, too, because prolonging a conflict and not actually doing anything to head it off will have its revenge in the future.

Conflicts may be time-consuming, as they require attention to be solved properly, plus they're likely to decrease people's efficiency for some time (even after being more or less resolved). Yet, I mentioned that a conflict may be something to benefit from. Why do I think that? It's because it's good if people talk about their problems openly. Conflict is nothing more than one of the ways of indicating there's something wrong. Perhaps you've missed it and could have prevented the conflict. You failed, but it's not all lost yet. Appreciate the conflict and do something to improve your teammates' working conditions or you'll see them giving up their jobs.

The hardest thing you may have to face is when it turns out there's nothing that you or anyone around can do to solve the problem. It's likely to happen if your team or the projects you're working on depend heavily on external issues like constantly evolving law regulations or an irritating, but well-paying customer. If you can't do anything about those things, you can only try to neutralize them somehow. Facing the facts is one of the ways of achieving this, making fun of

them is the other. However immature it may seem, hanging a *slightly* modified photo of the person being the root cause of all evil or transforming this photo into a dartboard may help to relieve the stress. If the negative emotions are being accumulated, they'll surely cause troubles.