Lessons Learned in Software Development

Posted on April 16, 2015 | 84 Comments

Here is my list of heuristics and rules of thumb for software development that I have found useful over the years:

DEVELOPMENT

1. Start small, then extend. Whether creating a new system, or adding a feature to an existing system, I always start by making a very simple version with almost none of the required functionality. Then I extend the solution step by step, until it does what it is supposed to. I have never been able to plan everything out in detail from the beginning. Instead, I learn as I go along, and this newly discovered information gets used in the solution.

I like this quote from John Gall: *"A complex system that works is invariably found to have evolved from a simple system that worked."*

2. Change one thing at a time. When you develop, and some tests fail, or a feature stops working, it's much easier to find the problem if you only changed one thing. In other words, use short iterations. Do one thing, make sure it works, repeat. This applies down to the level of commits. If you have to refactor the code before you add a new feature, commit the refactoring first, then (in a new commit) add the new feature.

3. Add logging and error handling early. When developing a new system, one of the first things I do is adding logging and error handling, because both are useful from the very beginning. For all systems that are

bigger than a handful of lines of code, you need some way of knowing what happens in the program. Perhaps not when it is working as expected, but as soon as it doesn't, you must be able to see what's happening. The same goes for error handling – errors and exceptions happen in the beginning too, so the sooner you handle them in a systematic way, the better.

4. All new lines must be executed at least once. Before you are done with a feature, you have to test it. Otherwise, how do you know that it does what it is supposed to do? Often, the best way is by automatic tests, but not always. But no matter what, *every new line of code has to be executed at least once*.

Sometimes it can be hard to trigger the right conditions. Fortunately, it's easy to cheat a bit. For example, the error handling on a database call can be checked by temporarily misspelling a column name. Or, an if-statement can be temporarily inverted ("if error" becomes "if not error") in order to trigger something that rarely happens, just to make sure that code is run and does what it should.

Sometimes I see bugs that show that a certain line of code can never have been run by the developer. It can look fine when reviewed, but still not work. You avoid embarrassment if your policy is to always execute every new line you write.

5. Test the parts before the whole. Well-tested parts save time. Often there are problems with integrating different parts, for example from mismatched or misunderstood interfaces between modules. If you can trust that the parts work as expected, it becomes much easier to track down the integration problems.

6. Everything takes longer than you think. Especially in programming. It is hard to estimate how much time a feature will take even if everything goes smoothly. But when developing software, it is quite common to run in to unexpected problems: a simple merge turns out to cause a subtle bug, an upgrade of a framework means some functions must be changed or an API call doesn't work as promised.

I think there is a lot of truth in Hofstadter Law: *It always takes longer than you expect, even when you take into account Hofstadter's Law*.

7. First understand the existing code. Most coding requires changing existing code in some way. Even if it is a new feature, it needs to fit into the existing program. And before you can fit the new stuff in, you need to understand the current solution. Otherwise you may accidentally break some of the existing functionality. This is means that *reading* code is a skill that is as necessary as *writing* code. It is also part of the reason why seemingly small changes can still take a long time – you must understand the context in which you make the change.

8. Read and run. Fortunately, there are two complementary methods for understanding code. You can read the code, and you can run the code. Running the code can be a great help when figuring out what it does. Be sure to make use of both methods.

TROUBLESHOOTING

9. There will always be bugs. I don't like approaches to software development that claim to "get it right the first time". No matter how much effort you put in, there will always be bugs (the definition of a bug pretty much is: "we didn't think of that"). A much better approach is to have a system in place that lets you quickly troubleshoot problems, fix the bugs and deploy the fixes.

10. Solve trouble reports. Every developer should spend a portion of their time handling trouble reports from customers and fixing bugs. It gives you a much better understanding of what the customers are trying to do, how

the system is used, how easy or hard it is to troubleshoot and how well the system is designed. It's also a great way of taking responsibility for what you develop. Don't miss out on all these benefits.

11. Reproduce the problem. The first step when fixing a bug is to reproduce the problem. Then you make sure that when the fix is added, the problem is gone. This simple rule makes sure you are not assuming something is a problem when it isn't, and makes sure the solution actually does what you think it does.

12. Fix the known errors, then see what's left. Sometimes there are several problems present that you know about. The different bugs can interact with each other and cause strange things to happen. Instead of trying to work out what happens in those cases, fix all the know problems and then see what symptoms remain.

13. Assume no coincidences. When testing and troubleshooting, never believe in coincidences. You changed a timer value, and now the system restarts more often. Not a coincidence. A new feature was added, and an unrelated feature becomes slower? Not a coincidence. Instead, investigate.

14. Correlate with timestamps. When troubleshooting, use the timestamp of events as a help. Look for even increments. For example, if the system restarted, and a request was sent out around 3000 milliseconds before, maybe a timer triggered the action that lead to the restart.

COOPERATION

15. Face to face has the highest bandwidth. When discussing how to solve a problem, being face to face beats video, call, chat and email. I am often amazed at how much better the solutions are after discussing them in person with colleagues.

16. Rubber ducking. Whenever you are stuck, go to a colleague and explain the problem to them. Many times, as you talk, you realize what the problem is, even if your colleague doesn't say a word. Sounds like magic, but works surprisingly often.

17. Ask. Reading and running the code is often great for figuring out what it does and how it works. But if you have the possibility to ask someone knowledgeable (perhaps the original author), use that option too. Being able to ask specific questions, and follow-up questions to those, can give you information in minutes that would otherwise take days to get.

18. Share credit. Make sure to give credit where credit is due. Say: "Marcus came up with the idea to try…" (if he did), instead of "we tried …". Go out of your way to mention who else helped or contributed.

MISCELLANEOUS

19. Try it. If you are unsure of how a certain language feature works, it is easy to write a little program that shows how it works. The same applies when testing the system you are developing. What happens if I set this parameter to -1? What happens if this service is down when I reboot the system? Explore how it works – fiddling around often reveals bugs, and at the same time it deepens your understanding of how the system works.

20. Sleep on it. If you are working on a difficult problem, try to get in a night's sleep before you decide. Then your subconscious mind works on the problem even when you aren't actively thinking about it. As a result, the solution can seem obvious the next day.

21. Change. Don't be afraid to change roles or jobs every once in a while. It is stimulating to work with different people, on a different product or in a different company. In my view, too many people just passively stay at the same job year after year, only changing if they are forced to.

22. Keep learning. One of the great things with software development is that there is always room to learn and know more. Try out different programming languages and tools, read books on software development, take MOOC courses. Small improvements soon add up to make a real difference in your knowledge and abilities.

RELATED

Working as a Software Developer
In "Learning"

Antifragility and Software Development
In "Programming"

Ph.D. or Professional Programmer?
In "Learning"

This entry was posted in Debugging, Learning, Programming, Work and tagged heuristics, lessons learned, rules of thumb, software development. Bookmark the permalink.

84 RESPONSES TO "LESSONS LEARNED IN SOFTWARE DEVELOPMENT"

Filipe Deschamps | April 20, 2015 at 12:35 pm | Reply

Awesome post! Years and years of knowledge, thanks for sharing!

> **Henrik Warne** | April 20, 2015 at 1:12 pm | Reply
>
> Thanks Filipe!

Johannes | April 20, 2015 at 1:19 pm | Reply

"1. Start small, then extend." This is a great way to start, but you always should keep the bigger picture in mind.
Building small – non general – systems can be easy in the beginning but pain when you have to add some more complex functionality .

> **Henrik Warne** | April 20, 2015 at 1:38 pm | Reply
>
> Hi Johannes,
>
> Thanks for commenting. I didn't make it very clear, but i know that the goal is the required functionality even as I make the first rudimentary version. This helps when completing the necessary features, even though I don't know how the system will evolve in the future.

Henrik Warne | April 20, 2015 at 1:39 pm | Reply

Hacker News discussion: https://news.ycombinator.com/item?id=9406465

Henrik Warne | April 20, 2015 at 1:40 pm | Reply

Reddit discussion: http://redd.it/337i1l

Padraig | April 20, 2015 at 2:38 pm | Reply

Nice post Henrik.
Point 20 – "Sleep on it". Amazing what stepping away from a programming problem you've been stuck on and spinning your wheels on can do.

Aaron Holbrook | April 20, 2015 at 3:17 pm | Reply

Reblogged this on Aaron Holbrook on WordPress.

Bill Torpey | April 20, 2015 at 3:25 pm | Reply

Re: "All new lines must be executed at least once." — it's interesting that Microsoft's standards require developers to step through all new or changed lines of code in a debugger, modifying values manually if necessary to ensure that every line gets executed. (IIRC, I first saw this in "Microsoft Secrets" by Michael Cusumano).

That seems like an excellent practice — it's amazing how much insight one can get from actually seeing the code execute. (Or, as Yogi Berra once said, "You can observe a lot just by watching" 😉

andomar | April 20, 2015 at 11:00 pm | Reply

You might enjoy the "Facts and Fallacies of Software Engineering" book. It has a lot of these points in one form or another!

> **Henrik Warne** | April 21, 2015 at 6:42 am | Reply
>
> Thanks for the tip! I think it's a great book! I wrote a review on it on Amazon back in 2002: http://www.amazon.com/review/R2H1CYIC6U5LBU/

ktest098 | April 21, 2015 at 4:41 am | Reply

I've been a professional software developer for a long time now, and every one of these items I readily agree with.

On the "Sleep On It" item near the end, that might come across as being mostly related to the problem solving section before. But it applies to every aspect of programming, including design. If something you are designing does not seem quite right sleep on it and let your mind unwind the correct path for the code to flow.

Erik Hanchett | April 21, 2015 at 6:08 am | Reply

"Everything takes longer than you think. " I agree completely. I think this is doubly true when you are new to a project or new to programming. Little issues just slow you down.

Chris Chalkitis | April 21, 2015 at 2:47 pm | Reply

Excellent simple rules to follow and succeed.

Ian Kaplan | April 21, 2015 at 11:12 pm | Reply

Thanks for the great and thoughtful post.
I've been thinking a lot recently about system development. One of the pitfalls for software engineers is designing and building too much. There is a good argument for building a small system (as you note) and then expanding it.
One of the projects that I'm working on is a scalable web app. If it succeeds it will go viral (or at least semi-viral), going from a few test users to tens of thousand of users. So I've put a lot of time into building infrastructure on Amazon Web Services.
The pattern with web applications that get noticed is that they get attacked and, frequently, successfully hacked. To avoid this I have carefully built in security features (salted passwords, locked down IP access).
In doing all this, however, I keep wondering if I'm yet another software engineer who has fallen into the temptation to build something needlessly complex.
I don't find any of these decisions easy. I suppose that in the end we do the best we can ("It seemed like a good idea at the time").

Chris | April 22, 2015 at 2:02 am | Reply

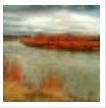
I'd be interested to hear a little more about giving credit to other people. Any stories associated with that tip?

> **Henrik Warne** | April 22, 2015 at 10:39 am | Reply
>
> 
>
> Good question Chris! I don't really have any good stories there, other than early on in my career I had a manager that did that very explicitly. Every time I was called out to have helped it felt very motivating to hear, so I started to do the same thing myself. I think being appreciated for what you do makes you want to keep doing a good job. I also think it is in some sense "morally right".

Shyam | April 22, 2015 at 10:10 am | Reply

Perfect article .. explained perfectly.

Andy K | April 22, 2015 at 10:26 am | Reply



Very nicely put. I now know I do a lot of "rubber ducking", I'd always referred to the practice as "four eyes are better than two" (which is odd in itself as I'm a pirate an always wear an eyepatch).

Maciek | April 22, 2015 at 11:34 am | Reply

An alternative or a complement to Rubber Ducking would be to write the problem down. Clarity in defining the problem often takes you straight to the solution.

> **Henrik Warne** | April 22, 2015 at 11:59 am | Reply
>
> 
>
> Yes! I like this quote from Charles Kettering:
> "A problem well stated is a problem half solved."

Roland Guilmet | April 22, 2015 at 1:39 pm | Reply

Excellent article. I'm a great believer in logs. Your team should agree on a standard log format and develop tools to process and analyze them (can be as simple as Excel and a macro). I've worked on too many projects where the log format was the whim of the individual programmer, thinking no one else was going to use it. Well, we know that's not true. Standardized log format means you can build one tool to help with log processing and analysis. Great for when you go into production, or have to

investigate issues from the field. Don't forget to enough depth in the log. It takes a long time to get information from the field. So log depth and management are also a consideration.

I'd also add that you can't automate a system or process if it doesn't already work well manually. How many times have you been on a project when the feeling was that programming will fix it.

> **Henrik Warne** | April 22, 2015 at 2:01 pm | Reply
>
> Thanks for commenting Roland. Yes, logging really is important. I've written more about logging in the past:
>
> https://henrikwarne.com/2014/01/01/finding-bugs-debugger-versus-logging/
> https://henrikwarne.com/2014/01/21/session-based-logging/
> https://henrikwarne.com/2013/05/05/great-programmers-write-debuggable-code/

raj lath | April 22, 2015 at 3:49 pm | Reply

Felt like something i have written .

kmoteoo | April 22, 2015 at 5:10 pm | Reply

Wonderful list! You really have a great knack for discovering hidden nuggets of wisdom and explaining them in a way that makes them seem like they should have been obvious all along (but too often aren't!). Thank you!

> **Henrik Warne** | April 22, 2015 at 7:56 pm | Reply
>
> Thanks kmoteoo!!

B. Clay Shannon | April 22, 2015 at 5:20 pm | Reply

I like the idea of sleeping on it. Now I have to decide whether to bring in a hammock or an air mattress.

Andrew Fry | April 22, 2015 at 5:28 pm | Reply

Good articles. I agree with everything, though Id add something about … seeking out and adopting design patterns and best practice … peer code reviews … seeking out and using whatever tools are available to help your test your code, eg online validators, Firebug for front-end web stuff, an IDE, etc.

Kurt Guntheroth | April 22, 2015 at 10:36 pm | Reply

"First understand the existing code". I heartily agree. However, I see a strong tendency for managers to prefer employees who rush to make changes, simply patching over the top of the existing code until it is a mess of overlapping and conflicting layers of spaghetti. Any words on how to live in that world?

> **Henrik Warne** | April 23, 2015 at 9:27 am | Reply
>
> Hi Kurt,

> Interesting question. I think we sometimes put that pressure to be fast on ourselves as well. But the times I've cut corners in understanding what's already there in order to finish something, I have usually regretted it later on. Of course it gets even worse if the managers push for fast changes without understanding.
>
> One thing that helps is if the existing code is easy to understand – for example functions with no state. But surprisingly often, the code ends up getting hard to understand once it has existed for a while (and been extended/modified many times). The best tactic I know is to use the "scout rule" (from e.g. Clean Code) – leave the code in better shape than you found it (when you modify something). In other words, continuously work to make the code base better.
>
> PS. I like your bog.

Jihdeh | April 23, 2015 at 1:14 am | Reply

Very good article. Thank you for sharing this. I will take this along all step of my journey in software development.

tharansakthi | April 23, 2015 at 5:32 am | Reply

nice post dude

crosh | April 23, 2015 at 5:51 am | Reply

I recently (as in started last Friday) playing around with Go – at the request of my employer who needs it for an interface to our internal RESTful API (it has something to do with Docker – and can only be done in Go apparently). So – I and a colleague were given time to learn Go and develop the system.

At first – it all felt weird (I still say Go is a mashup of Pascal and C) – we were literally fighting the compiler at every iteration of the code. Write some code, declare a variable, but don't use it? Go will complain and won't run. Declare a function, but never call it? Go will complain and won't run. Return a value inside an if-then but not outside of it – you guessed it…

It was frustrating. It was maddening (oh – and maps! they purposefully randomize them each time you run your code – so you won't get complacent about ordering!) – but what I and my colleague found was that our code – despite our lack of experience – despite out lack of understanding of Go – it worked – and it worked really well. In the course of only a few days, we went from zero knowledge of Go, to the basics of what our employer needed – and it all works.

Now – we still haven't tackled implementing unit tests (I've heard those can be a nightmare in Go?) – but we'll slog through them. I think that a lot of the points in this article could be mitigated by using a language like Go that doesn't allow you to get away with things and be sloppy (which was basically why Go was developed by Google, from what I have read).

Michael Eriksson | April 23, 2015 at 8:41 am | Reply

My own two implicit main rules (unfortunately disregarded by most of the colleagues I have worked with over the years):

o Use your head.

o Quality short-cuts tend to cost far more than they gain.

(I have never attempted this type of extended listing. However, I have written a fair bit on various similar topics. Cf. http://www.aswedeingermany.de/50SoftwareDevelopment/ , if you are interested)

Median Nguyen | April 23, 2015 at 9:07 am | Reply

Reblogged this on Trung Tuyê´n Nguyễn – Median Nguyen.

simonfrankau | April 23, 2015 at 2:07 pm | Reply

Great post. More people should read John Gall.

marcbrooks | April 23, 2015 at 5:08 pm | Reply

Sleep on it can ALSO be shower on it… 🙂

> **Henrik Warne** | April 23, 2015 at 10:15 pm | Reply
>
> 
>
> Absolutely!

Zigurd Mednieks | April 23, 2015 at 5:36 pm | Reply

I would add “Understand the bugs.” If you fix something, state the reason there was a bug along with the commit. If you can’t explain a fix, that’s a red flag.

Vương Hiê`n | April 23, 2015 at 5:44 pm | Reply

Thanks for sharing

Daaren Urthling | April 23, 2015 at 10:51 pm | Reply

Totally agree with this post! It’s amazing to realize how many of your best practices are part of my daily routine: I can confirm the other readers that they really work. About n.7, one of my favorite techniques (especially when I take code from some sample) is start changing the names of variables and routines to see if I can really glue all the pieces back together. Great job, keep on sharing!

Ralph Moses | April 24, 2015 at 12:53 am | Reply

Great, great article–I have used the same approach for close to 40 years. I would add two more items: First, use the binary tree method in debugging–is the problem in this half or that half, then this 1/4 or that 1/4…. Second, Use OPC (Other People’s Code) whenever possible as a starting point. I have written maybe 5 programs from scratch–all others sucked in known, working code from previous projects or other programmers.

Mercereaux | April 24, 2015 at 1:08 am | Reply

I appreciate how clearly and simply you’ve put all this. I agree 80%. the other 20% is the parts that i tend to skip over and wish i hadn’t.
I was ONCE able to roll out an application in less time than anyone thought possible. I’ll never do that again. I thought because it was the 4th or 5th instance of the same software in a different geographical location that it was safe. WRONG. The people didn’t understand and it turned into a horrible mess.

Start small and extend; change one thing at a time… those two have saved me many, many times. Execute every line of code? didn't recognize it quite that way, but have done that one too. Amazing what that'll save you from.

Kjell | April 24, 2015 at 5:30 am | Reply

Great post – obviously with lots of experience! I shared on LinkedIn 🙂

Zsuzsa Skeren | April 24, 2015 at 12:43 pm | Reply

Excellent post !!!!

mrwigster | April 24, 2015 at 4:56 pm | Reply

Great list.. #16 is so true! – Wonder if I could share this on http://trulycode.com ?

> **Henrik Warne** | April 24, 2015 at 7:07 pm | Reply
>
> 
>
> Thanks! Sure, go ahead!

horseshoe7 | April 25, 2015 at 9:40 am | Reply

Reblogged this on Thought Repository and commented:
Fantastic post!

brandonjwilhite | April 25, 2015 at 1:30 pm | Reply

This was a pleasing meditation on software dev. Thank-you.

Pingback: Lessons Learned in Software Development | Scott Brenner

shafiqksm | April 27, 2015 at 4:30 am | Reply

Reblogged this on mylearningbits.

Pingback: Five Blogs – 27 April 2015 | 5blogs

Musthaan | April 27, 2015 at 2:52 pm | Reply

Reblogged this on Musthaan.

Pingback: Уроки извлеченные из разработки ПО. | NSUkraine

seema | April 30, 2015 at 10:29 am | Reply

I want to learn this process

craigdolan | April 30, 2015 at 11:08 pm | Reply

Reblogged this on craigdolan and commented:
Brilliant blog post on software development in general.

Barry Wood | May 1, 2015 at 3:39 am | Reply

Should be a primer for a cyber entity.
My experiences.
A small company,team of 8.
A new project was reviewed, recorded sober, then totally smashed.
The whatifs, that emerged made my success.
Promoted,larger team, a public company, tried same, Fired.
Software success is not only good practice but understanding
the chain of incompetence.

T Nguyen | May 1, 2015 at 3:51 pm | Reply

Thank you for the post. It is great. I remember #4 the most. Yes, all new lines must be executed at least once.

martin1205 | May 3, 2015 at 6:48 pm | Reply

Great list! At first it felt like I'd heard it all before, but your practical view of it really added value.

Pingback: Regole di sviluppo del software (e non solo) | MelaBit

merezano | May 6, 2015 at 4:03 am | Reply

Trivia: name an author who wrote two of the books in the bookshelf pictured in this article.

Albrecht Scheidig (@scheidig) | May 6, 2015 at 10:42 am | Reply

Great post, comprehensive and process independent list!
In the troubleshooting section I would have add:
"Don't manage long backlogs of bugs. Backlogs are for features, but bugs should be either resolved as 'won't fix' or fixed with priority over features."

mayofcherries | May 8, 2015 at 9:55 pm | Reply

That is easily the best post i've read this year. It all seems obviously simple yet so hard to figure out sometimes.

Nikolay Bachiyski | May 9, 2015 at 7:56 pm | Reply

Reblogged this on ex·trap·o·late me and commented:
Solid, timeless, and buzzword-free software development advice by Henrik Warne.

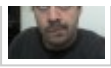
Pingback: PHP Annotated Monthly – May 2015 | JetBrains PhpStorm Blog

John Jackson | May 11, 2015 at 10:27 pm | Reply

Great post, good summary of all the small yet important things. I though this blog post which I just lately read would be a natural next step for good deployment http://stackify.com/ultimate-checklist-app-deployment-success/

Ricardo Miranda | May 12, 2015 at 2:40 pm | Reply

Excellent post. Is always good to read this kind of advice from more experienced developers.

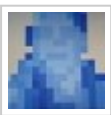


Pingback: "Lessons Learned in Software Development": A great post | i-proving.com

Michael Pickard | May 12, 2015 at 4:44 pm | Reply

This was a very interesting and well written post. I found points 1, 6, 7, 8, and 11 particularly thought provoking, and decided to share those thoughts on http://i-proving.com/

japollo27 | May 14, 2015 at 3:35 am | Reply



Reblogged this on JApollo.

Pingback: Compendium of Wondrous Links vol IX | Wrong Side of Memphis

Soluciones Web | May 17, 2015 at 2:23 am | Reply

What a great post. You did a great job listing those lessons, many of us have been dealing with this things since our first line fo code.

Pingback: ThoughtWire – Read This First – Heuristics I Wish I'd Known

markchagers | May 23, 2015 at 12:12 am | Reply

Wonderful article! It's eery how many of your points correspond to my own experiences (which I never articulated as well as you do).

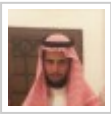
Deanna R. Jones | May 27, 2015 at 8:51 pm | Reply

You made a great point about how talking the problem out to one of your colleagues helps to overcome roadblocks in software development. Usually, I work alone, so it would be pretty nice to have another software developer to talk to about any problems that I'm having when working on a piece of software. I still work in a building with my colleagues, so perhaps I should seek someone out to talk to, even if they can't answer my question to help me come to the answer on my own.

Staff | June 1, 2015 at 1:39 pm | Reply

This is a nice post. Will keep some notes handy for our students.

ahimta | June 6, 2015 at 11:36 am | Reply



Reblogged this on ahimta.

nishant saini (@Nishantglobals1) | September 29, 2015 at 6:31 am | Reply

Hi , Nice Information For Software Development because software is field of changement as per the requirement of real world .....
Thanks
http://www.globalscope.in

Pingback: Professional Development – 2015 – Week 51

Pingback: Professional Development 12/21/15 – 12/27/15 | The Software Mentor

Scott | January 20, 2016 at 10:42 pm | Reply

Great article! Thanks for this!

Pingback: Lessons Learned in Software Development – The Nerd Cat

Nataraj Gandhi Arunachalam | June 9, 2016 at 7:42 pm | Reply

great article!

Pingback: Lessons Learned in Software Development | the DAKworks

---