Software Re-Engineering

Usman Ghani

Lecturer

Computer Science

Agenda

- Ch 10
- Stop writing legacy SW
- Update documentation
- Foster communication
- Periodic code reviews
- Automate
- Small is beautiful

Stop writing legacy SW

- By now you should have a good idea of how to start tackling any neglected legacy code that you inherit and nurse it back to health.
- We’ve looked at rewriting, refactoring, continuous inspection, toolchain updates, automation, and a whole lot more.
- But you probably spend at least some of your time writing new code as well.
- You may be wondering whether all code is doomed to become legacy, or if there’s anything you can do to prevent the code, you’re writing right now from becoming somebody else’s nightmare in a few years’ time.

Stop writing legacy SW

- We’ve covered an enormous range of material over the last nine chapters, but a few key themes kept appearing throughout the book, either explicitly mentioned or implicitly assumed.
- We’ve been discussing these ideas in the context of legacy code until now, but a lot of them are equally applicable to greenfield projects.
- These themes are as follows:

The source code is not the whole story

- From a programmer’s point of view, the source code is often the most important part of a software project.
- First, much of the work you do on legacy code will be refactoring.
- Second, and more importantly, I wanted to use this book to stress the idea that the source code is not the whole story.
- The most important thing, far more important than anything we can say about the code, is to build software that provides value to its users. If you’re building the wrong thing, nobody will care what the code looks like.

The source code is not the whole story

- Apart from that, there are plenty of other factors that affect the success of a software project. (I haven’t defined the notion of success, but it generally includes things like development speed, the quality of the resulting product, and how easy it is to maintain the code over time.) We’ve looked at a lot of these factors, some technical and others organizational.

Technical Factors

- Technical factors include selecting and maintaining a good development toolchain, automating provisioning, using tools such as Jenkins to perform CI and continuous inspection, and streamlining the release and deployment process as much as possible.

Organizational Factors

- Organizational factors include having good documentation, maximizing communication within and between development teams, making it easy for people outside of the team to contribute to the software, and fostering a culture of software quality throughout the organization so that developers are free to spend time on maintaining quality without facing pressure from other parts of the business.

Information doesn’t want to be free

- Sarcastic title of this section is, of course, a play on Stewart Brand’s famous declaration that “information wants to be free.”
- More accurately, I should say that information (about a piece of software) may want to be free, but developers won’t put much effort into helping it on its way.
- Ask any developer if it’s a good idea to share knowledge with their colleagues about the software that they work on, and of course they’ll agree that it is.

Information doesn’t want to be free

- But when it comes down to the nitty-gritty of sharing information, most developers are pretty bad at doing so.
- They don’t enjoy writing and maintaining documentation, and they rarely share information with colleagues through other means unless they’re prompted to do so.
- When these developers move away from the team, a huge amount of valuable information can be lost.
- What can we do to prevent this from happening?

Documentation

Technical documents can be an excellent way to pass information from developers both to contemporaneous colleagues and to future generations of maintainers. But documentation is only valuable if it is

1. Informative (that is, it doesn’t merely state what the code is doing; it tells you how and why it’s doing it)
2. Easy to write
3. Easy to find
4. Easy to read
5. Trustworthy

Documentation(example)

- Making documentation concise and putting it as close as possible to the source code (specifically, inside the same Git repository, if not embedded in the source code file itself) helps with all of these. Putting it inside the Git repository makes it easy to write and update, because developers can commit it alongside any changes they make to the code.
- This is much easier than, say, finding and updating a Word document on a network share.
- It also makes it easy for other developers to review the documents, just like they review changes to source code, which helps to keep them trustworthy.

Foster communication

- The other piece of the puzzle, namely encouraging developers to share information through means other than documentation, is more challenging.
- There are plenty of things you can try, but every team is unique.
- You’ll need to keep experimenting until you find tools that are a good fit for you and your team.

Foster communication

1. **Code reviews:** all changes to the code should be reviewed by at least one other developer.
2. **Pair programming:**
3. **Tech talks**
4. **Present your projects to other teams**
5. **Hack days:** A chance for developers to work with people from different teams, play with new technologies, and build cool stuff, preferably outside of the office environment.

Our work is never done

- Maintaining the quality of a codebase is a never-ending mission.
- You need to be constantly vigilant and tackle quality issues as they arise.
- Otherwise, they’ll quickly pile up and get out of control, and before you know it, you’ll have an unmaintainable mess of spaghetti code.
- The sooner you fix a piece of technical debt, the easier it will be.
- It’s difficult to do this work alone, so you also need to encourage a culture in which the team takes collective responsibility for the quality of the code.

Periodic code reviews

Every change to the code should be reviewed as part of the standard day-to-day development procedure. But if you only review at the level of individual changes, it’s easy to miss overarching problems

1. Ask everybody to take an hour or so to look through the code beforehand and take notes. This means people can spend the review discussing things, rather than just staring at the code in silence.

Periodic code reviews

2. Ask one person who is knowledgeable about the code to lead the review. They’ll spend a few minutes introducing the codebase and then go around the room asking people for their comments.
3. The review should take about one hour. If that’s not enough to cover the whole codebase, have multiple sessions over a number of weeks.
4. Write up a list of the review’s conclusions, divided into concrete actions and non-specific ideas or things to investigate. Share the document with the team and ask them to update it when they complete any of the actions. Check on progress weekly.

Fix one window

- The idea is that if an empty inner-city building is in good condition, people are likely to leave it alone. But as soon as it falls into disrepair, with a broken window here and there, people’s attitudes toward the building change. Vandals start to smash more windows and the disorder rapidly escalates.

Fix one window

- The analogy with software is, of course, that you need to keep on top of your codebase and keep it neat and tidy. Leave too many hacks and pain points unfixed, and the quality of the code will rapidly deteriorate. Developers will lose their respect for the code and start to get sloppy.
- Try making it a personal objective to fix one “broken window” in your codebase every couple of weeks, and make sure that these efforts are visible to other developers. Code review is a good tool for spreading the word.

Automate everything

- Throughout the book, I’ve touched on automation of various kinds, including using automated tests, automating builds, deployments, and other tasks with Jenkins, and using tools like Ansible and Vagrant to automate provisioning.
- You don’t need to automate everything at once, but every time you manage to automate something that previously depended on a human typing the right sequence of commands, or clicking a button, or holding some specialist knowledge inside their head, it’s a step in the right direction.

Automate everything

1. It makes life easier for you—Not only does it mean you don’t have to waste time performing the same tasks over and over again, it also means you have to write and maintain less documentation, and you don’t get people coming over to your desk and asking you to explain how to do stuff. And of course, it reduces the risk of your making a mistake and having to clean up after yourself.
2. It makes life easier for your successor—Automation makes it easier to pass a piece of software from one generation of developers to the next without losing any information

Write automated tests

- It goes without saying that any modern software should have some automated tests, but it’s worth pointing out that tests are particularly useful in the context of preventing legacification (let’s pretend that’s a real word) of new code.
- First, a suite of high-level tests (functional/integration/acceptance tests rather than unit tests) can act as a living specification document for the software.

Write automated tests

- Specification documents are often written at the start of a project and are rarely kept up to date as the software evolves, but automated tests are more likely to be in sync with how the software behaves.
- The reason is, of course, that if you change the behavior of the code without updating the tests to match, the tests will start to fail, and your CI server will complain.
- Second, code with automated tests is easier to maintain and therefore less likely to rot. A test suite gives developers the confidence to refactor code to keep it in shape and keep entropy at bay

Small is beautiful

- The larger a codebase gets, the more difficult it becomes to work with.
- It’s more difficult to understand how a change to one part can affect others.
- This makes it more difficult to refactor the code, meaning its quality will decrease over time.
- A large codebase is also difficult to rewrite. There’s a large psychological barrier that prevents us from throwing away a large amount of code in one go, no matter how horrible that code might be

Small is beautiful

- The key to keeping your codebase lithe and nimble and preventing it from becoming somebody’s Big Rewrite a few years down the line is simple: keep it small. Software should be designed to be disposable: make it so small that it can be thrown away and rewritten in a matter of weeks, if not days or even hours.
- It’s a little depressing for a developer to think that their lovingly crafted code will be dead and gone in a few years’ time, but it’s more sensible than keeping geriatric code on life support indefinitely just because it’s too big and important to die.

Summary

- If you want to keep your project healthy, don’t focus only on the source code. Documentation, toolchain, infrastructure, automation, and the culture of the team are all important.
- Information about your software will gradually leak away into the ether, unless you constantly guard against it.
- Good technical documentation is worth its weight in gold. And a team that communicates so well it doesn’t need documentation is even better. You need documentation to prevent knowledge being lost as team members leave, but if developers prefer asking each other questions rather than referring to the documentation, that’s a sign of a healthy team.
- Build large software out of components small enough to be discarded and rewritten without risk

The End