KDAB's Software Development Best Practices

General Development

Milian Wolff | Senior Software Engineer, KDAB

The software engineering community has learned hard lessons over time about the best way to build software. Following these tried-and-true practices provides a huge number of benefits: more resilient software, faster release schedules, higher quality products, more effective teamwork, and happier developers.

While some methodologies, workflows, and tools are specific to an individual domain, many are applicable regardless of whether you're developing a web service, embedded device, mobile app, or desktop application. Here are a few of those general best practices that we believe are widely applicable to most development projects.

1. Tools and tooling

1.1. Automation

Your tools let you rapidly create reproducible and reliable results, so whenever reasonable, automate everything.

Whether using IDE shortcuts for inserting boilerplate code, script recipes for tasks the team commonly needs, or specially configured make files, you should let your tools do the work for you.

Time spent on project tooling up front will save valuable time over the life of your project. It will also come in extremely handy whenever there are pressure cooker situations –difficult bugs, schedule overruns, or fast-approaching releases – because you can trust that all the appropriate steps to build and test the application are always being handled properly. There are many times throughout the development process when building scripts to automate a repetitive task makes a lot of sense:

- **Application control** – starting and stopping the application with or without testing, spinning up various services needed, resetting any state information so the app is always in a known state
- **Onboarding** – downloading all the tools and dependency packages a new developer needs to get their environment up and working
- **CI** – building, packaging, and testing the application on all supported platforms

- **Deployment** – pushing out a new release: versioning, labeling, license management, server uploads, etc

1.2. Source code control and branch management

The source control model we find works very well in practice for modern software is trunk-based development. We **use trunk-based development to support CI/CD workflows**.

In this method, all developers work on the trunk or main branch; it's the operational code that everyone is working on, and it's always release ready. Once we are close to a stable release, we branch off for that release. This allows the release branch to freeze all dependencies so changes no longer impact the main branch.

This said, there is more than one way to do branch management. Whether this style works for you may depend on your software structure, especially if you have certain atypical characteristics in your workflow like many large binary files or massive numbers of daily commits.

2. Develoment

2.1. Containers

We've previously written a couple of whitepapers on how containers can significantly improve your development process. We won't repeat everything here, but we recommend you download our Developer's Guide to Containers and Containers: Cloud Tech comes to Embedded for more details.

One of the ways that we use containers is to **host your build in a container**. One benefit of this is that it isolates the build environment from your development environment on developer workstations, preventing messy situations or broken or conflicting dependencies. This self-contained isolation makes it extremely easy to switch between multiple projects and to spin up as many new build instances as needed, making it a great component of a CI/CD workflow.

2.2. Dependency management

If your project is of any reasonable size, it may have dozens or even hundreds of dependencies: runtime environments and libraries you incorporate, open-source components you rely on, and developer tools that are part of your toolchain. How you manage these can be critically important to keeping your project on track. That's especially true when you have multiple CI platforms to support, since some packages may have differing versions that support different platforms. Ensuring that every developer has the right version of all dependencies can be problematic, especially since a mismatched dependency can cause very subtle problems that are very difficult to find.

Our solution to this is to **build everything from source**. We maintain one code base for all platforms, which includes all project dependencies as source. This allows us to cherry-pick which dependency fixes we need if necessary, ensuring software functionality and versions match across all platforms. We use git submodules to incorporate these separate repositories into our main repo.

Downsides to building from source

As with solutions to any complex problem, there are downsides to fully building dependencies from source. One is that every developer needs to "build the world" on their individual workstations for a fresh build. One way around this is to use a package manager approach like vcpkg or conan and distribute precompiled packages to all developer machines. These package managers are great if your dependencies are more or less stable however they impose more work (like a mini-release), making a bigger burden on the development team. For big slow-moving dependencies like the Qt library, Python, or LVVM, we see this as a good trade-off. The smaller dependencies (that is, everything else), we build from scratch. Of course, how you want to manage that trade-off depends on your particular software stack.

Another downside is that currently we need to manually manage those external dependencies to see when new patches must be integrated. This is one of the things on our list of things to automate – potentially

with dependabot.

2.3. Continuous integration (CI)

CI, when it works well, is a delight. It offers automated builds, automated testing, static analysis, and sanitizers for assessing code quality – all things that make a developer's life easier and code quality measurably better. We recommend adding **as much analysis you can to your CI process** so that your development team always has full access to the test results that could reveal issues in the code before they become problems.

However, if it takes several hours to churn through the scripts to deliver that feedback, it's not working. Slow times can break asynchronous development, fundamentally disrupting the value of a CI process. The solution here is to **have enough hardware**. If you have multiple platforms, make sure you have enough of each platform to host building and testing as well as any other platform-dependent items in your CI process. You may be able to do this by moving resources to cloud-based tools that can scale. However, NDAs, certifications, privacy regulations, or aggressive EULAs could inhibit you from taking advantage of cloud resources. In these cases, you'll need to build an on-prem solution that has enough horsepower and scalability to run all the testing and analysis you need.

2.4. Refactoring

We all know refactoring code improves its maintainability and makes it better to use. But no matter how large the refactor, we **never halt development for a rewrite**. We also don't do the refactor on a branch. The branch starts going stale almost immediately, and it requires a lot of work to keep the branch valid, many times even ending up throwing it away. Instead, we always work in the main branch.

We use an "atomic commit" principle that we adopted from Gerrit. That is, we do use branches, but only locally and each refactor is made up of smaller single-commit patch units that can go into

master right away. We also get immediate feedback from our testers when something breaks. With an "atomic commit" style of branch management, while the main branch still might get stale over time, it won't grow indefinitely as fixes pile up. Rather, more and more of it should be incorporated into master over time, so we never have to merge huge refactors in one go.

Refactors will make the code measurably better in the long run and, in our opinion, as long as you have the tests in place to verify you haven't broken anything, a refactor is always worth it. But what do you do in a large code base where this will be very painful, and you don't want to stop development? In this case, we would probably **build a parallel version of the API**, gradually cutting over the code base as the new, replicated, and refactored API gets tested out. Allowing the old and new code to co-exist is only really necessary with very large core features that are multi-developer and multi-month in scope and when necessary changes would take more than a day or two.

3. Architecture

3.1. Documentation

Everyone knows that you don't document the "what" but the "why" in your code – that's a given. But something most programmers still have trouble with is **documenting the big picture**. How are components tied together? What architectural choices were made and why – as well as what good ideas were rejected? What diagrams could save the next reader of this code hours of puzzling together the overall software structure?

Admittedly this is a life-long art to master, and we all get better at it. It often helps when you pick up old code you've written and realize you don't remember why you wrote something the way that you did. Capturing your overarching design and architectural thoughts in the comments while they're fresh is something to keep in mind when starting a new project, module, or feature. You're helping your future self!

3.2. API Stability

Any large application will be composed of many multiple different layers of libraries, connected by APIs. Should you attempt to keep these APIs stable through the life of your project? If you're able to write bug-free code that never changes and works for all anticipated use cases, then yes. But the answer for the rest of us is that you should **change your internal APIs as frequently as you need to keep** the interfaces clean, easy-to-use, and orthogonal. This contrasts with the alternative approach of keeping core APIs set in stone. In our experience, this never really works. We're always finding things we didn't anticipate in designing the original APIs and discovering ways to improve their usability, reliability, performance, or flexibility. We'd much rather periodically refactor our core APIs when required and pay the price of some momentary additional work to keep the code base in as good of shape as possible and reduce the long-term cruft that accumulates.

Where does this approach break down? As soon as those APIs are accessed by external agents. Whether that's an API you publish for third-party plug-ins or for a customer's scripting language, you should **keep external APIs frozen as long as possible**. This makes sense because every change in your external API ripples that breakage out to your customers and partners. In cases like this, we may create a separate layer in between the external API and the internal code that helps us maintain that external stability while we continue to improve the internal interfaces. That said, we keep this separate layer as small as reasonably possible and make sure it doesn't leak out any internal implementation details that could hamper our ability to change it later.

4. Testing

4.1. Build strong test scaffolding

You don't have to be doing test-driven development, but you should **learn to love your unit tests**. They are your safety net that give you the confidence to continually improve the code and to release code

that meets a high-quality bar. You should have a lot of unit tests with as large a code coverage surface as possible. Here are some of the ways we think about tests.

- If during code review you identify some code that doesn't have an associated unit test, fix it. Add tests for that software before you commit.
- As you fix bugs, add tests that validate that particular failure mode so you can test that it is fixed (and it doesn't return).
- If a bit of code isn't easily testable, that often indicates the API design can be improved as well. Write to improve the API for testability, and you'll probably improve the API for all its users.
- Tests are often thought of as a chore – the "janitorial work" of development. Don't think of it that way, because tests are all about creating solid code without cutting corners. Your software's unit tests will save your neck time and time again – they're the mark of a professional craftsperson that cares about their code.
- If software crashes, don't get defensive about it. Get curious why that crash evaded your tests, and once you find the problem, write a new test for it.
- Make test deployment easy. Every developer should be able to simply deploy tests for the part they're writing as well as the whole project and see immediately which tests fail on which platforms.

4.2. Create a crash reporting mechanism

Any top-tier software product has the ability to **capture crash reports in the field**, and your product should too. Mobile developers have this easy since the app store frameworks provide this functionality for them. Embedded and desktop developers however need to supply their own mechanism to get crash information from software running outside of the engineering lab.

You don't need a full core dump. In fact, you almost certainly don't want one – it's too large, too costly to transmit and store, and can contain private customer information that makes proper handling more complex. Instead, we use minidumps that provide the instruction pointer, registers, and stack frames for all running threads.

The minidump, along with the build version and current log files, lets us diagnose a problem without the headaches that accompany a full core dump.

A back-end system that receives, catalogs, and associates symbolic information with that software release is another necessity. We've built our own system based on software like sentry.io and Google breakpad that lets us gather crash information, automatically associates the proper symbol information, and aggregates related crashes offering more detail to the developer.

More recent solutions like crashpad aren't as good in our opinion because it's much better to have a working solution, than none at all. Breakpad is easily integrated while crashpad is that much more complex and difficult to integrate, so trying to use crashpad would mean that we wouldn't have a working soluiton for a long time. Breakpad gives us a running start and has enough to deliver what we need.

USE A MAGNET TO FIND THE NEEDLE

Crash reports can give you insights into failures that are impossible to gain any other way. We had a customer that experienced random crashes in their application as well as other system services. Our crash reports were telling us where code was failing, but the multiple failure locations didn't seem at all related. After a lengthy debugging period – including repeatedly questioning the proper functioning of the crash reporting service – we finally realized that the crash always occurred when the CPU was executing the same instruction. Investigation with the silicon vendor revealed that we had uncovered a hardware bug, and it required a CPU micro-code patch to fix it. Without the detailed information from our crash reports, this issue probably would have never been found.

4.3. Allow for in-field bug reports

Customers have a special talent for exercising edge cases and finding new ways for code to fail. But bugs are sometimes code that doesn't outright crash, however it doesn't work as the user expects. Make sure that your crash reporting mechanism can also be used if the customer's software isn't crashing but is still malfunctioning. Thread stacks and logs – as well as a customer's description – can offer a way to fix other production problems as well.

4.4. Data-driven tests

To reduce the overhead of adding new tests, you can try making some of them data driven. Whether you create your own code or use an existing tool or framework, the point is to reduce the amount of effort and specialized code to add coverage for new corner cases. It's far easier to add new lines to a data file, extra rows in a spreadsheet, or new files to a test directory that provide the input and expected result for bits of your program. While this technique is especially useful for parsers, there are lots of data manipulation and interpretation tasks that can be tested much easier with a data-driven technique, making it much easier to get comprehensive code coverage.

An additional concept to consider with data-driven testing is to use your own log files or other execution outputs as a test input "script". This does two things – it lets you easily generate new testing data, and it lets you replicate behaviors that cause crashes or problems. It requires work to set this up, but it might be only necessary for especially difficult code. For example, you could auto-extract database queries from your log to make up a new set of database tests.

5. Management

This may feel out of place in a discussion about best software engineering practices, but good management plays an important role. It is possible to build products without proper management support – but it's not easy. Make sure that your software project is **staffed with the necessary decision-making roles** that make the rest of the engineering effort go smoothly.

For example, there should be a product manager or product owner who is the final say on the product's feature set. There needs to be a project manager who is keeping track of the project's progress and can dictate where efforts should be focused when there are multiple competing priorities. There should be a release manager who sets the release date and can determine what bugs must be fixed or

features implemented in each release. If appropriate for your domain, there should be a validation manager who ensures that the project meets all appropriate criteria for certification and verification.

Having a single, definitive point of contact for these functions will prevent your software engineering project from becoming side-tracked by committee decision making, derailed by indecision, or forcing management to make calls about issues they don't fully understand. These roles do not all need to be held by separate people. In other words, one person may be the project manager as well as the release manager – or any other sensible combination of roles. However, these roles are important enough (with enough associated workload) that it is a mistake to considered them just an "extra task" for individuals with existing jobs.

What is KDAB's Software Development Best Practice series?

This series of whitepapers captures some of the hard-won experience that our senior engineering staff has developed over many years and projects. Offered up as a grab bag of techniques and approaches, we believe that these tips have helped us improve the overall development experience and quality of the resulting software. We hope they can offer the same benefits to you.

View all three parts of this whitepaper series online at: ***www.kdab.com/publications/bestpractices/***

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build run-times, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages.

www.kdab.com