WHITEPAPER

Best Practices for a Successful Development Project

In this paper, Qt Professional Services engineering teams share some of the best practices they have found over the years and continue to work by. We hope these practices will help you make confident decisions from the very first stage of your development project, keep your organization agile and effective, and avoid spending more time and effort than you need to. Although many examples within this paper reference the Qt framework, these best practices are applicable to most software projects.

Contents

Introduction........ 3
The Cost of Errors........ 4
Getting the Requirements Right........ 5
Developing for Tomorrow........ 6
Making Thoughtful Hardware Choices........ 7
Starting with Adequate Training........ 8
Practicing Continuous Integration........ 8
Aligning Teams........ 9
Avoid Reinventing the Wheel........ 10
Optimizing Performance and Refactoring........ 11
Conclusion........ 12

Introduction

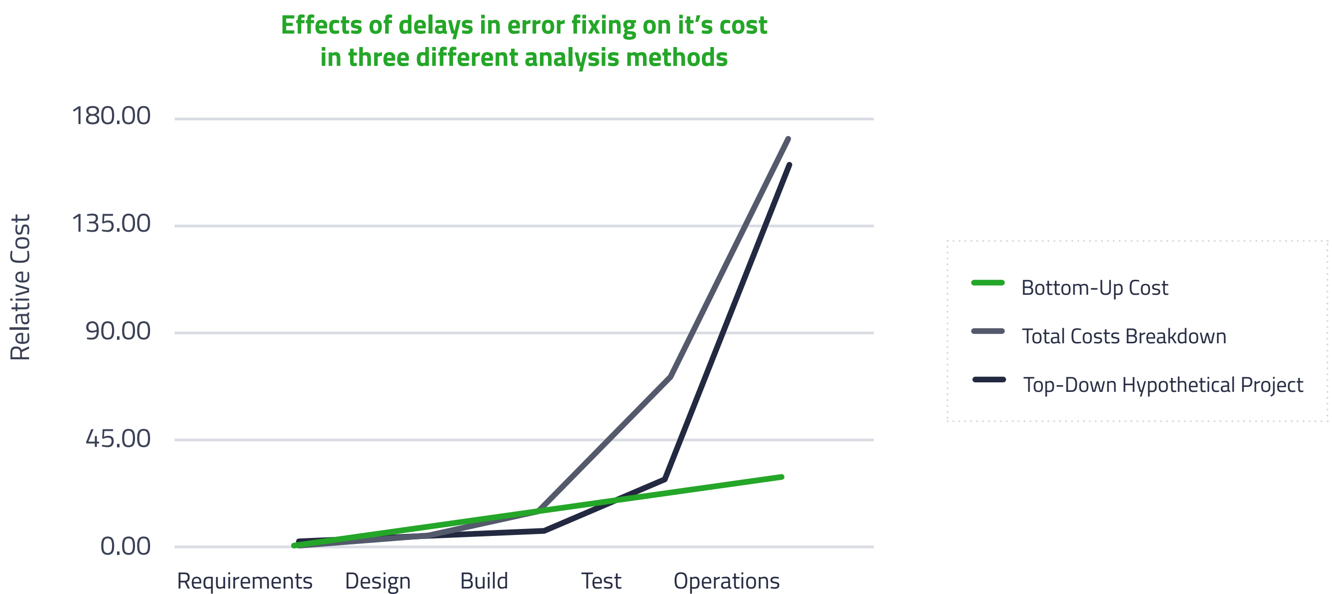
Successfully releasing a product can be a challenge even in a well-planned project. This especially applies to software development due to incorrectly scoped requirements, misaligned skillsets, and a host of other challenges.

In these situations, embracing best practices is critical to ensuring high quality and fast delivery. When consistently applied, best practices make programmers and code more agile, lead to rock-solid code that can be readily extended over time, and help organizations avoid expensive errors and delays.

The Cost of Errors

Here's something you've no doubt heard before: Errors become exponentially more costly the later in the project they are discovered – up to a factor of one hundred[1].

Teams at NASA[2] came to the same conclusion when they analyzed three development projects: a large spacecraft, a military aircraft, and a communications satellite. They used three different methods (bottom-up cost, total-cost breakdown, and top-down hypothetical) to estimate the cost of fixing errors depending on the time of discovery in each project's lifecycle. All three methods revealed a distinct relationship between the increase in cost and the project phase – some even showed cost growth to be exponential.



These results are additional proof that early error detection is key. Something important that isn't illustrated in this study is that this problem doesn't just impact R&D. A delayed market entry may drive customers to seek solutions elsewhere, which adversely affects a company's entire financial performance.

[1] Boehm, B. W. (1981). Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ.

[2] Stecklein, J. M., Dabney, J., Brandon, D, Haskins, B., Lowell, R., & Moroney, G. (2004). Error Cost Escalation Through the Project Life Cycle. Report JSC-CN-8435 (Houston, TX: NASA Johnson Space Center, 2004).

Getting the Requirements Right

Both quantity and quality of requirements directly impacts the product architecture. This, in turn, influences the costs of implementation and error correction. As we've seen time and again, fixing errors early in the design phase is a lot cheaper than writing a bunch of software workarounds for problems that shouldn't exist in the first place. While you're determining the requirements (independent of whether you're using agile, waterfall, or another methodology), it's critical to ensure that the system you are about to build will meet usability expectations. We suggest you start by conducting an analysis of end-user needs and market requirements and documenting them in detail. Once you've gathered these requirements, it's a good idea to build multiple use cases to describe each action that a user will take in the new system. If you're using agile or another iterative project management methodology, you'll need to validate your assumptions with early prototyping sessions. This will also ensure you're able to meet the product requirements within your hardware constraints. If you don't have the hardware at hand, when developing with Qt the initial stages of embedded development can be done on a desktop with the help of an emulator.

Regardless of your overall development methodology, we suggest building multiple prototypes; iterations and refinements are important ingredients for success. QML easily allows building mock-ups and prototypes. It's an important reminder that prototype code rarely embodies the high quality or proper structure needed in a finished product. Your prototypes will probably need several rounds of iterations. That way, you will likely be able to create some solid reusable components in the process, which you may be able to salvage and refactor if you follow established coding practices.

Developing for Tomorrow

It's easy to assume that code only needs to run on the platform defined in your requirements. Since code tends to live for a long time, this almost always changes. A couple of years from now, someone may want your application to run on a phone, a tablet, or some other device.

Our advice is to assume from the beginning that your application will need to run on more than one form factor, which means dealing with different screen sizes, resolutions, and aspect ratios. The best approach to this challenge is to separate the business logic from the presentation layer. This gives you the option to rewrite the UI at a later date without the risk of compromising the application's functionality. This is invaluable if you need to move it from the desktop to a touch-screen device, for example. Such a change might also require a change in the UI design approach (like moving from Qt Widgets to QML), a task that becomes much more complex if the business logic is mixed into the UI layer. For more information on how to future-proof application development based on UI scalability, check out our online documentation on scalability[3]. Another good practice is to create a consistent look and feel that can be maintained even when new platforms are introduced. A consistent visual identity goes a long way towards creating and maintaining brand recognition for your product portfolio. Yet, companies often release applications that look completely different on different devices. You can also create custom-made visual elements and even import designs from 3rd party design tools like Photoshop, Maya, MODO, and Blender.

For more information on UI design with Qt, check out our online documentation on GUI concepts[4], or talk with our engineering and productization staff[5].

Making Thoughtful Hardware Choices

Proper hardware evaluation is critical and many projects fail without it. Developers often start development on a desktop or software emulator until their embedded hardware is ready. This is a perfectly acceptable practice. However, it's very important to get your application up and running on the actual hardware or a reference board as soon as possible to verify that your hardware will properly run what you've envisioned. (A software stack like Boot2Qt[6] can help with board startup on a prototype). If using a reference board, choose one that's as close as possible to the final target hardware so that you can properly evaluate cost, performance, and resource consumption.

Neglecting to include a hardware evaluation step as early as possible in the development process is one of the main reasons that projects end up with high development costs, production delays, and slow applications. For example, if you've selected a low powered processor for cost considerations, don't expect to include lots of fancy animations or high-throughput graphics. Given enough time and energy, you can always eke out a little more performance from an underpowered chip, but there are always hard limits. If platform memory constraints are a big concern, the Qt Lite Configuration Tool[7] lets you make smaller versions of the executable by trimming out non-essential functionality, allowing you to use smaller flash and initially smaller RAM footprint. This is often likely to reduce costs within Bill of Materials (BOM). For other tips on designing for low-end hardware, visit the Gofore blog[8].

One of Qt's strengths lies in its multi-platform support, so finding suitable hardware shouldn't be a problem. Take a look at the supported device and configuration list[9] for each released Qt version. We benchmark performance on evolving and ever-shrinking hardware pieces and can advise on which hardware best suits your project in order to save you from expensive experimentation. If you don't see a platform and/or configuration of interest, talk to us[10]. We have experience in deploying and maintaining support for non-listed configurations.

Starting with Adequate Training

Qt is a versatile and flexible platform that provides many different ways to solve every problem. Properly weighing the pros and cons of each approach is essential to avoid starting off your project on the wrong foot. That's why we recommend at least minimal training on Qt basics before beginning any major endeavour. There are numerous ways to educate yourself:

- Qt Training – tailored and effective
- Community resources like forums and blogs
- High-quality docs with lots of examples
- Self-study materials

Practicing Continuous Integration

Continuous integration (CI) is a practice that merges all developer working copies into a shared mainline early and often. This prevents integration problems that are sometimes referred to as "integration hell". Whether you integrate once or several times a day, the concept is the same and can minimize development time and costs in the long run.

If your application has any performance-related requirements (such as a fast start-up or context-switching time), we suggest continuously measuring them against your target hardware. This makes changes that have a negative effect on performance much easier and quicker to spot.

You may want to consider using Qt Test, a framework for unit-testing Qt-based applications and libraries. It provides all the functionalities commonly found in unit-testing frameworks as well as extensions for testing graphical user interfaces to save you a number of headaches during integration. You can find useful tutorials on our website.

By adding a comprehensive unit-testing framework to your build process, you can block poor-quality commits as well as regularly run exhaustive configuration tests. If you further communicate the test results to your developers in a clear, concise, and user-friendly way, you'll create an efficient and responsive team, and maximize your investment in continuous integration.

Aligning Teams

Regardless of how talented your engineering teams are, one of the biggest problems we see in distributed development projects is the lack of alignment. Unless the different teams communicate effectively and consistently with each other, your application is liable to suffer from mismatched APIs, poorly defined boundary conditions, improperly used modules, and so on. We've seen numerous situations where several teams work on individual portions of a larger project within their own silos, using distinct methodologies, devices, and tools. After months of isolated development, they try to assemble their work but things don't work as planned.

To avoid this, unify distributed development teams with the same toolchain and framework components – easy to do with Qt – and reuse code across tools and features whenever possible. One effective way to do this is to create your own custom SDK based on Qt. This ensures consistency among team members and across projects.

Avoid Reinventing the Wheel

We often see developers unnecessarily writing code from scratch. Quite often, developers and entire organizations favor internally-developed solutions, even if there are suitable, ready-made, external solutions to their software development problems. This is expensive, especially compared with Qt's many ready-made and pre-tested technology components. You also get to capitalize on the expertise from the entire Qt community.

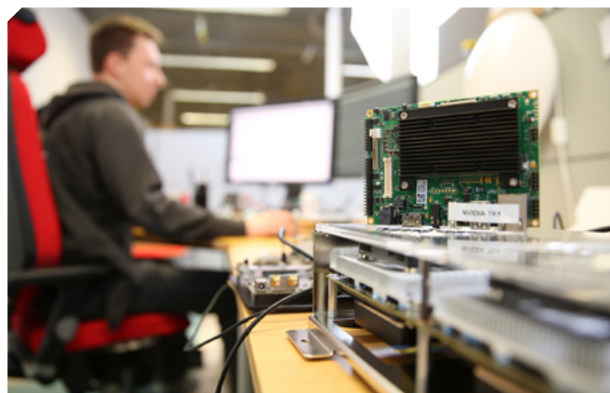
Even when you are writing new code, make sure you take advantage of all the tools Qt provides. For example, Qt has very strong cross-platform abstractions so there shouldn't be a need to write platform-, processor- or OS-specific code. If you're writing code with "#ifdef OS_LINUX" tests, something has already gone wrong.

What if you're writing code for multiple targets with different hardware, drivers, or OS support – like sound, Bluetooth, or threading primitives? Qt usually provides an abstraction layer where platform-specific solutions can be added through plugins and consistently accessed through Qt APIs. This simplifies adding support for new devices (mobile, embedded) and enables development on the desktop without the need for a simulator environment.

What you need might already exist and could save you development time and effort. If Qt doesn't yet have the feature you need, be sure to check out the Qt roadmap – it may release just in time to use in your current project.

Optimizing Performance and Refactoring

Whenever you're trying to decide whether the time is right to optimize, make sure you first consider the impact on your overall project. We've learned that a good practice in most cases is to design first, code second, and then profile/ benchmark the working code to see which parts should be optimized.



At the development stage, we're advocates of reasonable (not costly and time-consuming) performance considerations, following best practices, and lots of testing. With this in mind, don't spend your time profiling and optimizing code that may not be ultimately needed.
Ensure you have a functional working base and save the optimization iterations for later.
Generally, we recommend optimization to come last. But if you're working in an industry where boot time is a concern (such as automotive), you may want to pay special attention to boot time optimization at the outset of your project as a fast-booting system needs proper architectural design. While it's possible to make Qt-powered devices boot extremely fast with the help of good design (as well as with a number of Qt Quick tips and tricks, suitable hardware, and a lot of system image optimization), we recommend setting a target goal early, reaching it early, and keeping it through development. Check out our recommended practices and pay special attention to the Models and Views – an area where we often see quite a few problems.
We are also believers in refactoring code to increase readability and reduce complexity, which will ultimately improve maintainability and extensibility. If done properly, a regular cleaning of your code may simplify the underlying logic and eliminate unnecessary complexity, which can resolve hidden, dormant, or undiscovered bugs and vulnerabilities. Keep in mind: Refactoring doesn't have to be done entirely manually; Qt Creator offers a great way to quickly and conveniently refactor your code with computer guidance.

Conclusion

Developers are often eager to find ways to improve both the software they create and the process they use to create it. Sometimes it's through the addition of new processes and methodologies that take the guesswork out of development. Other times, it's techniques that – through daily practice – can make the act of writing software more enjoyable. But as developers, it's always helpful to better our skillset, work more effecively as a team, and quickly produce more reliable products.

At the Qt Company, we believe in continual software improvement. Hopefully you can incorporate these best practices into your personal development or your organization's process. If you find that you need a bit of help to improve your software practices – whether individually or company-wide – we're eager to share our experiences and help you along your journey.

The Qt Company develops and delivers the Qt development framework under commercial and open source licenses. We enable the reuse of software code across all operating systems, platforms and screen types, from desktops and embedded systems to wearables and mobile devices. Qt is used by approximately one million developers worldwide and is the platform of choice for in-vehicle digital cockpits, automation systems, medical devices, Digital TV/STB and other business critical applications in 70+ industries. With more than 250 employees worldwide, the company is headquartered in Espoo, Finland and is listed on Nasdaq Helsinki Stock Exchange. To learn more visit http://qt.io

© All rights reserved