The 9 Key Components to Building Software Competency

Christoph Sterz | Senior Software Engineer, KDAB

KDAB

For organizations with decades of hardware experience, the transition to software as a key differentiator can be as difficult as it is necessary. But it can also be an opportunity for growth, to maintain leadership, or to leapfrog to the front.

"We see software becoming more and more essential for creating value for our customers" comments Nico Meijerman, a chip designer who moved over to software and is now leading a team developing software competency in NTS Group, a tier-one manufacturer supplying the oil and gas industry.

Meijerman's comment neatly sums up the state of affairs for many hardware companies: Customers now require software-driven features, and their suppliers must re-orient their core competencies to deliver them – or someone else will.

Here are nine tips to help smooth the path when you bring software competency into your hardware company.

It's best to develop software competency as an integral part of your company's organization and culture. Quick-fix solutions such as outsourcing to a third party, buying a software company, or creating a separate software unit are not recommended strategies.

1. Set realistic objectives

The first step towards achieving software competency is to set achievable objectives and realistic timelines.

Remember that hardware and software engineering are different disciplines, and their practitioners think differently. Nurturing software competency in an organization that lives and breathes hardware requires a significant cultural shift.

For starters, the definitions of development and done for hardware and software are different. Hardware development is usually linear with a definitive endpoint. For example, to manufacture silicon chips, sand is made into ingots; these ingots are cut into wafers, then the wafers are etched, and so on to the final packaging, which means "done".

With software development takes multiple paths that diverge and converge, especially in organizations that have adopted continuous integration. There is no definitive endpoint that means "done", only markers for updates, which are released as needed.

As well as being realistic with your objectives, you should be flexible. As your organization adopts and adapts, you'll learn what is feasible in the short term and what must wait. You'll likely also

learn that some of your original ideas were wrong. Be ready to be surprised, and be ready to adjust your plans.

2. Find a mentor

You are venturing into uncharted territory – but it's not uncharted for everyone. We recommend you work with a mentor, someone who knows how it's done and can help orient you until you have gained the expertise and confidence to continue alone.

In particular, you may need help choosing and growing your teams. You'll be hiring software-competent staff and identifying people in your organization who are most likely to adapt early and lead others. A mentor can help find key people with the necessary traits and experience.

In short, some help from an organization with a track record helping hardware companies develop their software competency can save you time and money. We are not suggesting you outsource, however. You should aim to integrate software competency across the relevant parts of your organization, something outsourcing will not do.

Part of your transition plan should involve identifying who needs to learn what, and when. As an example, some key people transitioning into the software team would probably benefit from reading Robert C. Martin's Agile Software Development.

3. Move to an OS?

When little was required of software except, say, to respond to a few sensors and control some valves, an efficient solution was often to forgo an OS and boot directly into the application running on a microcontroller. Today, as end-users require increasingly sophisticated software functionality, an OS has become indispensable.

Without an OS, you're going to be building a lot of the system from ground zero. You have no built-in features you can leverage such as a network stack you can simply configure to implement your over-the-air updates. You must build everything yourself, which means you can offer your customers only what you can build, when you can build it.

With an OS comes support for third-party applications, and portability across numerous boards, chip sets, and architectures. For example, with a Linux or other common OS you can develop your added-value features and offer them to customers on their preferred hardware, say Arm or x86. Linux, in particular, brings

with it a huge community, hence a deep talent pool you can draw from when you are staffing.

Remember, though, that for engineers accustomed to writing everything from scratch, the shift to an OS takes adjustment. Even the most talented developers will have a learning curve to understand the power of their chosen OS - what capabilities they get for free and what they need to do themselves.

4. Modify communication

Use code repositories, bug-tracking, and code-review tools to keep communication that's important to understand through the life of a project right where developers need it.Robert C. Martin's Agile Software Development.

Meetings and telephone conversations are invaluable for working through issues and planning as they are for getting buy in or a quick answer to a difficult question. However, unless they are followed up by detailed notes, they leave no record against which you can check your understanding of what was said and decided.

Meetings and telephone calls are also almost universally hated by programmers. They interrupt their thinking and torpedo productivity. This is worth taking into account when you're building software competency, especially since many of the methods available for asynchronous communication (e.g., bug-tracking systems, Kanban boards) are also integral to the infrastructure you need to build.

Asynchronous communication interrupts no one. The recipient can ignore the message until she has finished the task at hand. The sender can get on with other work until a response comes in.

The variety of asynchronous communication tools seems to grow continuously: email, chats, document sharing, as well as code-review and bug-tracking tools.

Different tools are best for different types of communication. For example, chats tend to be best for short, time-sensitive messages: "Do you know why that new function either returns zero or crashes?"

At the other end of the spectrum, feature- and bug-tracking systems as well as code-review tools are best for more complex tasks requiring communication through the life of a project. For example, with review tools, code posted is compared to code in the repository so multiple reviewers can easily identify the changes, discuss them, request revisions, and suggest improvements.

Everyone in your organization who interacts with the software in some way – from definition to debugging – should be familiar with the bug tracking system.

5. Adopt a bug-tracking system

Bug-tracking systems warrant their own discussion, not because no software is ever bug free, but because these systems are fundamental to software development. These systems, however, are only as good as their data.

You should aim to use your bug-tracking system to provide a complete record of all that needs to be done and the progress of every item. A code optimization discussed with an engineer over lunch doesn't exist until it is recorded in the tracking system. Neither does a feature proposed by the CTO in a hallway conversation.

Properly configured and used, a bug-tracking system should follow each item from inception through design, development, testing, and release on every branch where it is implemented. Even if a bug fix is rejected, it should be in the system, with the reason for the rejection.

In our experience, it is best to bring this discipline in across your organization so that everyone interacting with the software – from definition to debugging – knows how to use the bug tracking system. There is a bit of overhead needed to do this, but it's much more efficient for managing engineering workloads and schedules, to say nothing of its benefits to management for tracking progress.

Bug-tracking systems and review tools provide detailed audit trails that are needed for safety-critical systems. Safety auditors rely on these trails when evaluating software for certification to standards such as IEC

6. Use a repository

Your repository is the foundation of your transition to software competency. A repository tool, such as git and its various cloud-based instantiations (like github or BitBucket) brings version control and a safety net to your development.

A repository manages branching, merges, and versioning, freeing developers from the minutiae of managing code iterations, revisions, variants, and so on, and allowing them to focus on design and development.

For anyone who has seen their work vanish into thin air or who has spent infuriating hours tracking down who made a code change and why, a repository is a godsend. It provides a common backup to which changes are easily committed and just as easily reversed, along with an audit trail linking changes and fixes back

to their authors' requirements and bug reports (for example, gitblame). With a repository you can try out changes, confident that you can back them out cleanly – no orphaned bits of code left behind.

A repository also fosters better collaboration. With code readily available from a single source, developers can check out updates, fixes, and experiments for their own use at whatever stage of development they happen to be. They can reuse the code in their own work, in turn checking in that work to make it available to others.

With continuous integration (CI), you don't wait for major features to be complete before bringing them back into the mainline code. Regular builds ensure that incompatibilities and other problems are identified immediately, leading to better architectural decisions early on and better-designed code.

7. Test – realistically

With a repository and a bug-tracking system in place, testing is easily integrated into the workflow with tests linked back to requirements, bug-reports, code reviews, and so on, no matter how many variants, versions, and code iterations are developed, packed, and shipped. It is important to remember that even the most sophisticated infrastructure can't resolve the continuous tension between testing feasibility and completeness.

That is, you'll need to ensure that every code change is tested, but you'll also need to remember that no code is error-free, and that it's impossible to verify every possible execution path in anything but the most trivial program.

Thus, you'll need to determine what needs to be tested and to what extent. If your system has safety-critical components, their integrity must be validated through testing and other techniques such as formal proofs. For components whose failure implies only an inconvenience, less comprehensive validation may suffice.

For example, we have seen multiple project plans that insisted on complete testing of every conceivable path through the UI. Unfortunately, such comprehensive UI testing takes a great deal of time, often for very little return, and such testing may remain incomplete for years.

The trick, of course, is to be realistic. Especially at the beginning, you may need to limit the scope of your testing to the critical components and activities, with carefully reasoned justifications for the limits you set. If it won't impact core functionality, does it really matter if controls on a rarely used configuration screen aren't perfectly left justified?

An increase in software functionality leads to increased exposure to cybersecurity risks.

8. Build and release

Builds are where everything comes together. If the infrastructure is sound, builds will be simple to run and releases easy, no matter their contents or their frequency.

Builds should be simple to configure and trigger so that in addition to scheduled builds, developers, testers, and even managers can run builds as needed. We recommend though that your organization designate a build master to ensure consistent results.

The biggest adjustment you'll need to make is to abandon the notion of a release as the delivery of a single, standard artifact. Your repository will contain all the components you'll need for your builds, which will assemble them as required to deliver diverse software images tailored to your customers' specific needs.

To someone coming from a more traditional, linear development environment workflow, agile development may at first appear chaotic. In fact, agile development requires strict discipline, clear communication, and an appropriate infrastructure – all key components of waterfall or V model development.

9. Integrate security

Increasingly, even the most basic software functionality depends on connections to the cloud. Even if your system doesn't need to be connected for day-to-day functionality, updates are now commonly distributed over-the-air.

Unfortunately, connectivity brings with it increased exposure to cyberattacks. In your transition from a hardware-centric to a software-competent organization, this fact is all too easily overlooked until late in the game.

It is important that from the start someone in your organization become reasonably competent in cybersecurity. As this is a complex and rapidly evolving domain, you'll likely want some third-party help to design things with security considerations from the beginning. But, as with the other aspects of your transition, you shouldn't simply hand things over and hope for the best. You need to become sufficiently competent to know what you want, to make sure your provider is actually delivering, and to integrate the security into your infrastructure and your products.

Conclusion

For many companies building and delivering hardware (from flow control valves to excavators), software has become an essential component of their product offerings. Thus, to meet customer demands, hardware-centric companies must add software competency to their portfolio. This requires new tools and infrastructures. It also requires planning, discipline, and perseverance – such changes aren't made overnight. Perhaps most importantly, these changes require a cultural shift, an adjustment of how things are done: from linear to non-linear, and a new definition of done: from a single, conclusive release to continuous releases.

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