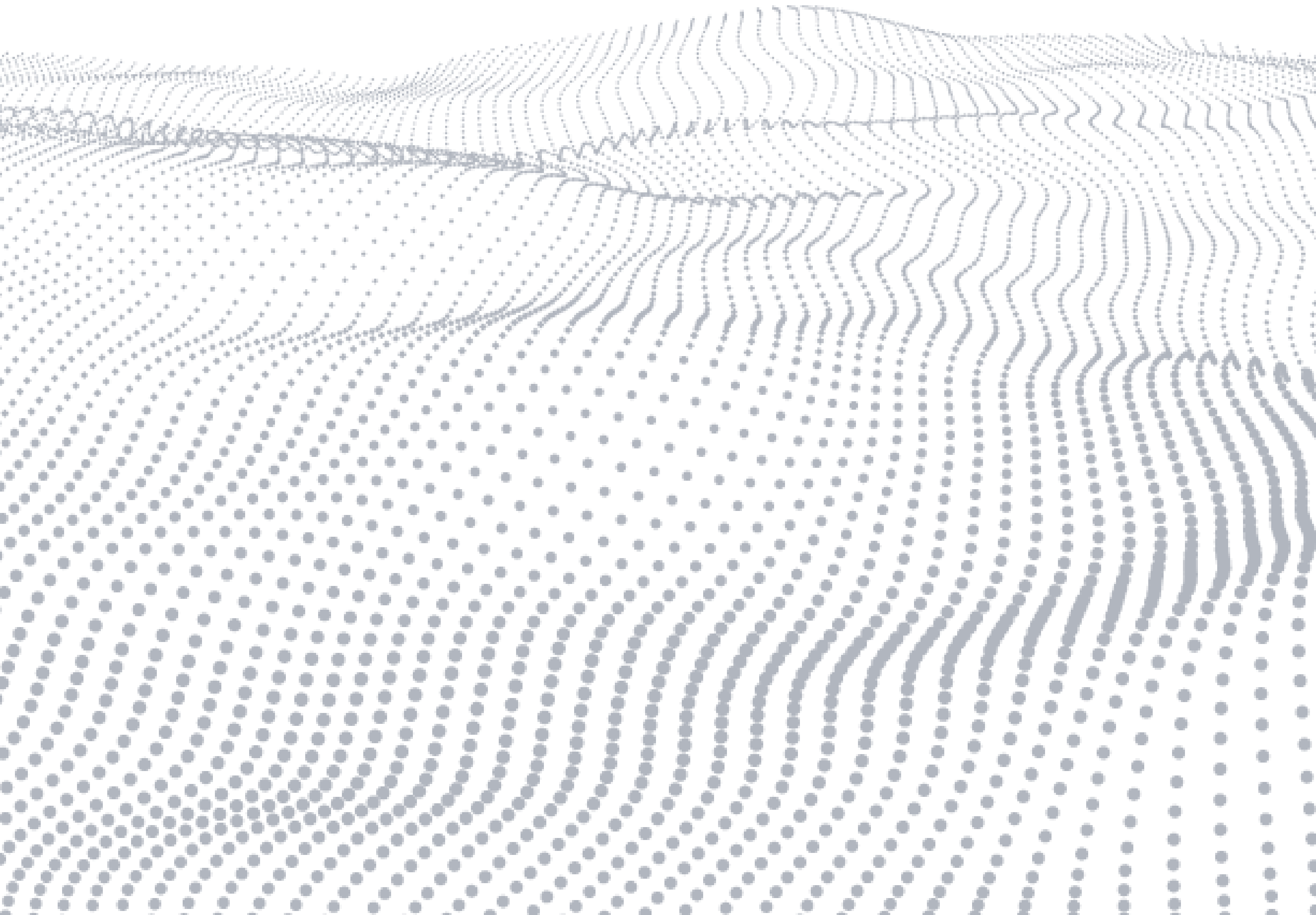
WHITE PAPER

Understanding Software Development, Technical Debt, and Risk

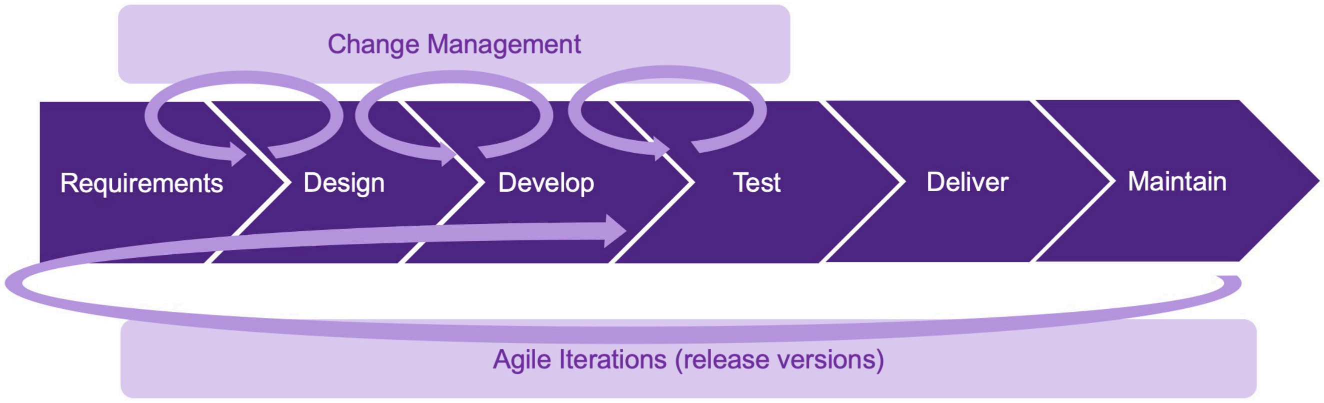
A crash course for lawyers and investors



Insight into how software is developed and what kinds of issues can lurk in a codebase can help businesspeople and lawyers better understand software risks and how to mitigate them.

Software development is a process. If the software is well-designed and developed through a process that is governed and followed, it will be of higher quality along a number of dimensions. For example, if the process is aimed at producing user-friendly software and that process followed to the letter, the result will be high usability. Apple's process includes intense user testing of software, so it is no accident that the company delivers on a great reputation for easy-to-use products.

A good process is well-defined and generally covers requirements and planning, design, development, testing, and delivery. Processes also extend into maintenance and support, and anticipate new releases in which bugs are fixed and features added. While the process looks linear on paper, new releases start back at the beginning, so it's iterative. And in fact, most of the steps in the process are iterative in themselves.



In its early days, software was built from scratch. Today, most of what developers do is assemble already-built components into a whole. With the complexity of modern software, developing it all from a blank screen would be nearly impossible. Computer hardware went through this same evolution. Early machines were developed transistor by transistor. Today, Black Duck® builds tools for hardware development supporting chips that have billions of transistors, all buried in sophisticated components that developers assemble in clever ways. Similarly, software developers may rely on parts they or their colleagues built, but they rely even more on third-party components. Today, it's the norm for 80% of a proprietary application to be built from open source components that are freely available on the internet.

Of course, software development groups always *intend* to produce quality software, but that's not always the result. A well-designed process should produce good software. But not all development processes are well-designed or well-defined, particularly in less mature organizations. Often, a development methodology that worked to get a small development team going in a startup becomes insufficient as the organization grows. And good processes only ensure quality software if they are followed. It's surprisingly easy for development to go astray.

Time or budget pressure from the business (perhaps due to poor planning) can push developers to take shortcuts. The "iron triangle" of project management is *scope, schedule, cost—pick any two.* Shortcutting may let developers pick all three, but at the expense of quality.

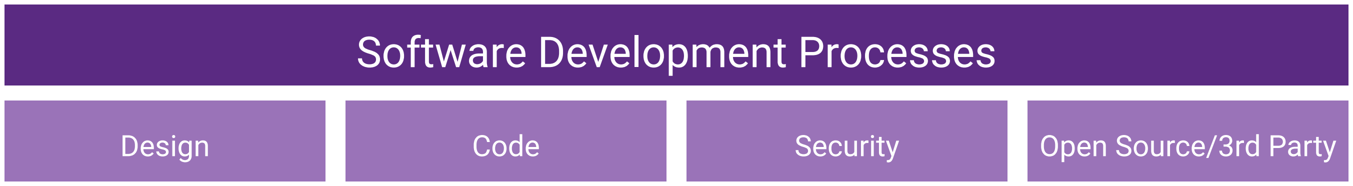
Shortcuts will seem perfectly rational at the time and may not have significant short-term consequences, but they will create problems that must be fixed later, thus compromising *future* scope, schedule, and cost.

A real-world analogy: A condo owner does a quick fix on a water leak plaguing her partially below-grade unit. But a year later, she has to call in a mold specialist. Ultimately, her shortcut has let the water issue fester and means she has to take down all the walls a second time so a proper job can be done. And any money she "saves" initially results in a considerably higher price later.

The same is true in software development; quick, short-term "solutions" lead to longer-term problems. Taking shortcuts can be justified, however, a well-disciplined organization will do so deliberately and have a process for tracking and addressing them. But too often, the pressures that drove a team in this release will continue into the next, and thus shortcuts get piled on top of shortcuts. Such quick fixes almost always require future work and mean the team is essentially "borrowing" from future capacity to get the software shipped today. That borrowing is why an accumulation of issues in software is referred to as "technical debt."

All organizations carry technical debt, but just as skyrocketing credit balances can ruin family finances, so can too much technical debt put a huge drag on a development organization. But while a credit card balance is clear and visible, technical debt is much harder to see. That's because it's the sum of the shortcuts taken by many developers, and each one may be known only to the individual who took it—if those developers even recognize what they've done. Only an organization that proactively manages such issues knows the extent.

There are many kinds of shortcuts and they can manifest as different types of problem in the code. The issues generally fall into one or more of these categories: design quality, code quality, security, and open source and third-party content.



The problems that fall into these categories may be the result of shortcuts, but they can also be a function of the developers simply not knowing how to address the issues up front. Training in these areas is essential, but knowledge and good intentions can be overridden by the verities of schedule pressure.

Design quality

Best practices for development require defining an appropriate architecture that determines how functionality will be segmented and how the functions will interact. A good design will have clearly defined interfaces between functions, and no one function will be too big for one person to understand. If a function is too big, it should be broken down into smaller pieces in a hierarchy. And functions should not be duplicated, so if there's a fix required it only has to be implemented in one place.

A nicely hierarchical and modular design will be easy to maintain because it allows a developer to work on one piece of the code without breaking anything else. You may have heard problematic code referred to as "spaghetti," meaning everything is connected to everything else. The problem with that is if a developer wants to add a function, they can't easily anticipate the effects and problems that their code changes will create. In such "brittle" code, it's possible that 70% of development effort goes into fixing things developers break and only 30% of developers' time is actually adding new functionality.

How does a codebase get into this state? Again, deficient processes and shortcuts are the problem. Often, software starts out aligning with the initial architectural diagram, but as a codebase ages it often deviates from the original design. A disciplined process will enforce standards that maintain clear relationships in a coded hierarchy and modularity. It also defines separation of functional concerns, with maximum limits on the size of functional pieces.

But imagine a developer in a hurry to add some functionality that bloats a particular file or adding that functionality to the wrong level in the hierarchy. Best practice would be a simple "refactoring"—breaking the function into "bite-size" pieces and moving them to appropriate locations in the code stack, but that will take some time. Business pressures can cause a developer to opt to move on to the next task and worry about this later, and maybe leave an "I need to fix this" note in the code. Now imagine 100 developers all doing the same thing over several releases and suddenly… fettucine without the tasty sauce.

Code quality

In buggy code, the software doesn't do what it was designed to do. It might misbehave, have a weird result, or simply crash. There are other aspects to code quality, but bugginess is a simple way to think about the concept. Another aspect of code quality is code maintainability. Is the software written in a way that makes it easy to maintain? Coding standards and inline documentation notes are supposed to make it practical for someone other than the author, who might not still be involved when the code needs a fix, to figure out what's going on in code and thereby be able to fix a problem or add a feature.

Here too, shortcuts are often a culprit. Circumventing coding standards or not bothering to write the needed documentation are quick ways to move on to the next task, but failure to adhere to such standards can create a future mess. And it might not be just a single developer taking shortcuts. Looming deadlines can push against the discipline of performing a full QA cycle, the last line of defense against bugs. The further bugs make it through the process, the more "expensive" they are. Worst are bugs that impact customers; addressing them will surely disrupt future development cycles and "cost" the organization future capacity.

Also, the rush of a deadline may cause a developer get careless and introduce bugs in the software. One very common type is a copy/paste bug—something lawyers might relate to. Software developers, like lawyers engage in a lot of reuse. Picture a lawyer copying a few paragraphs from one contract to another. In the first contract, a party might have been referred to as "Customer" while in the second, it's "Client." The lawyer has to copy and paste the text, and then correct the term in the pasted code. In a rush, the lawyer might overlook instances and end up with a mismatch in language. Software developers often do exactly the same thing with, for example, the names of variables, and end up with a bug that could crash the application or cause other erroneous behavior.

Security

Cybersecurity is a big topic today. With software such an important part of every company's operations, software applications and systems must be "hardened" again hackers' attempts to steal or change data or disrupt operations.

Compare this to physical security. A software firewall is like a fence around a property, a first line of defense. The house is the application—it needs to be accessible to the folks who live there, so doors and windows are necessary. But those access points are potential weaknesses, so they need to be designed in a way to hold off burglars, with proper locks and such. The builder must put a lock on each and install them all properly. If a door purchased from a third party comes with a lock, the builder needs to ensure it's a good one.

Similarly, software access points for users represent weaknesses, so those access points need to be securely designed. Developers need to properly implement security controls and ensure that they don't introduce vulnerabilities along the way. If a developer is in a rush or not trained in security, they might not design the proper controls or implement them properly. There might not be a process for vetting third-party open source components for vulnerabilities, or the developer may circumvent the process. Any of these shortfalls can lead to vulnerabilities in the code.

For example, a common type of vulnerability is SQL injection. When you fill out a webform with your name, it gets passed to a database. Databases can be modified using SQL, a standard language. Hackers figured out that if, instead of entering a name, they enter a long, cleverly formed string of characters representing commands from the SQL language, they can take control of, modify, and extract data from the database. Secure coding standards avoid this vulnerability by writing simple code to check that the user-provided text looks like a name—it is not absurdly long, and contains only alphabetic characters—before passing it to the database. A super easy fix, but it requires developers to take the time to follow secure coding standards. Many companies are not great at enforcing a process to ensure this.

Open source and third-party content

Developers heavily leverage third-party code when assembling software. Most of that is open source. Research shows that 80% of code in a typical codebase is open source, and most applications include hundreds of open source components. Developers can get their job done quickly by getting much of the functionality they need from the open source components freely available on the internet.

This makes perfect sense, and companies need to lean on open source to be competitive, but they also need to recognize that not all open source is created equal. Ideally, developers will select popular components with active communities supporting them under appropriate software licenses, and components free from vulnerabilities that could compromise security. But with millions of open source components out there, it's easy to adopt components that have been abandoned by their community or that have licenses that are problematic for commercial use. The security of these components is even trickier given that 10 to 15 new vulnerabilities in open source are discovered daily. That means adopting an open source component requires monitoring them over time and keeping up with new versions and patches. This is all doable, but with companies typically using hundreds or thousands of components, keeping up with new patches and versions is not trivial. A development organization needs to be well-organized and deliberate to stay on top of the latest.

Best practices dictate that a company should have a top-down strategy for using open source, well-documented and communicated policies to guide developers in their decision-making, and processes that the team is trained on and that embed the policies and technology into the entire development tool chain. This will help take the friction out of the system and ensure developer productivity along with compliance. And all those systems need to be continually monitored.

Sophisticated companies, including some small ones, may handle all this reasonably well. But the majority of companies don't have the infrastructure in place to maintain an accurate software Bill of Materials for their applications—table stakes for being able to assess and manage risks.

Managing risks in software

Companies dependent on software (and who isn't these days?) need to manage all these issues in the software they develop. That means creating well-defined and documented processes, using software tools that help guide the process, and ensuring that developers are trained to understand the processes and stick with them. Additionally, sticking with the program requires organizational commitment and discipline to minimize the need for shortcuts and to track and address them before they build up into an overwhelming pile of debt.

Many companies don't have this all down, particularly smaller companies that might be targets in an acquisition. So in acquisition, it's important to evaluate a target company's process and as well as its code. Evaluating the process typically involves interviews with technologists about how things really work across the software development process. In theory, this should correlate to the state of the code. For example, a solid QA process should lead to code with low bug density. However, experience shows that's often not the case. The QA process might look good on paper, but it's useless if it's not followed in practice. It's also good to know if a process was put in place relatively recently, so code circa six months ago didn't enjoy the same scrutiny. Or maybe the process is usually followed but business pressures cause developers to resort to shortcuts. There are good reasons to complement an interview-based due diligence process with expert analysis of the code, assisted by sophisticated tools that assess its design quality, code quality, security, and open source content.

If you are involved in or advising on a transaction, insight into how software is developed and the potential problems that can result will help you better plan software due diligence. And that can help ensure that the definitive agreement, business case, and integration plans take proper account of the problems that will inevitably arise.

About Black Duck

Black Duck® offers the most comprehensive, powerful, and trusted portfolio of application security solutions in the industry. We have an unmatched track record of helping organizations around the world secure their software quickly, integrate security efficiently in their development environments, and safely innovate with new technologies. As the recognized leaders, experts, and innovators in software security, Black Duck has everything you need to build trust in your software. Learn more at www.blackduck.com.

©2024 Black Duck Software, Inc. All rights reserved. Black Duck is a trademark of Black Duck Software, Inc. in the United States and other countries. All other names mentioned herein are trademarks or registered trademarks of their respective owners. August 2024