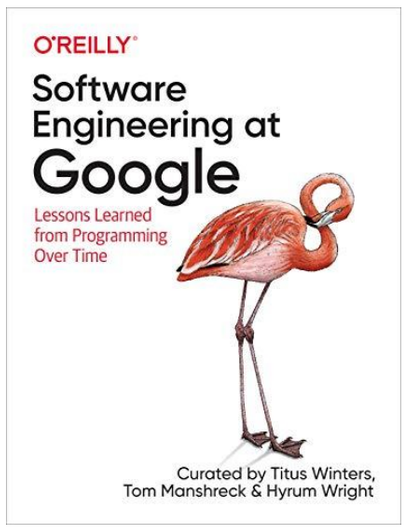

kindle

Software Engineering at Google: Lessons Learned from Programming Over Time (English Edition)

de Winters, Titus, Manshreck, Tom, Wright, Hyrum

Prévisualisation rapide Kindle gratuite : https://read.amazon.com/kp/kshare?asin=B0859PF5HB

237 surlignements | Jaune (237)

Page 4

Surlignement (Jaune) | Page 4

> The great thing about tech is that there is never only one way to do something. Instead, there is a series of trade-offs we all must make depending on the circumstances of our team and situation.

Page 18

Surlignement (Jaune) | Page 18

> Within Google, we sometimes say, “Software engineering is programming integrated over time.”

Surlignement (Jaune) | Page 18

> we might need to delineate between programming tasks (development) and software engineering tasks (development, modification, maintenance).

Page 19

Surlignement (Jaune) | Page 19

> Your project is sustainable if, for the expected life span of your software, you are capable of reacting to whatever valuable change comes along, for either technical or business reasons.

Page 24

Surlignement (Jaune) | Page 24

> So, concretely, how does short-term programming differ from producing code with a much longer expected life span? Over time, we need to be much more aware of the difference between “happens to work” and “is maintainable.” There is no perfect solution for identifying these issues. That is unfortunate, because keeping software maintainable for the long-term is a constant battle.

Page 25

Surlignement (Jaune) | Page 25

> Hyrum's Law represents the practical knowledge that — even with the best of intentions, the best engineers, and solid practices for code review — we cannot assume perfect adherence to published contracts or best practices.

Page 28

Surlignement (Jaune) | Page 28

> We've taken to saying, "It's programming if 'clever' is a compliment, but it's software engineering if 'clever' is an accusation."

Page 34

Surlignement (Jaune) | Page 34

> "If a product experiences outages or other problems as a result of infrastructure changes, but the issue wasn't surfaced by tests in our Continuous

Surlignement (Jaune) | Page 34

> Integration (CI) system, it is not the fault of the infrastructure change." More colloquially, this is phrased as "If you liked it, you should have put a CI test on it," which we call "The Beyoncé Rule."13

Page 37

Surlignement (Jaune) | Page 37

> Stagnation is an option, but often not a wise one.

Surlignement (Jaune) | Page 37

> One of the broad truths we've seen to be true is the idea that finding problems earlier in the developer workflow usually reduces costs.

Page 39

Surlignement (Jaune) | Page 39

> It is important for there to be a decider for any topic and clear escalation paths when decisions seem to be wrong, but the goal is consensus, not unanimity.

Surlignement (Jaune) | Page 39

> It’s fine and expected to see some instances of “I don’t agree with your metrics/valuation, but I see how you can come to that conclusion.”

Page 49

Surlignement (Jaune) | Page 49

> Software is sustainable when, for the expected life span of the code, we are capable of responding to changes in dependencies, technology, or product requirements. We may choose to not change things, but we need to be capable.

Page 50

Surlignement (Jaune) | Page 50

> Being data driven is a good start, but in reality, most decisions are based on a mix of data, assumption, precedent, and argument. It’s best when objective data makes up the majority of those inputs, but it can rarely be all of them.

Page 57

Surlignement (Jaune) | Page 57

> Being a genius is most definitely not an excuse for being a jerk: anyone — genius or not — with poor social skills tends to be a poor teammate.

Page 62

Surlignement (Jaune) | Page 62

> “Many eyes make sure your project stays relevant and on track.”

Page 70

Surlignement (Jaune) | Page 70

> better way to say the same thing might be, “Hey, I’m confused by the control flow in this section here. I wonder if the xyzzy code pattern might make this clearer and easier to maintain?”

Page 79

Surlignement (Jaune) | Page 79

> Sharing expertise across an organization is not an easy task. Without a strong culture of learning, challenges can emerge.

Page 80

Surlignement (Jaune) | Page 80

> SPOFs can arise out of good intentions: it can be easy to fall into a habit of “Let me take care of that for you.” But this approach optimizes for short-term efficiency (“It’s faster for me to do it”) at the cost of poor long-term scalability (the team never learns how to do whatever it is that needs to be done). This mindset also tends to lead to all-or-nothing expertise.

Page 81

Surlignement (Jaune) | Page 81

> In this scenario, knowledge and responsibilities continue to accumulate on those who already have expertise, and new team members or novices are left to fend for themselves and ramp up more slowly.

Surlignement (Jaune) | Page 81

> Mimicry without understanding.

Surlignement (Jaune) | Page 81

> haunted graveyards are characterized by people avoiding action because of fear and superstition.

Page 82

Surlignement (Jaune) | Page 82

> Personalized, one-to-one advice from an expert is always invaluable.

Surlignement (Jaune) | Page 82

> Documented knowledge, on the other hand, can better scale not just to the team but to the entire organization.

Surlignement (Jaune) | Page 82

> that scalability comes with some trade-offs: it might be more generalized and less applicable to individual learners’ situations, and it comes with the added maintenance cost required to keep information relevant and up to date over time.

Page 86

Surlignement (Jaune) | Page 86

> If you take away only a single thing from this chapter, it is this: always be learning; always be asking questions.

Page 87

Surlignement (Jaune) | Page 87

> In fact, the more you know, the more you know you don’t know. Openly

Page 88

Surlignement (Jaune) | Page 88

> engineers have a tendency to reach for “this is bad!” far more quickly than is often warranted, especially for unfamiliar code, languages, or paradigms.

Page 99

Surlignement (Jaune) | Page 99

> In the worst cases, the group reduces to its most toxic members.

Page 100

Surlignement (Jaune) | Page 100

> but being an expert and being kind are not mutually exclusive.

Page 142

Surlignement (Jaune) | Page 142

> Traditional managers worry about how to get things done, whereas great managers worry about what things get done (and trust their team to figure out how to do it).

Page 146

Surlignement (Jaune) | Page 146

> “Hope is not a strategy.”

Page 150

Surlignement (Jaune) | Page 150

> Steve Jobs once said: “A people hire other A people; B people hire C people.”

Page 170

Surlignement (Jaune) | Page 170

> the way to make people the happiest and most productive isn’t to motivate them extrinsically (e.g., throw piles of cash at them); rather, you need to work to increase their intrinsic motivation.

Surlignement (Jaune) | Page 170

> you can increase intrinsic motivation by giving people three things: autonomy, mastery, and purpose.

Page 186

Surlignement (Jaune) | Page 186

> “I can see the forest through the trees.” In other words, you can define a high-level strategy. Your strategy needs to cover not just overall technical direction, but an organizational strategy as well. You’re building a blueprint for how the ambiguous problem is solved and how your organization can manage the problem over time. You’re continuously mapping out the forest, and then assigning the tree-cutting to others.

Surlignement (Jaune) | Page 186

> The book Debugging Teams2 has

Page 187

Surlignement (Jaune) | Page 187

> This is what good management is about: 95% observation and listening, and 5% making critical adjustments in just the right place.

Surlignement (Jaune) | Page 187

> your “customers” are not end users out in the world, but your coworkers.

Page 190

Surlignement (Jaune) | Page 190

> Somehow you need to solve both problems now, which likely means that the original problem still needs to be managed with half as many people in half the time. You need the other half of your people to tackle the new work! We refer to this final step as the compression stage: you're taking everything you' ve been doing and compressing it down to half the size.

Surlignement (Jaune) | Page 190

> Larry Page, one of Google's founders, would probably refer to this spiral as "uncomfortably exciting."

Page 192

Surlignement (Jaune) | Page 192

> Remember that your job as a leader is to do things that only you can do, like mapping a path through the forest.

Page 193

Surlignement (Jaune) | Page 193

> Regularly block out two hours or more to sit quietly and work only on important-but-not-urgent things — things like team strategy, career paths for your leaders, or how you plan to collaborate with neighboring teams.

Page 194

Surlignement (Jaune) | Page 194

> But as a leader of leaders, your time and attention are under constant attack. No matter how much you try to avoid it, you end up dropping balls on the floor — there are just too many of them being thrown at you. It's overwhelming, and you probably feel guilty about this all the time.

Page 195

Surlignement (Jaune) | Page 195

> Instead, mindfully identify the balls that strictly fall in the top 20% — critical things that only you can do — and focus strictly on them. Give yourself explicit permission to drop the other 80%.

Page 196

Surlignement (Jaune) | Page 196

> Typically, this means being aware of how much energy you have at any given moment, and making deliberate choices to “recharge”

Page 198

Surlignement (Jaune) | Page 198

> Better to get nothing done that day than to do active damage.

Page 213

Surlignement (Jaune) | Page 213

> To help people remember all five components, we use the mnemonic “QUANTS”:

Page 216

Surlignement (Jaune) | Page 216

> Consider, for example, measuring code quality. Although academic literature has proposed many proxies for code quality, none of them have truly captured

Page 218

Surlignement (Jaune) | Page 218

> Table 7-2. Goals, signals, and metrics

Page 223

Surlignement (Jaune) | Page 223

> Before measuring productivity, ask whether the result is actionable, regardless of whether the result is positive or negative. If you can’t do anything with the result, it is likely not worth measuring.

Page 231

Surlignement (Jaune) | Page 231

> overhead. If just one or two engineers are getting something wrong, adding to everyone’s mental load by creating new rules doesn’t scale.

Surlignement (Jaune) | Page 231

> Python style guide, when discussing conditional expressions, we recognize that they are shorter than if statements and therefore more convenient for code authors. However, because they tend to be more difficult for readers to understand than the more verbose if statements, we restrict their usage. We value “simple to read” over “simple to write.”

Page 232

Surlignement (Jaune) | Page 232

> example, our Java, JavaScript, and C++ style guides mandate use of the override annotation or keyword whenever a method overrides a superclass method.

Page 235

Surlignement (Jaune) | Page 235

> Consistency enables scaling. Tooling is key for an organization to scale, and consistent code makes it easier to build tools that can understand, edit, and generate code.

Page 241

Surlignement (Jaune) | Page 241

> When necessary, we permit concessions to optimizations and practicalities that might otherwise conflict with our rules.

Surlignement (Jaune) | Page 241

> Consistency is vital; adaptation is key.

Page 253

Surlignement (Jaune) | Page 253

> Automated rule enforcement ensures that rules are not dropped or forgotten as time passes or as an organization scales

Page 254

Surlignement (Jaune) | Page 254

> Some technical rules explicitly call for human judgment. In the C++ style guide, for example: “Avoid complicated template metaprogramming.” “Use auto to avoid type names that are noisy, obvious, or unimportant

Page 267

Surlignement (Jaune) | Page 267

> The primary reviewer can focus on code correctness and the general validity of the code change; the code owner can focus on whether this change is appropriate for their part of the codebase without having to focus on the details of each line of code.

Page 268

Surlignement (Jaune) | Page 268

> They are more concerned with questions such as: “Will this code be easy or difficult to maintain?” “Does it add to my technical debt?” “Do we have the expertise to maintain it within our team?”

Page 280

Surlignement (Jaune) | Page 280

> Remember that you are not your code, and that this change you propose is not “yours” but the team’s.

Surlignement (Jaune) | Page 280

> Remember that part of the responsibility of an author is to make sure this code is understandable and maintainable for the future.

Page 283

Surlignement (Jaune) | Page 283

> A code review is not just something that you do in the present time; it is something you do to record what you did for posterity.

Page 296

Surlignement (Jaune) | Page 296

> Each language is a tool in the toolbox. Documentation should be no different: it’s a tool, written in a different language (usually English) to accomplish a particular task. Writing documentation is not much different than writing code.

Surlignement (Jaune) | Page 296

> Like code, documents should also have owners. Documents without owners become stale and difficult to maintain. Clear ownership also makes it easier to handle documentation through existing developer workflows: bug tracking systems, code review tooling, and so forth.

Page 297

Surlignement (Jaune) | Page 297

> Documentation is often so tightly coupled to code that it should, as much as possible, be treated as code. That is, your documentation should: Have internal policies or rules to be followed Be placed under source control Have clear ownership responsible for maintaining the docs Undergo reviews for changes (and change with the code it documents) Have issues tracked, as bugs are tracked in code Be periodically evaluated (tested, in some respect) If possible, be measured for aspects such as accuracy, freshness, etc. (tools have still not caught up here)

Page 298

Surlignement (Jaune) | Page 298

> Case Study: The Google Wiki

Surlignement (Jaune) | Page 298

> Because there were no true owners for documents, many became obsolete.3

Surlignement (Jaune) | Page 298

> Another problem with GooWiki became apparent over time: the people who could fix the documents were not the people who used them.

Page 300

Surlignement (Jaune) | Page 300

> Instead, before you begin writing, you should (formally or informally) identify the audience(s) your documents need to satisfy. A design document might need to persuade decision makers. A tutorial might need to provide very explicit instructions to someone utterly unfamiliar with your codebase. An API might need to provide complete and accurate reference information for any users of that API, be they experts or novices. Always try to identify a primary audience and write to that audience.

Page 301

Surlignement (Jaune) | Page 301

> Obviously, writing such documents is a balancing act and there’s no silver bullet, but one thing we’ve found is that it helps to keep your documents short. Write descriptively enough to explain complex topics to people unfamiliar with the topic, but don’t lose or annoy experts. Writing a short document often requires you to write a longer one (getting all the information down) and then doing an edit pass, removing duplicate information where you can.

Surlignement (Jaune) | Page 301

> As Blaise Pascal once said, “If I had more time, I would have written you a shorter letter.”

Page 302

Surlignement (Jaune) | Page 302

> between that of a customer (e.g., a user of an API) and that of a provider (e.g., a member of the project team). As much as possible, documents intended for one should be kept apart from documents intended for the other. Implementation details are important to a team member for maintenance purposes; end users should not need to read such information.

Page 303

Surlignement (Jaune) | Page 303

> Such documents fail because they don’t serve a single purpose (and they also get so long that no one will read them; some notorious wiki pages scrolled through several dozens of screens). Instead, make sure your document has a singular purpose, and if adding something to that page doesn’t make sense, you probably want to find, or even create, another document for that purpose.

Page 304

Surlignement (Jaune) | Page 304

> Code comments are the most common form of reference documentation that an engineer must maintain. Such comments can be divided into two basic camps: API comments versus implementation comments. Remember the audience differences between these two:

Surlignement (Jaune) | Page 304

> Most reference documentation, even when provided as separate documentation from the code, is generated from comments within the codebase itself.

Page 309

Surlignement (Jaune) | Page 309

> The canonical design document templates at Google require engineers to consider aspects of their design such as security implications, internationalization, storage requirements and privacy concerns, and so on. In most cases, such parts of those design documents are reviewed by experts in those domains.

Surlignement (Jaune) | Page 309

> good design document should cover the goals of the design, its implementation strategy, and propose key design decisions with an emphasis on their individual trade-offs. The best design documents suggest design goals and cover alternative designs, denoting their strong and weak points.

Page 310

Surlignement (Jaune) | Page 310

> Most tutorials require you to perform a number of steps, in order. In those cases, number those steps explicitly. If the focus of the tutorial is on the user (say, for external developer documentation), then number each action that a user needs to undertake. Don’t number actions that the system may take in response to such user actions. It is critical and important to number explicitly every step when doing this. Nothing is more annoying than an error on step 4 because you forget to tell

Page 311

Surlignement (Jaune) | Page 311

> Example: A bad tutorial made better

Page 312

Surlignement (Jaune) | Page 312

> Note how each step requires specific user intervention.

Surlignement (Jaune) | Page 312

> In almost all cases, a conceptual document is meant to augment, not replace, a reference documentation set. Often this leads to duplication of some information, but with a purpose: to promote clarity. In those cases, it is not necessary for a conceptual document to cover all edge cases (though a reference should cover those cases religiously). In this case, sacrificing some accuracy is acceptable for clarity. The main point of a conceptual document is to impart understanding.

Page 313

Surlignement (Jaune) | Page 313

> “Concept” documents are the most difficult forms of documentation to write.

Surlignement (Jaune) | Page 313

> comments are the unit tests of documentation, conceptual documents are the integration tests.

Surlignement (Jaune) | Page 313

> concept document needs to be useful to a broad audience: both experts and novices alike. Moreover, it needs to emphasize clarity, so it often needs to sacrifice completeness (something best reserved for a reference) and (sometimes) strict accuracy.

Page 314

Surlignement (Jaune) | Page 314

> Most engineers are members of a team, and most teams have a “team page” somewhere on their company’s intranet.

Surlignement (Jaune) | Page 314

> ensure that a landing page clearly identifies its purpose, and then include only links to other pages for more information.

Surlignement (Jaune) | Page 314

> Most poorly configured landing pages serve two different purposes: they are the “goto” page for someone who is a user of your product or API, or they are the home page for a team.

Page 319

Surlignement (Jaune) | Page 319

> Traditionally, the first section denotes the problem, the middle section goes through the recommended solutions, and the conclusion summarizes the takeaways.

Surlignement (Jaune) | Page 319

> Most engineers loathe redundancy, and with good reason. But in documentation, redundancy is often useful.

Page 320

Surlignement (Jaune) | Page 320

> In each case, a “good document” is defined as the document that is doing its intended job. As a result, you rarely want a document doing more than one job.

Page 321

Surlignement (Jaune) | Page 321

> Over time, documents become stale, obsolete, or (often) abandoned. Try as much as possible to avoid abandoned documents, but when a document no longer serves any purpose, either remove it or identify it as obsolete (and, if available, indicate where to go for new information). Even for unowned documents, someone adding a note that “This no longer works!” is more helpful than saying nothing and leaving something that seems authoritative but no longer works.

Page 324

Surlignement (Jaune) | Page 324

> change the quality of engineering documentation, engineers — and the entire engineering organization — need to accept that they are both the problem and the solution. Rather than throw up their hands at the state of documentation, they need to realize that producing quality documentation is part of their job and saves them time and effort in the long run.

Page 327

Surlignement (Jaune) | Page 327

> The act of writing tests also improves the design of your systems. As the first clients of your code, a test can tell you much about your design choices.

Page 328

Surlignement (Jaune) | Page 328

> A bad test suite can be worse than no test suite at all.

Surlignement (Jaune) | Page 328

> At Google, we have determined that testing cannot be an afterthought. Focusing on quality and testing is part of how we do our jobs. We have learned, sometimes painfully, that failing to build quality into our products and services inevitably leads to bad outcomes. As a result, we have built testing into the heart of our engineering culture.

Page 331

Surlignement (Jaune) | Page 331

> When it comes to testing, there is one clear answer: automation.

Page 333

Surlignement (Jaune) | Page 333

> Even in companies where QA is a prominent organization, developer-written tests are commonplace.

Page 334

Surlignement (Jaune) | Page 334

> Of course, writing tests is different from writing good tests.

Page 335

Surlignement (Jaune) | Page 335

> After all, the act of writing tests can take just as long (if not longer!) than implementing a feature would take in the first place. On the contrary, at Google, we've found that investing in software tests provides several key benefits to developer productivity:

Page 336

Surlignement (Jaune) | Page 336

> Note that tests work best as documentation only if care is taken to keep them clear and concise.

Page 340

Surlignement (Jaune) | Page 340

> This means that small tests aren't allowed to access the network or disk. Testing code that relies on these sorts of operations requires the use of test doubles (see Chapter 13) to replace the heavyweight dependency with a lightweight, in-process dependency.

Page 343

Surlignement (Jaune) | Page 343

> In some cases, you can limit the impact of flaky tests by automatically rerunning them when they fail. This is effectively trading CPU cycles for engineering time. At low levels of flakiness, this trade-off makes sense. Just keep in mind that rerunning a test is only delaying the need to address the root cause of flakiness.

Surlignement (Jaune) | Page 343

> test flakiness continues to grow, you will experience something much worse than lost productivity: a loss of confidence in the tests.

Surlignement (Jaune) | Page 343

> Our experience suggests that as you approach 1% flakiness, the tests begin to lose value.

Surlignement (Jaune) | Page 343

> All tests should strive to be hermetic: a test should contain all of the information necessary to set up, execute, and tear down its environment.

Page 344

Surlignement (Jaune) | Page 344

> example, they should not rely on a shared database.

Surlignement (Jaune) | Page 344

> As a corollary to this, we also strongly discourage the use of control flow statements like conditionals and loops in a test.

Surlignement (Jaune) | Page 344

> Code is read far more than it is written, so make sure you write the test you'd like to read!

Page 345

Surlignement (Jaune) | Page 345

> It's quite common for a class to have many dependencies or other classes it refers to, and these dependencies will naturally be invoked while testing the target class. Though some other testing strategies make heavy use of test doubles (fakes or mocks) to avoid executing code outside of the system under test, at Google, we prefer to keep the real dependencies in place when it is feasible to do so.

Page 346

Surlignement (Jaune) | Page 346

> Just as we encourage tests of smaller size, at Google, we also encourage engineers to write tests of narrower scope. As a very rough guideline, we tend to aim to have a mix of around 80% of our tests being narrow-scoped unit tests that validate the majority of our business logic; 15% medium-scoped integration tests that validate the interactions between two or more components; and 5% end-to-end tests that validate the entire system.

Page 349

Surlignement (Jaune) | Page 349

> But, unit tests cannot verify the interactions between components, like a contract between two systems developed by different teams.

Surlignement (Jaune) | Page 349

> Beyoncé Rule. Succinctly, it can be stated as follows: “If you liked it, then you shoulda put a test on it.”

Surlignement (Jaune) | Page 349

> The Beyoncé Rule is often invoked by infrastructure teams that are responsible for making changes across the entire codebase.

Page 350

Surlignement (Jaune) | Page 350

> Instead of waiting for a failure, write automated tests that simulate common kinds of failures. This includes simulating exceptions or errors in unit tests and injecting Remote Procedure Call (RPC) errors or latency in integration and end-to-end tests.

Page 351

Surlignement (Jaune) | Page 351

> A better way to approach the quality of your test suite is to think about the behaviors that are tested. Do you have confidence that everything your customers expect to work will work? Do you feel confident you can catch breaking changes in your dependencies? Are your tests stable and reliable? Questions like these are a more holistic way to think about a test suite.

Page 353

Surlignement (Jaune) | Page 353

> Some of the worst offenders of brittle tests come from the misuse of mock objects. Google’s codebase has suffered so badly from an abuse of mocking frameworks that it has led some engineers to declare “no more mocks!” Although that is a strong statement, understanding the limitations of mock objects can help you avoid misusing them.

Page 354

Surlignement (Jaune) | Page 354

> however, if a “wait-and-check” is embedded in a widely used utility, pretty soon you have added minutes of idle time to every run of your test suite. A better solution is to actively poll for a state transition with a frequency closer to microseconds.

Page 355

Surlignement (Jaune) | Page 355

> Basically, treat your tests like production code. When simple changes begin taking nontrivial time, spend effort making your tests less brittle.

Surlignement (Jaune) | Page 355

> addition to developing the proper culture, invest in your testing infrastructure by developing linters, documentation, or other assistance that makes it more difficult to write bad tests. Reduce the number of frameworks and tools you need to support to increase the efficiency of the time you invest to improve things.8 If you don’t invest in making it easy to manage your tests, eventually engineers will decide it isn’t worth having them at all.

Page 359

Surlignement (Jaune) | Page 359

> And every change is expected to include both the feature code and tests. Reviewers are expected to review the quality and correctness of both. In fact, it is perfectly reasonable to block a change if it is missing tests.

Page 360

Surlignement (Jaune) | Page 360

> The belief was that successful ideas would spread, so the focus became demonstrating success.

Page 361

Surlignement (Jaune) | Page 361

> term for this technique is Exploratory Testing. Exploratory Testing is a fundamentally creative endeavor in which someone treats the application under test as a puzzle to be broken, maybe by executing an unexpected set of steps or by inserting unexpected data.

Page 365

Surlignement (Jaune) | Page 365

> Imagine this scenario: Mary wants to add a simple new feature to the product and is able to implement it quickly, perhaps requiring only a couple dozen lines of code. But when she goes to check in her change, she gets a screen full of errors back from the automated testing system. She spends the rest of the day going through those failures one by one.

Page 366

Surlignement (Jaune) | Page 366

> First, the tests she was working with were brittle: they broke in response to a harmless and unrelated change that introduced no real bugs. Second, the tests were unclear: after they were failing, it was difficult to determine what was wrong, how to fix it, and what those tests were supposed to be doing in the first place.

Page 367

Surlignement (Jaune) | Page 367

> Therefore, the ideal test is unchanging: after it's written, it never needs to change unless the requirements of the system under test change.

Page 368

Surlignement (Jaune) | Page 368

> When an engineer refactors the internals of a system without modifying its interface, whether for performance, clarity, or any other reason, the system's tests shouldn't need to change.

Page 369

Surlignement (Jaune) | Page 369

> the presence of the bug suggests that a case was missing from the initial test suite, and the bug fix should include that missing test case. Again, bug fixes typically shouldn't require updates to existing tests.

Surlignement (Jaune) | Page 369

> Changing a system's existing behavior is the one case when we expect to have to make updates to the system's existing tests.

Page 370

Surlignement (Jaune) | Page 370

> that is, make calls against its public API rather than its implementation details.

Page 373

Surlignement (Jaune) | Page 373

> Tests using only public APIs are, by definition, accessing the system under test in the same manner that its users would. Such tests are more realistic and less brittle because they form explicit contracts: if such a test breaks, it implies that an existing user of the system will also be broken.

Page 374

Surlignement (Jaune) | Page 374

> If a method or class exists only to support one or two other classes (i.e., it is a “helper class”), it probably shouldn’t be considered its own unit, and its functionality should be tested through those classes instead of directly.

Surlignement (Jaune) | Page 374

> but it is designed to provide a general piece of functionality useful in a range of contexts (i.e., it is a “support library”), it should also be considered a unit and tested directly. This will usually create some redundancy in testing given that the support library’s code will be covered both by its own tests and the tests of its users. However, such redundancy can be valuable: without it, a gap in test coverage could be introduced if one of the library’s users (and its tests) were ever removed.

Surlignement (Jaune) | Page 374

> Google, we’ve found that engineers sometimes need to be persuaded that testing via public APIs is better than testing against implementation details.

Page 375

Surlignement (Jaune) | Page 375

> Test State, Not Interactions

Surlignement (Jaune) | Page 375

> interaction tests check how a system arrived at its result, whereas usually you should care only what the result is.

Page 376

Surlignement (Jaune) | Page 376

> This test more accurately expresses what we care about: the state of the system under test after interacting with

Surlignement (Jaune) | Page 376

> The most common reason for problematic interaction tests is an over reliance on mocking frameworks.

Page 377

Surlignement (Jaune) | Page 377

> When a test fails, an engineer's first job is to identify which of these cases the failure falls into and then to diagnose the actual problem. The speed at which the engineer can do so depends on the test's clarity. A clear test is one whose purpose for existing and reason for failing is immediately clear to the engineer diagnosing a failure.

Page 378

Surlignement (Jaune) | Page 378

> With an unclear test, you might never understand its purpose, since removing the test will have no effect other than (potentially) introducing a subtle hole in test coverage.

Surlignement (Jaune) | Page 378

> Two high-level properties that help tests achieve clarity are completeness and conciseness. A test is complete when its body contains all of the information a reader needs in order to understand how it arrives at its result. A test is concise when it contains no other distracting or irrelevant information.

Page 379

Surlignement (Jaune) | Page 379

> Test Behaviors, Not Methods

Surlignement (Jaune) | Page 379

> The first instinct of many engineers is to try to match the structure of their tests to the structure

Page 382

Surlignement (Jaune) | Page 382

> Behavior-driven tests tend to be clearer than method-oriented tests for several reasons. First, they read more like natural language, allowing them to be naturally understood rather than requiring laborious mental parsing. Second, they more clearly express cause and effect because each test is more limited in scope. Finally, the fact that each test is short and descriptive makes it easier to see what functionality is already tested and encourages engineers to add new streamlined test methods instead of piling onto existing methods.

Page 385

Surlignement (Jaune) | Page 385

> Method-oriented tests are usually named after the method being tested

Surlignement (Jaune) | Page 385

> The test name is very important: it will often be the first or only token visible in failure reports, so it’s your best opportunity to communicate the problem when the test breaks. It’s also the most straightforward way to express the intent of the test.

Surlignement (Jaune) | Page 385

> A test’s name should summarize the behavior it is testing. A good name describes both the actions that are being taken on a system and the expected outcome.

Page 387

Surlignement (Jaune) | Page 387

> Don’t Put Logic in Tests

Page 389

Surlignement (Jaune) | Page 389

> When the whole string is written out, we can see right away that we’re expecting two slashes in the URL instead of just one.

Surlignement (Jaune) | Page 389

> in test code, stick to straight-line code over clever logic, and consider tolerating some duplication when it makes the test more descriptive and meaningful.

Page 390

Surlignement (Jaune) | Page 390

> Because the first assertion only receives a Boolean value, it is only able to give a generic error message like “expected <true> but was <false>,” which isn’t very informative in a failing test output.

Page 393

Surlignement (Jaune) | Page 393

> newUser().setState(State.NORMAL).build();

Page 394

Surlignement (Jaune) | Page 394

> These tests have more duplication, and the test bodies are a bit longer, but the extra verbosity is worth it. Each individual test is far more meaningful and can be understood entirely without leaving the test body. A reader of these tests can feel confident that the tests do what they claim to do and aren’t hiding any bugs.

Page 395

Surlignement (Jaune) | Page 395

> Example 12-21. Shared values with ambiguous names

Page 396

Surlignement (Jaune) | Page 396

> Engineers are usually drawn to using shared constants because constructing individual values in each test can be verbose. A better way to accomplish this goal is to construct data using helper methods (see Example 12-22) that require the test author to specify only values they care about, and setting reasonable defaults7 for all other values.

Surlignement (Jaune) | Page 396

> languages without named parameters can use constructs such as the Builder pattern to emulate them (often with the assistance of tools such as AutoValue):

Page 398

Surlignement (Jaune) | Page 398

> One risk in using setup methods is that they can lead to unclear tests if those tests begin to depend on the particular values used in setup. For example, the test in Example 12-23 seems incomplete because a reader of the test needs to go hunting to discover where the string “Donald Knuth” came from.

Page 401

Surlignement (Jaune) | Page 401

> Sometimes, it can also be valuable to share code across multiple test suites. We refer to this sort of code as test infrastructure.

Surlignement (Jaune) | Page 401

> Test infrastructure needs to be treated as its own separate product, and accordingly, test infrastructure must always have its own tests.

Page 402

Surlignement (Jaune) | Page 402

> careless use of unit testing can result in a system that requires much more effort to maintain and takes much more effort to change without actually improving our confidence in said system.

Surlignement (Jaune) | Page 402

> Strive for unchanging tests.

Page 403

Surlignement (Jaune) | Page 403

> In many cases, it can even be useful to slightly randomize the default values returned for fields that aren’t explicitly set. This helps to ensure that two different instances won’t accidentally compare as equal, and makes it more difficult for engineers to hardcode dependencies on the defaults.

Page 405

Surlignement (Jaune) | Page 405

> If the codebase isn’t designed with testing in mind and you later decide that tests are needed, it can require a major commitment to refactor the code to support the use of test doubles.

Page 406

Surlignement (Jaune) | Page 406

> In many cases, test doubles are not suitable and engineers should prefer to use real implementations instead.

Surlignement (Jaune) | Page 406

> If the behavior of a test double significantly differs from the real implementation, tests that use the test double likely wouldn’t provide much value

Page 407

Surlignement (Jaune) | Page 407

> One lesson we learned the hard way is the danger of overusing mocking frameworks,

Surlignement (Jaune) | Page 407

> these tests were easy to write, we suffered greatly given that they required constant effort to maintain while rarely finding bugs.

Page 409

Surlignement (Jaune) | Page 409

> Dependency injection is a common technique for introducing seams.

Page 411

Surlignement (Jaune) | Page 411

> Writing testable code requires an upfront investment. It is especially critical early in the lifetime of a codebase because the later testability is taken into account, the more difficult it is to apply to a codebase. Code written without testing in mind typically needs to be refactored or rewritten before you can add appropriate tests.

Page 415

Surlignement (Jaune) | Page 415

> Similar to stubbing, interaction testing is typically done through mocking frameworks.

Page 416

Surlignement (Jaune) | Page 416

> As discussed later in this chapter, interaction testing is useful in certain situations but should be avoided when possible because overuse can easily result in brittle tests.

Surlignement (Jaune) | Page 416

> Although test doubles can be invaluable testing tools, our first choice for tests is to use the real implementations of the system under test's dependencies; that is, the same implementations that are used in production code.

Surlignement (Jaune) | Page 416

> At Google, the preference for real implementations developed over time as we saw that overuse of mocking frameworks had a tendency to pollute tests with repetitive code that got out of sync with the real implementation and made refactoring difficult.

Surlignement (Jaune) | Page 416

> Preferring real implementations in tests is known as classical testing. There is also a style of testing known as mockist testing, in which the preference is to use mocking

Surlignement (Jaune) | Page 416

> Prefer Realism Over Isolation

Page 417

Surlignement (Jaune) | Page 417

> We prefer realistic tests because they give more confidence that the system under test is working properly. If unit tests rely too much on test doubles, an engineer might need to run integration tests or manually verify that their feature is working as expected in order to gain this same level of confidence.

Page 418

Surlignement (Jaune) | Page 418

> Example 13-10. The @DoNotMock annotation @DoNotMock("Use SimpleQuery.create() instead of mocking.")
>
> public abstract class Query {
>
> public abstract String getQueryValue();
>
> }

Page 419

Surlignement (Jaune) | Page 419

> However, for more complex code, using a real implementation often isn’t feasible. There might not be an exact answer on when to use a real implementation or a test double given that there are trade-offs to be made, so you need to take the following considerations into account.

Surlignement (Jaune) | Page 419

> As a result, a test double can be very useful when the real implementation is slow.

Surlignement (Jaune) | Page 419

> How slow is too slow for a unit test? If a real implementation added one millisecond to the running time of each individual test case, few people would classify it as slow. But what if it added 10 milliseconds, 100 milliseconds, 1 second, and so on? There is no exact answer here

Page 420

Surlignement (Jaune) | Page 420

> A real implementation can be much more complex compared to a test double, which increases the likelihood that it will be nondeterministic. For example, a real implementation that utilizes multithreading might occasionally cause a test to fail

Page 421

Surlignement (Jaune) | Page 421

> test double should be used to prevent the test from depending on an external server. If using a test double is not feasible, another option is to use a hermetic instance of a server, which has its life cycle controlled by the test.

Surlignement (Jaune) | Page 421

> Instead of relying on the system clock, a test can use a test double that hardcodes a specific time.

Surlignement (Jaune) | Page 421

> When using a real implementation, you need to construct all of its dependencies. For example, an object needs its entire dependency tree to be constructed: all objects that it depends on, all objects that these dependent objects depend on, and so on.

Page 422

Surlignement (Jaune) | Page 422

> Rather than manually constructing the object in tests, the ideal solution is to use the same object construction code that is used in the production code, such as a factory method or automated dependency injection.

Surlignement (Jaune) | Page 422

> If using a real implementation is not feasible within a test, the best option is often to use a fake in its place. A fake is preferred over other test double techniques because it behaves similarly to the real implementation:

Page 423

Surlignement (Jaune) | Page 423

> public class FakeFileSystem implements FileSystem

Surlignement (Jaune) | Page 423

> Fakes can be a powerful tool for testing: they execute quickly and allow you to effectively test your code without the drawbacks of using real implementations. A single fake has the power to radically improve the testing experience of an

Page 424

Surlignement (Jaune) | Page 424

> A fake also requires maintenance: whenever the behavior of the real implementation changes, the fake must also be updated to match this behavior. Because of this, the team that owns the real implementation should write and maintain a fake.

Page 425

Surlignement (Jaune) | Page 425

> Perhaps the most important concept surrounding the creation of fakes is fidelity; in other words, how closely the behavior of a fake matches the behavior of the real implementation.

Page 426

Surlignement (Jaune) | Page 426

> fake might not need to have 100% of the functionality of its corresponding real implementation, especially if such behavior is not needed by most

Surlignement (Jaune) | Page 426

> tests (e.g., error handling code for rare edge cases). It is best to have the fake fail fast in this case; for example, raise an error if an unsupported code path is executed. This failure communicates to the engineer that the fake is not appropriate in this situation.

Surlignement (Jaune) | Page 426

> fake must have its own tests to ensure that it conforms to the API of its corresponding real implementation.

Page 427

Surlignement (Jaune) | Page 427

> One approach to writing tests for fakes involves writing tests against the API's public interface and running those tests against both the real implementation and the fake (these are known as contract tests). The tests that run against the real implementation will likely be slower, but their downside is minimized because they need to be run only by the owners of the fake.

Surlignement (Jaune) | Page 427

> If the owners of an API are unwilling or unable to create a fake, you might be able to write your own. One way to do this is to wrap all calls to the API in a single class and then create a fake version of the class that doesn't talk to the API.

Page 429

Surlignement (Jaune) | Page 429

> A key sign that stubbing isn't appropriate for a test is if you find yourself mentally stepping through the system under test in order to understand why certain functions in the test are stubbed.

Surlignement (Jaune) | Page 429

> Stubbing leaks implementation details of your code into your test. When implementation details in your production code change, you'll need to update your tests to reflect these changes.

Surlignement (Jaune) | Page 429

> With stubbing, there is no way to ensure the function being stubbed behaves like the real implementation,

Surlignement (Jaune) | Page 429

> example, if you call database.save(item) on either a real implementation or a fake, you might be able to retrieve the item by calling database.get(item.id()) given that both of these calls are accessing internal state, but with stubbing, there is no way to do this.

Page 431

Surlignement (Jaune) | Page 431

> Example 13-14. Refactoring a test to avoid stubbing

Page 432

Surlignement (Jaune) | Page 432

> A test that requires many functions to be stubbed can be a sign that stubbing is being overused, or that the system under test is too complex and should be refactored.

Page 434

Surlignement (Jaune) | Page 434

> At Google, we've found that emphasizing state testing is more scalable; it reduces test brittleness, making it easier to change and maintain code over time.

Surlignement (Jaune) | Page 434

> The primary issue with interaction testing is that it can't tell you that the system under test is working properly; it can only validate that certain functions are called as expected. It requires you to make an assumption about the behavior of the code; for example, "If database.save(item) is called, we assume the item will be saved to the database."

Surlignement (Jaune) | Page 434

> Some people at Google jokingly refer to tests that overuse interaction testing as change-detector tests because they fail in response to any change to the production code, even if the behavior of the system under test remains unchanged.

Page 440

Surlignement (Jaune) | Page 440

> There is often no exact answer regarding whether to use a real implementation or a test double, or which test double technique to use. An engineer might need to make some trade-offs when deciding the proper approach for their use case.

Page 443

Surlignement (Jaune) | Page 443

> The primary reason larger tests exist is to address fidelity. Fidelity is the property by which a test is reflective of the real behavior of the system under test (SUT).

Page 446

Surlignement (Jaune) | Page 446

> At Google, configuration changes are the number one reason for our major outages.

Page 447

Surlignement (Jaune) | Page 447

> Unit tests are limited by the imagination of the engineer writing them.

Page 448

Surlignement (Jaune) | Page 448

> Larger tests often violate all of these constraints. For example, larger tests are often flakier because they use more infrastructure than does a small unit test. They are also often much slower, both to set up as well as to run. And they have trouble scaling because of the resource and time requirements, but often also because they are not isolated — these tests can collide with one another.

Surlignement (Jaune) | Page 448

> Additionally, larger tests present two other challenges.

Surlignement (Jaune) | Page 448

> Unlike unit tests, larger tests suffer a lack of standardization in terms of the infrastructure and process by which they are written, run, and debugged.

Page 455

Surlignement (Jaune) | Page 455

> The rate of distinct scenarios to test in an end-to-end way can grow exponentially or combinatorially depending on the structure of the system under test, and that growth does not scale. Therefore, as the system grows, we must find alternative larger testing strategies to keep things manageable.

Surlignement (Jaune) | Page 455

> However, the value of such tests also increases because of the decisions that were necessary to achieve this scale. This is an impact of fidelity: as we move toward larger-N layers of software, if the service doubles are lower fidelity (1-epsilon), the chance of bugs when putting it all together is exponential in N.

Page 456

Surlignement (Jaune) | Page 456

> Even for integration tests, smaller is better — a handful of large tests is preferable to an enormous one. And, because the scope of a test is often coupled to the scope of the SUT, finding ways to make the SUT smaller help make the test smaller.

Surlignement (Jaune) | Page 456

> One way to achieve this test ratio when presented with a user journey that can require contributions from many internal systems is to “chain” tests,

Surlignement (Jaune) | Page 456

> This is done by ensuring that the output of one test is used as the input to another test by persisting this output to a data repository.

Page 458

Surlignement (Jaune) | Page 458

> An SUT with high hermeticity will have the least exposure to sources of concurrency and infrastructure flakiness.

Surlignement (Jaune) | Page 458

> The SUT’s accuracy in reflecting the production system being tested.

Page 461

Surlignement (Jaune) | Page 461

> There are particularly painful testing boundaries that might be worth avoiding. Tests that involve both frontends and backends become painful because user interface (UI) tests are notoriously unreliable and costly: UIs often change in look-and-feel ways that make UI tests brittle but do not actually impact the underlying behavior. UIs often have asynchronous behaviors that are difficult to test.

Surlignement (Jaune) | Page 461

> Although it is useful to have end-to-end tests of a UI of a service all the way to its backend, these tests have a multiplicative maintenance cost for both the UI and the backends.

Surlignement (Jaune) | Page 461

> Therefore, it is not recommended to have automated tests use a real third-party API, and that dependency is an important seam at which to split tests.

Page 462

Surlignement (Jaune) | Page 462

> The key is to identify trade-offs between fidelity and cost/reliability, and to identify reasonable boundaries.

Page 463

Surlignement (Jaune) | Page 463

> At Google, we do something a little bit different. Our most popular approach (for which there is a public API) is to use a larger test to generate a smaller one by recording the traffic to those external services when running the larger test and replaying it when running smaller tests. The larger, or “Record Mode” test runs continuously on post-submit, but its primary purpose is to generate these traffic logs (it must pass, however, for the logs to be generated). The smaller, or “Replay Mode” test is used during development and presubmit testing.

Surlignement (Jaune) | Page 463

> What happens for new tests or tests where the client behavior changes significantly? In these cases, a request might no longer match what is in the recorded traffic file, so the test cannot pass in Replay mode. In that circumstance, the engineer must run the test in Record mode to generate new traffic, so it is important to make running Record tests easy, fast, and stable.

Page 469

Surlignement (Jaune) | Page 469

> becomes difficult to mitigate noise from the production

Page 475

Surlignement (Jaune) | Page 475

> For example, a prober might perform a Google search at www.google.com and verify that a result is returned, but not actually verify the contents of the result. In that respect, they are “smoke tests” of the production system, but they provide early detection of major issues.

Page 479

Surlignement (Jaune) | Page 479

> Although standard unit test infrastructure might not apply, it is still critical to integrate larger tests into the developer workflow.

Page 482

Surlignement (Jaune) | Page 482

> If there are hardcoded timeouts or (especially) sleep statements in the production code to account for production system delay, these should be made tunable and reduced when running tests.

Page 493

Surlignement (Jaune) | Page 493

> the more users of a system, the higher the probability that users are using it in unexpected and unforeseen ways, and the harder it will be to deprecate and remove such a system.

Page 494

Surlignement (Jaune) | Page 494

> staffing a team and spending time removing obsolete systems costs real money, whereas the costs of doing nothing and letting the system lumber along unattended are not readily observable. It can be difficult to convince the relevant stakeholders that deprecation efforts are worthwhile, particularly if they negatively impact new feature development. Research techniques, such as those described in Chapter 7, can provide concrete evidence that a deprecation is worthwhile.

Page 495

Surlignement (Jaune) | Page 495

> Many software engineers are attracted to the task of building and launching new systems, not maintaining existing ones.

Page 501

Surlignement (Jaune) | Page 501

> “hope is not a strategy.”

Page 502

Surlignement (Jaune) | Page 502

> We’ve learned at Google that without explicit owners, a deprecation process is unlikely to make meaningful progress, no matter how many warnings and alerts a system might generate.

Page 505

Surlignement (Jaune) | Page 505

> statically determine which customers use a given library, and often to sample existing usage to see what sorts of behaviors customers are unexpectedly depending on. Because runtime dependencies generally require some static library or thin client use, this technique yields much of the information needed to start and run a deprecation process. Logging and runtime sampling in production help discover issues with dynamic dependencies. Finally, we treat our global test suite as an oracle to determine whether all references to an old symbol have been removed.

Page 506

Surlignement (Jaune) | Page 506

> prevent deprecation backsliding on a micro level, we use the Tricorder static analysis framework to notify users that they are adding calls into a deprecated system and give them feedback on the appropriate replacement.
