# SOFTWARE TESTING IN THE REAL WORLD pdf

## 1: Software Testing: Top Tips, Tricks and Strategies

*Software Testing In The Real World provides the reader with a tool-box for effectively improving the software testing process. The book gives the practicing software.*

Whether their obscure environments break your client software or they somehow trigger impossible game logic bugs, the sheer number of real-world users and the variation in their interactions are going to find problems you never even imagined were possible. More users also means more unhappy customers and a whole new pile of problems to fix. For better or for worse, all software projects have some variation of real-world testing and a corresponding feedback loop for fixing the problems it finds. What are the benefits and costs of real-world testing? Who is it for: Some of the testing can also measure whether the software meets organizational needs. Consider a high traffic website which makes money by advertising. Changes to the design of the website can have a significant impact on ad revenue, but the only way to truly measure the impact is go live with the changes. Real-world testing will catch problems that other forms of testing have not. And if done right, real-world testing can mitigate the cost of defects by channeling user unhappiness into productive bug reports, and by redirecting the main testing effort to users who are more willing to take on the cost of finding defects. Feedback from real-world testing is slow to non-existent, and for the most part only occurs after problems have impacted users or production systems. This reduces the benefits of finding problems since discovered problems are hard to reproduce and fixing the problem happens late in the development cycle. Developing software infrastructure to help users report problems, and make debugging problems easier, can take significant resources. Focusing testing on particular subset of users may be easy or difficult, depending on the particulars of a product. Voice and exit You will recall from earlier chapters that unhappy users have two choices: Since software products usually have no purpose without users your organization usually want your users to stick around. And since your users are after all testing your software for you, hearing their problems means your development team actually have a chance to fix the issues they find. Your goal then is to reduce the costs of defects in your software by encouraging voice and discouraging exit. You can do so by: Making it easy to report problems. Actively gathering feedback from users who would otherwise just grumble to themselves, or worse yet have already given up on the product. Fixing problems quickly, before they hit other users. The feedback loop for fixing problems requires noticing the problem, communicating it to the development team, figuring out how to address it and then distributing the fix. If you get all of them right, you can reduce the negative impact of defects caught by real-world testing and take advantage of the testing your users are providing. Noticing problems The first step to dealing with the problems discovered by real-world testing is to notice them: Crashes are easy to notice, but other problems may be harder to detect. A confusing user interface may irritate your users, but perhaps not consciously enough to articulate the problem. Similarly, missing documentation, minor bugs and missing functionality may go unnoticed, as motivated users will find a workaround and less motivated users simply give up. Ongoing usability testing can also help you find these issues. Online applications can also use analytics to discover problems, e. You should also be on the lookout for problems that the resiliency of your system hides from users: Monitoring of your system is key to discovering these problems before they get bad enough to start affecting users. Communicating problems Once your users notice a problem you want them to communicate with the development team. We want this to happen as close as possible in time to when the problem occurred: To help with diagnosis. User recall varies from decent in the 5 second range to "there was a flying moose on the screen? Additionally, if we have a two-way communication channel with the user reporting the problem we can ask for more details or further investigation in situ. To reduce the latency in our problem feedback loop. Faster reporting mean faster solutions. We also want to make it as easy as possible for the user to report the problem. Each additional step, e. For example, an unhelpful error page on a website will say something like "An error has occurred, please refresh your browser. If the issue persists please report the problem. You can do better by: Providing a form for reporting the error. If you want to add additional feedback about what you were doing please do so in the form below. Every user interface page or screen

should have a way to give feedback. Command-line tools never seem to do this. This may be reasonable for an open source project with limited resources, but commercial organizations can and should do better. Your software should report every obvious error to you as automatically as possible. This is commonly done for crashes in desktop applications, where a post-crash dialog asks users if they wish to report the problem. Look for other opportunities to explicitly ask for feedback, since some users will never get around to reporting problems they encounter. If at all possible, gather contact information to allow for clarification and questions about the problem later on. Try to identify users who have chosen exit, e. Thank your users for giving you feedback and try to respond as quickly as possible. A note from a human being will be much more effective than an automated response, but the latter is better than silence. And that means your users will continue to be unhappy. Many bug reports are similarly unhelpful. Even if the bug report is sufficiently detailed it can often be hard to reproduce the problem. Most other forms of testing happened in controlled environments, which makes investigation and reproduction easier. Real-world testing happens out in the wild, and reproducing that elusive crash may well be impossible. For crashes, you should be automatically capturing stack traces, what the user was doing, and so on, and as discussed above sending a bug report automatically. More generally, you can add extensive logging to your program and include the relevant logs either automatically in the bug report, or perhaps allow log extraction and uploading via an easy-to-use tool. Releasing and distributing the fix You or a user have noticed a problem, communicated it to you, after which managed to diagnose and fix it. The slower and less automated this process, and the release process in general, the more time will elapse between fix and release. At the fastest extreme are systems like websites where an appropriate architecture can allow Continuous Deployment CD: Even in cases where distribution and release is more difficult, a concerted effort can lower release times by orders of magnitude. More recently the time from important bug fix to users getting an update is less than 24 hours. In addition to waiting for bug reports to arrive you can also take more active measures: Pre-releases If you distribute your software to end users as a fixed release, releasing work-in-progress versions of your software can both catch problems earlier and make your users happier. Since pre-releases are opt-in, the problems they contain will only encountered by more adventurous users. By warning them upfront that they are participating in a testing effort you can reduce the chances of users getting angry when they encounter bugs. Staged rollout For live online systems like websites staged rollout is an equivalent to pre-releases. You ask some percentage of users to opt-in to a new version of the code that is running in parallel to the old code. Providing a way to fall back to the old code keeps adventurous users happy if the new version is sufficiently broken, and also provides an avenue for feedback. Over time you can switch more users to the new code, and eventually you can switch the remaining users over automatically without an opt-in. Consider the advertising-driven website which has to worry about even the smallest of design changes decreasing ad revenue. At the end of each day if revenue is lower than expected you can roll back any recently changes deployed. You show some small percentage of randomly chosen visitor a new design; the rest continue to see the current design. You can then compare the revenue from the current and new designs in isolation from other changes. If the new design improves revenue you can deploy it as the default; if it decreases revenue then the cost of discovering this was low. In cases where the software you are writing is also software you can use in your development team or organization you can test your software by using it. For example, Google uses GMail internally for email and so can test new features or designs on employees before releasing the software publicly. Members of your development team or organization can be useful initial users insofar as they have the ability and motivation to voice and communicate problems, and are far less likely to choose exit as a solution. Even when you can apply these techniques, real-world testing still provides slow, costly feedback. As we discussed in previous chapters, other testing techniques can find defects before users starts using your product, providing faster feedback and hopefully reduced costs as well.

# SOFTWARE TESTING IN THE REAL WORLD pdf

## 2: News, Tips, and Advice for Technology Professionals - TechRepublic

*Based on real-world issues and examples, it brings together key methods of software testing with practical implementation techniques, and presents a simple, practical approach for those getting started.*

User acceptance testing UAT is the last phase of the software testing process. During UAT, actual software users test the software to make sure it can handle required tasks in real-world scenarios, according to specifications. UAT is one of the final and most critical software project procedures that must occur before newly developed software is rolled out to the market. UAT is also known as beta testing, application testing or end user testing. Free Webinar Register Today! UAT can be implemented by making software available for a free beta trial on the internet or through an in-house testing team comprised of actual software users. The following are the steps involved in in-house UAT: The UAT strategy is outlined during the planning step. Test cases are designed to cover all the functional scenarios of the software in real-world usage. They are designed in a simple language and manner to make the test process easier for the testers. Selection of testing team: The testing team is comprised of real-world end users. Executing test cases and documenting: The testing team executes the designated test cases. Sometimes it also executes some relevant random tests. All bugs are logged in a testing document with relevant comments. Responding to the bugs found by the testing team, the software development team makes final adjustments to the code to make the software bug free. When all bugs have been fixed, the testing team indicates acceptance of the software application. This shows that the application meets user requirements and is ready to be rolled out in the market. UAT is important because it helps demonstrate that required business functions are operating in a manner suited to real-world circumstances and usage.

# SOFTWARE TESTING IN THE REAL WORLD pdf

## 3: Software Testing as a Service (TaaS)

*Software Testing in the Real World provides the reader with a tool-box for effectively improving the software testing process. The book contains many testing techniques and guidance for creating a strategy for continuous, sustainable improvement within the organization - whatever its size or level of process maturity.*

The technique has been known for almost 20 years [22], but it is only in the last five years that we have seen a tremendous increase in its popularity. So far, information on at least 20 tools that can generate pairwise test cases has been published [1]. Most tools, however, lack practical features that are necessary for them to be used in the industry. This article pays special attention to usability of the pairwise-testing technique. In particular, it does not describe any radically new method of efficient generation of pairwise test suitesâ€”a topic that has already been researched extensivelyâ€”neither does it refer to any specific case studies or results that have been obtained through this method of test-case generation. It does focus on ways in which the pure pairwise-testing approach must be modified to become practically applicable, and on the features that tools must offer to support the tester who is trying to use pairwise testing in practice. This article makes frequent references to PICT, an existing and publicly available tool that is built on top of a flexible combinatorial test-caseâ€”generation engine, which implements several of the concepts that are described herein. A set of possible inputs for any nontrivial piece of software is too large to be tested exhaustively. Techniques such as equivalence partitioning and boundary-value analysis [17] help convert even a large number of test levels into a much smaller set with comparable defect-detection power. Still, if software under test SUT can be influenced by a number of such factors, exhaustive testing again becomes impractical. Over the years, a number of combinatorial strategies have been devised to help testers choose subsets of input combinations that would maximize the probability of detecting defects: Increase in number of exhaustive and pairwise tests with number of test levels Pairwise-testing strategy is defined as follows: Given a set of N independent test factorsâ€”f1, f2, Covering all pairs of tested factor levels has been extensively studied. Mandl described using orthogonal arrays in the testing of a compiler [16]. Tatsumi, in his paper on the Test Case Design Support System used in Fujitsu Ltd [22] , talks about two standards for creating test arrays: When making that crucial distinction, he references an earlier paper by Shimokawa and Satoh [19]. Over the years, pairwise testing was shown to be an efficient and effective strategy of choosing tests [4, 5, 6, 10, 13, 23]. However, as shown by Smith et al. Because the problem of finding a minimal array that covers all pairwise combinations of a given set of test factors is NPcomplete [14], a considerable amount of research has understandably gone into efficient creation of such arrays. Several strategies were proposed in an attempt to minimize the number of tests that were produced [11]. Authors of these combinatorial test-caseâ€”generation strategies often describe additional considerations that must be taken into account before their solutions become practical. In many cases, they propose methods of handling these in the context of their generation strategies. Tatsumi [22] mentions constraints as a way of specifying unwanted combinations or, more generally, dependencies among test factors. Sherwood [18] explores adapting conventional t-wise strategy to invalid testing and the problem of preventing input masking. This article describes PICT, a test-caseâ€”generation tool that has been in use at Microsoft Corporation since , which implements both the t-wiseâ€”testing strategy and features that make the strategy feasible in the practice of software testing. Figure 3 shows an example of a simple model that is used to produce test cases for volume creation and formatting. It is possible, however, to specify a different order of combinations. In the preparation phase, PICT computes all of the information that is necessary for the generation phase. This includes the set P of all parameter interactions to be covered. Each combination of values to be covered is reflected in a parameter-interaction structure. For example, given three parameters A, B two values each , and C three values , and pair-wise generation, three parameter-interaction structures are set up: Each of these has a number of slots that correspond to possible value combinations that participate in a particular parameter interactionâ€”four slots for AB, and six slots each for AC and BC. Parameter-interaction structures Each slot can be marked uncovered, covered, or excluded. All of the uncovered slots in all parameter interactions constitute the set of combinations to be covered. If any constraints were defined in a

model, they are converted into a set of exclusionsâ€”value combinations that must not appear in the final output. Corresponding slots are then marked excluded in parameter-interaction structures and, therefore, removed from the combinations to be covered. The slot becomes covered when the algorithm produces a test case that satisfies that particular combination. The algorithm terminates when there are no uncovered slots. The core generation algorithm is a greedy heuristic. It builds one test case at a time, locally optimizing the solution. It is similar to the algorithm that is used in AETG [6], with the key differences being that the PICT algorithm is deterministic and that it does not produce candidate tests. Therefore, two executions on the same input produce the same output. PICT heuristic algorithm The generation algorithm does not assume anything about the combinations to be covered. It operates on a list of combinations that is produced in the preparation phase. This flexibility of the generation algorithm allows for adding interesting new features easily. The algorithm is also quite effective. It can compute test suites that are comparable in size to other tools that exist in the field, and it is fast enough for all practical purposes. For instance, for 50 parameters with 20 values each , PICT generates a pairwise test suite in under 20 seconds on an Intel Pentium M 1. Advanced Features Mixed-Strength Generation Most commonly, when t-wise testing is discussed, it is assumed that all parameter interactions have a fixed-order t. It is sometimes useful, however, to be able to define different orders of combinations for different subsets of parameters. For example, interactions of parameters B, C, and D might require better coverage than interactions of A or E. We should be able to generate all possible triplets of B, C, and D, and cover all pairs of all other parameter interactions. Possibly, experience has shown that interactions of these parameters are at the root of proportionally more defects than other interactions; therefore, they should be tested more thoroughly. On the other hand, setting a higher t on the entire set of test parameters could produce too many test cases. Using mixed-strength generation might be a way to achieve higher coverage where necessary, without incurring the penalty of having too many test cases. Fixed-strength and mixed-strength generation Cohen et al. AETG actually uses seeding to achieve this. In PICT, because the generation phase operates solely on parameter-interaction structures, they can be manipulated to reflect the need for higher-order interactions of certain parameters. Creating a Parameter Hierarchy To complement the mixed-strength generation, PICT allows a user to create a hierarchy of test parameters. This scheme allows for certain parameters to be t-wiseâ€”combined first, and then that product is used for creating combinations with parameters on upper levels of the hierarchy. This is a useful technique that can be used to 1 model test domains with a clear hierarchy of test parametersâ€”that is, API functions taking structures as arguments and user-interface UI windows with additional dialog boxesâ€”or 2 to limit the combinatorial explosion of certain parameter interactions; 1 is intuitive, and 2 requires explanation. When describing the process of analyzing test parameters [22], Tatsumi distinguishes between input parameters, which are direct inputs to the SUT, and environmental parameters, which constitute the environment in which the SUT operates. Typically, input parameters can be controlled and set much easier than environmental ones compare supplying an API function with different values for its arguments with calling the same function on different operating systems. Because of that, it is sometimes better to constrain the number of environments to the absolute minimum. Consider the example that is shown in Figure 7, which contains the same test parameters as Figure 3, but with hardware specification added. To cover all pairwise combinations of all nine parameters, PICT generated 31 test cases, which included 17 different combinations of the hardware-related parameters: Two-level hierarchy of test parameters Instead, hardware parameters can be designated as a sub-model and pairwise-combined first. The result of this generation is then used to create the final output, in which six individual input parameters and one compound environment parameter take part. The result is a larger test suite 54 tests , but it contains only nine unique combinations of the hardware parameters. Users of this feature have to be cautious, however, and understand that not all t-wise combinations of all nine parameters will be covered in this scheme. If the goal is to achieve low volatility of a certain subset of parameters, one might implement an even better solution. Namely, generate all required t-wise combinations at the lower level platform, CPUs, and RAM , and use them for the higher-level combinations all nine parameters , with the requirement that, in any test case, a combination of platform, CPUs, and RAM must come from the result of the lower-level generation. In this case, one would achieve low volatility and not lose t-wise coverage. This feature has yet to be implemented in

PICT. However, in practice, this is rarely the case. That is why constraints are an indispensable feature of a test-case generator. They describe limitations of the test domainâ€”that is, combinations that are impossible to be successfully executed in the context of given SUT. Going back to the example in Figure 3, the FAT file system cannot actually be applied to volumes that are larger than 4 gigabytes GB. One might think that removing such test cases from the resulting test suite would solve the problem. However, such a test case might cover other, valid combinations for example, [FAT, RAID5] that are not covered elsewhere in the test suite. Researchers recognized this problem very early. Tatsumi describes the concept of constraints and proposes special handling of those by marking test cases with excluded combinations as errors [22]. Later, several different ways in which constraints can be handled were proposed. The simplest methods involve asking a user to manipulate the definition of test parametersâ€”either by splitting parameter definitions onto disjoint subsets [18] or by creating hybrid parameters [25]â€”so that unwanted combinations cannot possibly be chosen. Other methods could involve post-processing of resulting test suites and modifying test cases that violate one or more constraints so that the violation is avoided. PICT uses a similar language of constraint rules. Parameters for volume creation and formatting augmented with constraints PICT internally translates constraints into a set of combinations that are called exclusions and uses those to mark appropriate slots as excluded in parameter-interaction structures. This method poses two practical problems: How to ensure that all combinations that must be excluded are, in fact, marked excluded How to handle exclusions that are more granular than the corresponding parameter-interaction structureâ€”that is, they refer to a larger number of parameters than there are in the parameter-interaction structure The first problem can be resolved by calculating dependent exclusions. Consider the example that is shown in Figure 9. Constraints on that model create a circular dependency loop between values A: In the end, if the generation is to proceed, we must ensure that we do not pick A: Instead of the initial three, five combinations must be excludedâ€”among them, all combinations of A: Calculating dependent exclusions The second problem is a case in which directly marking combinations as excluded in parameter-interaction structures is impossible. Consider the example that is shown in Figure 10, in which three-element exclusions are created, but parameter-interaction structures refer only to two parameters at a time. In such a case, one more parameter-interaction structure ABC is set up. Appropriate combinations are marked as excluded, and the rest of them are marked as covered. Handling exclusions that are more granular than parameter-interaction structures Both stepsâ€”expanding the set of exclusions to cover all dependent exclusions and adding auxiliary parameter-interaction structuresâ€”happen in the preparation phase.

# SOFTWARE TESTING IN THE REAL WORLD pdf

## 4: What is Functional Testing? Types, Tips, Limitations & More

*Software Testing in the Real World - 2 or 3 day seminar Developers are under great pressure to deliver more complex software with increasingly aggressive schedules and with limited resources. Testers are expected to validate the quality of software even faster and with even fewer resources.*

It is one step in the ongoing process of agile software development. Testing takes place in each iteration before the development components are implemented. Accordingly, software testing needs to be integrated as a regular and ongoing element in the everyday development process. Encourage clarity in bug reporting. A good bug report can save time by avoiding miscommunication or the need for additional communication. Similarly, a bad bug report can lead to a quick dismissal by a developer. Both of these can create problems. Teach people to write good reports, but hold your developers to high standards as well. Just as developers expect detailed and well-written bug reports, so too should testers expect detailed and well-written issue resolutions. Retesting is important, and clear resolutions facilitate better retesting. Treat testing like a team effort. A tester is only as efficient as their team. When everyone understands what the application entails, testers can effectively cover the test cases. Giving testers access to early knowledge will allow them to prepare early test environments. This will avoid any unforeseen issues, preventing any delays or risks while also being cost-effective. Use tools to make testing easy. Ideally, all developers should be able to run all tests, in a single click, right from in their IDE. Recognize that the goal of testing is to mitigate risk, not necessarily eliminate it. Therefore, your definition of quality may vary by application. As you initiate a project, get the right roles involved to ask the right questions: What constitutes perfect versus good enough versus unacceptable? Your ability to achieve quality is improved because the application development team is not charged with unrealistically perfect expectations. Rather, it is chartered with a definition of quality that fits the given time, resource, and budget constraints. This improvement will help you meet business requirements and achieve a satisfying user experience. Business stakeholders and the entire application development team will need to implement this practice. Your user documentation should be tested, too. End users are people who can fall under certain categories and be united by the notion of target audience, but, nevertheless, they are still just a bunch of unique human beings. So, some functionality that is clear to one person is rocket science to another. This proves two points: Keep open lines of communication between testing teams. Communications allow the team to compare results and share effective solutions to problems faced during the test. This will also ensure clear assignment of each task. All members of the team should get updated with the current status of the test. Testing is about reducing risk. Impact can happen with the frequency of an error or undesired functionality, or it can be because of the severity of the problem. However, that would be a high enough frequency to be very annoying to the customer. Think outside of the box. They require testers to become real users for some time and try the most unthinkable scenarios. What we recommend is to start thinking out of the box. There are some useful pieces of advice that might be of help to any tester: Find out what the software under test is not expected to be doing. Try those things out. So you are finding yourself in the middle of Apple Watch testing. How will it act if an iPhone it is paired to runs out of battery, etc.? If possible, get the system or device under test out of your working premises and try it in a real environment. Regular calls and actually talking to each other can work miracles here. For example, we sometimes use a total number of expected defects during test planning and then compare actual defects per hour found versus what we would expect, during test execution. Each of these rules of thumb aids us in managing the information we deal with as testers and QA managers. But complicated, important or security related code greatly benefits from code reviews and will improve your code quality a lot. Manage defects in code during development, particularly for complex code. Once the defects have been prioritized, developers should be able to automatically find all of the places the defect exists across projects and code branches â€” thus minimizing duplication of efforts. Then they should be able to collaborate with other developers to share triage information across distributed teams and geographic boundaries. Report findings in the context of business value. Engage the end user. Have them give frequent feedback on the product for future improvement and development; software developers who respond quickly

to customer feedback are generally more successful. In a world of lay-off paranoia, it is a good idea to rise above it all, gain immunity and feel secure. The best way to do so is to make learning a habit. Bug summaries must be thorough. This is especially true when they have more bugs to review. Use Test Maturity Model integration. In the face of this truism, numerous techniques to reduce the number and severity of defects in software have been developed, with the ultimate, albeit unobtainable, goal of defect elimination. Such optimistic thinking has led to significant improvements in software quality over the past decade, notwithstanding increased software complexity and customer demands. Broadly, these are structures which state where an organization sits on a maturity scale, where its failings lie and what should be done to improve the situation using process improvement frameworks. Always start with a product map. A graphical model for example, a mind map can provide a concise, easy-to-understand representation of the product, and the modeling process is likely to help you uncover features that you may not previously have been aware of. Getting testers involved from the start means you can eliminate many errors even before reaching the development stage. Writing test scripts, quality testers assist developers that can later use these scripts for making product creation easier. Thus, involving testers into work at the first stages of development has a range of advantages: Choose flexible test management tools that can adapt to your needs. Keeping this in mind, you should look for a test management tool which not only fits your day-to-day testing needs today but should also offer flexibility if your testing approach changes course in the future. Create sample test data if needed. Since in many testing environments creating test data takes many pre-steps or test environment configurations which are very time-consuming. Also If test data generation is done while you are in test execution phase, you may exceed your testing deadline. Make sure developers have the test cases. It ensures that re-work would be minimum since most important part of the application is taken care by the developer himself. Follow a proven process for functional testing. Quite simply, functional testing looks at what software is supposed to do and makes sure it actually does that. If they do not match assuming you properly understand what the outcome should have been and used the correct input , then there is an issue with the software. Understand the data flow. Hence, recognize how the data is used within the application early on in order to report bugs and defects faster. Write your tests for the correct features to cut your maintenance costs. Talk with your stakeholders and product owners. Find out what keeps them awake at night. It will cut your maintenance costs dramatically. Channel an attacker for security testing. Start with the most common methods and attack scenarios. For apps, including the device in your testing plan is imperative. Part of the application testing strategy , if you are developing for situations like this, should include the testing of the robustness of the device itself in adverse operating conditions. If you fail to include the device in your test plan, the app might be great â€” but it might also crash at a critical moment if the end device fails. Both are capable of turning the crank. The methodologies and concepts need to be grasped before you start turning the crank. Extracting anything useful from automated testing requires a robust set of test cases. You need to have clear goals â€” launch the app, poke this set of buttons, get this result. This is true regardless of which method is used. Why a clear and robust test plan is essential , Possible Mobile; Twitter: This means that you need to test the same feature across browsers with different expectations of what is correct. Sometimes it is better to use a well-established approach for everybody. We know from the Browser Wars of the s that this comes at a cost. Restrict your choice of newer features to those that will have the biggest net benefit to the user.

# SOFTWARE TESTING IN THE REAL WORLD pdf

## 5: Software Testing in the Real World: Improving the Process by Edward Kit

*Open Library is an initiative of the Internet Archive, a (c)(3) non-profit, building a digital library of Internet sites and other cultural artifacts in digital form.*

Overview[ edit ] Although testing can determine the correctness of software under the assumption of some specific hypotheses see hierarchy of testing difficulty below , testing cannot identify all the defects within software. These oracles may include but are not limited to specifications, contracts , [3] comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria. A primary purpose of testing is to detect software failures so that defects may be discovered and corrected. Testing cannot establish that a product functions properly under all conditions, but only that it does not function properly under specific conditions. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed. For example, the audience for video game software is completely different from banking software. Therefore, when an organization develops or otherwise invests in a software product, it can assess whether the software product will be acceptable to its end users, its target audience, its purchasers and other stakeholders. Software testing aids the process of attempting to make this assessment. Defects and failures[ edit ] Not all software defects are caused by coding errors. One common source of expensive defects is requirement gaps, e. Software faults occur through the following processes. A programmer makes an error mistake , which results in a defect fault, bug in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure. For example, defects in dead code will never result in failures. A defect can turn into a failure when the environment is changed. Examples of these changes in environment include the software being run on a new computer hardware platform, alterations in source data , or interacting with different software. Input combinations and preconditions[ edit ] A fundamental problem with software testing is that testing under all combinations of inputs and preconditions initial state is not feasible, even with a simple product. More significantly, non-functional dimensions of quality how it is supposed to be versus what it is supposed to do â€" usability , scalability , performance , compatibility , reliability â€"can be highly subjective; something that constitutes sufficient value to one person may be intolerable to another. Combinatorial test design enables users to get greater test coverage with fewer tests. Whether they are looking for speed or test depth, they can use combinatorial test design methods to build structured variation into their test cases. More than a third of this cost could be avoided, if better software testing was performed. Until the s, the term "software tester" was used generally, but later it was also seen as a separate profession. Regarding the periods and the different goals in software testing, [11] different roles have been established, such as test manager, test lead, test analyst, test designer, tester, automation developer, and test administrator. Software testing can also be performed by non-dedicated software testers. Testing approach[ edit ] Static vs. Reviews , walkthroughs , or inspections are referred to as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Dynamic testing takes place when the program itself is run. These two approaches are used to describe the point of view that the tester takes when designing test cases. A hybrid approach called grey-box testing may also be applied to software testing methodology. White-box testing White-box testing also known as clear box testing, glass box testing, transparent box testing, and structural testing verifies the internal structures or workings of a program, as opposed to the functionality exposed to the end-user. In white-box testing, an internal perspective of the system the source code , as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. While white-box testing can be applied at the unit , integration , and system levels of the software testing process, it is usually done at the unit level. Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements. Techniques used in white-box testing include: This allows the software team to examine parts of

a system that are rarely tested and ensures that the most important function points have been tested. This is helpful in ensuring correct functionality, but not sufficient since the same code may process different inputs correctly or incorrectly. Pseudo-tested functions and methods are those that are covered but not specified it is possible to remove their body without breaking any test case. Black-box testing Black box diagram Black-box testing also known as functional testing treats the software as a "black box", examining functionality without any knowledge of internal implementation, without seeing the source code. The testers are only aware of what the software is supposed to do, not how it does it. Test cases are built around specifications and requirements, i. It uses external descriptions of the software, including specifications, requirements, and designs to derive test cases. These tests can be functional or non-functional , though usually functional. Specification-based testing may be necessary to assure correct functionality, but it is insufficient to guard against complex or high-risk situations. Whatever biases the programmers may have had, the tester likely has a different set and may emphasize different areas of functionality. On the other hand, black-box testing has been said to be "like a walk in a dark labyrinth without a flashlight. This method of test can be applied to all levels of software testing: Component interface testing Component interface testing is a variation of black-box testing , with the focus on the data values beyond just the related actions of a subsystem component. One option for interface testing is to keep a separate log file of data items being passed, often with a timestamp logged to allow analysis of thousands of cases of data passed between units for days or weeks. Tests can include checking the handling of some extreme data values while other interface variables are passed as normal values. Visual testing[ edit ] The aim of visual testing is to provide developers with the ability to examine what was happening at the point of software failure by presenting the data in such a way that the developer can easily find the information she or he requires, and the information is expressed clearly. Visual testing, therefore, requires the recording of the entire test process â€” capturing everything that occurs on the test system in video format. Output videos are supplemented by real-time tester input via picture-in-a-picture webcam and audio commentary from microphones. Visual testing provides a number of advantages. The quality of communication is increased drastically because testers can show the problem and the events leading up to it to the developer as opposed to just describing it and the need to replicate test failures will cease to exist in many cases. The developer will have all the evidence he or she requires of a test failure and can instead focus on the cause of the fault and how it should be fixed. Ad hoc testing and exploratory testing are important methodologies for checking software integrity, because they require less preparation time to implement, while the important bugs can be found quickly.

# SOFTWARE TESTING IN THE REAL WORLD pdf

## 6: Software testing in the real world ( edition) | Open Library

*Includes bibliographical references (p. ) and index.*

The importance of any particular factor varies from application to application. In the typical business system usability and maintainability are the key factors, while for a one-time scientific program neither may be significant. Our testing, to be fully effective, must be geared to measuring each relevant factor and thus forcing quality to become tangible and visible. The drawbacks are that it can only validate that the software works for the specified test cases. A finite number of tests can not validate that the software works for all situations. On the contrary, only one failed test is sufficient enough to show that the software does not work. Dirty tests, or negative tests, refers to the tests aiming at breaking the software, or showing that it does not work. A piece of software must have sufficient exception handling capabilities to survive a significant level of dirty tests. A testable design is a design that can be easily validated, falsified and maintained. Because testing is a rigorous effort and requires significant time and cost, design for testability is also an important design rule for software development. For reliability estimation [Kaner93] [Lyu95] Software reliability has important relations with many aspects of software, including the structure, and the amount of testing it has been subjected to. Based on an operational profile an estimate of the relative frequency of use of various inputs to the program [Lyu95] , testing can serve as a statistical sampling method to gain failure data for reliability estimation. Software testing is not mature. It still remains an art, because we still cannot make it a science. We are still using the same testing techniques invented years ago, some of which are crafted methods or heuristics rather than good engineering methods. Software testing can be costly, but not testing software is even more expensive, especially in places that human lives are at stake. Solving the software-testing problem is no easier than solving the Turing halting problem. We can never be sure that a piece of software is correct. We can never be sure that the specifications are correct. No verification system can verify every correct program. We can never be certain that a verification system is correct either. Key Concepts Taxonomy There is a plethora of testing methods and testing techniques, serving multiple purposes in different life cycle phases. Classified by purpose, software testing can be divided into: Classified by life-cycle phase, software testing can be classified into the following categories: By scope, software testing can be categorized as follows: Correctness testing Correctness is the minimum requirement of software, the essential purpose of testing. Correctness testing will need some type of oracle, to tell the right behavior from the wrong one. The tester may or may not know the inside details of the software module under test, e. Therefore, either a white-box point of view or black-box point of view can be taken in testing software. We must note that the black-box and white-box ideas are not limited in correctness testing only. Black-box testing The black-box approach is a testing method in which test data are derived from the specified functional requirements without regard to the final program structure. Because only the functionality of the software module is of concern, black-box testing also mainly refers to functional testing -- a testing method emphasized on executing the functions and examination of their input and output data. In testing, various inputs are exercised and the outputs are compared against specification to validate the correctness. All test cases are derived from the specification. No implementation details of the code are considered. It is obvious that the more we have covered in the input space, the more problems we will find and therefore we will be more confident about the quality of the software. Ideally we would be tempted to exhaustively test the input space. But as stated above, exhaustively testing the combinations of valid inputs will be impossible for most of the programs, let alone considering invalid inputs, timing, sequence, and resource variables. Combinatorial explosion is the major roadblock in functional testing. To make things worse, we can never be sure whether the specification is either correct or complete. Due to limitations of the language used in the specifications usually natural language , ambiguity is often inevitable. Even if we use some type of formal or restricted language, we may still fail to write down all the possible cases in the specification. Sometimes, the specification itself becomes an intractable problem: And people can seldom specify clearly what they want -- they usually can tell whether a prototype is, or is not, what they want after they have been finished. Specification problems contributes approximately 30 percent of all bugs in software.

It is not possible to exhaust the input space, but it is possible to exhaustively test a subset of the input space. Partitioning is one of the common techniques. If we have partitioned the input space and assume all the input values in a partition is equivalent, then we only need to test one representative value in each partition to sufficiently cover the whole input space. Domain testing [Beizer95] partitions the input domain into regions, and consider the input values in each domain an equivalent class. Domains can be exhaustively tested and covered by selecting a representative value s in each domain. Boundary values are of special interest. Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not. Boundary value analysis [Myers79] requires one or more boundary values selected as representative test cases. The difficulties with domain testing are that incorrect domain definitions in the specification can not be efficiently discovered. Good partitioning requires knowledge of the software structure. A good testing plan will not only contain black-box testing, but also white-box approaches, and combinations of the two. White-box testing Contrary to black-box testing, software is viewed as a white-box, or glass-box in white-box testing, as the structure and flow of the software under test are visible to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure. White-box testing is also called glass-box testing, logic-driven testing [Myers79] or design-based testing [Hetzel88]. There are many techniques available in white-box testing, because the problem of intractability is eased by specific knowledge and attention on the structure of the software under test. The intention of exhausting some aspect of the software is still strong in white-box testing, and some degree of exhaustion can be achieved, such as executing each line of code at least once statement coverage , traverse every branch statements branch coverage , or cover all the possible combinations of true and false condition predicates Multiple condition coverage. Test cases are carefully selected based on the criterion that all the nodes or paths are covered or traversed at least once. By doing so we may discover unnecessary "dead" code -- code that is of no use, or never get executed at all, which can not be discovered by functional testing. In mutation testing, the original program code is perturbed and many mutated programs are created, each contains one fault. Each faulty version of the program is called a mutant. Test data are selected based on the effectiveness of failing the mutants. The more mutants a test case can kill, the better the test case is considered. The problem with mutation testing is that it is too computationally expensive to use. The boundary between black-box approach and white-box approach is not clear-cut. Many testing strategies mentioned above, may not be safely classified into black-box testing or white-box testing. It is also true for transaction-flow testing, syntax testing, finite-state testing, and many other testing strategies not discussed in this text. One reason is that all the above techniques will need some knowledge of the specification of the software under test. Another reason is that the idea of specification itself is broad -- it may contain any requirement including the structure, programming language, and programming style as part of the specification content. We may be reluctant to consider random testing as a testing technique. The test case selection is simple and straightforward: Study in [Duran84] indicates that random testing is more cost effective for many programs. Some very subtle errors can be discovered with low cost. And it is also not inferior in coverage than other carefully designed testing techniques. One can also obtain reliability estimate using random testing results based on operational profiles. Effectively combining random testing with other testing techniques may yield more powerful and cost-effective testing strategies. Performance testing Not all software systems have specifications on performance explicitly. But every system will have implicit performance requirements. The software should not take infinite time or infinite resource to execute. Performance has always been a great concern and a driving force of computer evolution. Performance evaluation of a software system usually includes: Typical resources that need to be considered include network bandwidth requirements, CPU cycles, disk space, disk access operations, and memory usage [Smith90]. The goal of performance testing can be performance bottleneck identification, performance comparison and evaluation, etc. The typical method of doing performance testing is using a benchmark -- a program, workload or trace designed to be representative of the typical system usage. It is related to many aspects of software, including the testing process. Directly estimating software reliability by quantifying its related factors can be difficult. Testing is an effective sampling method to measure software reliability. Guided by the operational

profile, software testing usually black-box testing can be used to obtain failure data, and an estimation model can be further used to analyze the data to estimate the present reliability and predict future reliability. Therefore, based on the estimation, the developers can decide whether to release the software, and the users can decide whether to adopt and use the software. Risk of using software can also be assessed based on reliability information. There is agreement on the intuitive meaning of dependable software: The robustness of a software component is the degree to which it can function correctly in the presence of exceptional inputs or stressful environmental conditions. It only watches for robustness problems such as machine crashes, process hangs or abnormal termination. The oracle is relatively simple, therefore robustness testing can be made more portable and scalable than correctness testing.

## 7: Software testing in the real world : improving the process (Book, ) [www.enganchecubano.com]

*The Graybox Testing Methodology is used to test embedded systems. Recent studies have confirmed that the Graybox method can be applied in real-time using software executing on the target platform, expanding the capabilities of this method to include not only path coverage verification but also worst.*

## 8: Software Testing

*www.enganchecubano.com How to Turbo Charge your testing with Crowdsourcing. Software Testing in the Real World.*

## 9: What is User Acceptance Testing (UAT)? - Definition from Techopedia

*Releasing software typically involves testing beyond real-world testing, as we'll discuss in later chapters. The slower and less automated this process, and the release process in general, the more time will elapse between fix and release.*

# SOFTWARE TESTING IN THE REAL WORLD pdf

*Removing anti-Judaism from the pulpit The syntax of existential sentences in Serbian Jutta M. Hartmann and Natalya Milicevic The One That Is Both Precalculus right triangle 3rd edition The GPs guide to personal development plans The value of investigating stakeholder involvement in diversity management by Anne-marie Greene and Gill The early history of factory legislation in Massachusetts, by C. E. Persons. Rang dales pharmacology flash cards updated edition The sorcerers crossing V.8-10. The ring and the book. Conceptual integrated science practice book Design a label seal orange create labels The Berenstain Bears lend a helping hand Ignou date sheet 2017 Zenfone 2 user guide Dynamical Groups and Spectrum Generating Algebras Chebyshev and fourier spectral methods second revised edition Abstract algebra dummit Eight Presidents and Indochina Ten Miles of Bad Road List of hospitals in australia Memorial History of Mulanje Mission. Church of Central Africa Presbyterian in Malawi Revenge Files, The The no-nonsense general-class license study guide filetype Occupation and administration The Nimbus and the Aureole Running QuickBooks 2008 Premier Editions His private honour Saturday Morning, Mozart And Burnt Toast UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools 5500 quilt block designs Un manual para servidores de dios Prehistoric art of india To learn more about Roberto Clemente. World of consumption Ronald Reagan Presidential Library Outlines Highlights for Understanding Human Sexuality by Hyde, ISBN The role of beliefs in inference for rational expectations models Kikkoman Oriental Cooking Google books in format*