

Reducing Test Case Bloat

By

Lanette Creamer



Adobe Systems, Inc

2009

Author Bio

Lanette Creamer has been with Adobe Systems since 2000 testing products such as Adobe InDesign versions 1.5 through CS, Adobe InCopy 1.0, Shared Technologies across applications such as XMP, XML, and has been working as the Quality Lead for the Adobe Creative Suites Workflow Team since 2006, promoting a delightful user experience across products. Lanette studied Graphic Design at Western Washington University, but her true love was for Photoshop, Illustrator, and PageMaker. After attending an inspiring seminar at the CAST conference in 2007, she started a testing blog at <http://blog.testyredhead.com/> hoping to find other people who are passionate about testing. Her technical paper, published in the 2008 Pacific Northwest Software Quality Conference Proceedings and corresponding presentation, "Testing for the User Experience", was selected by attendees for a follow-up presentation to close the conference.

Introduction

We may be ready to move on to the future and work on the “new” and “innovative” features that are being created. For many of us, if we don’t deal with the past, it can easily come back to haunt us, slowing down the creation of new projects and robbing our testing time unexpectedly, often to the point that testing becomes the bottleneck that slows innovation to a crawl.

For those of us who work on software that already exists, exciting new functionality and improvements are the main things that drive upgrades, as well as compatibility with new platforms. However, if end users can’t trust the quality of the legacy features they rely on, they will be reluctant to upgrade, or worse, your new versions will get a reputation of being “unstable” harming overall adoption. Embarrassingly, in some cases users will downgrade their software to an earlier version because they are unhappy with the quality of the newer release. Some users will go so far as to ask for a product that is no longer sold trying to get to a version that they feel is reliable, dragging the company reputation through the mud in the process.

This paper is not specifically about the logical part of testing or about proving that some of your tests are unnecessary using formal mathematics or “combinatorics”^{1,2} to prove some tests are inefficient. It also is not a step by step instruction list you should follow for the most technically efficient list of test cases which you can prove without a doubt are best to cut for your project.

This paper is about the subjective and difficult part of testing which has no provable mathematical “correct” answer. It is about risk management, test planning, cost, value, and being thoughtful about which tests to run in the context of your specific project. I’ll cover why to reduce test case bloat, when it can be done, who does it, along with a few examples that I’ve used in practice. I’ll share the criteria we used to reduce our test cases when faced with the challenge to cover over three times the applications while our test team was reduced from sixteen down to four testers from the previous product cycle.

As testers we are actively participating in building the future of software. We are facing increasingly complex software as we are building new features on top of software that is maturing. While the scope of our testing is expanding, we are often expected to “do more with less”. In these tough economic times we often must balance testing existing features that customers rely on every day with potentially risky new features and interactions. How we address testing legacy code has a significant impact on the quality of future software.

This overview and experience report can help you consider the legacy test cases you are maintaining and running. If you would like to spend more of your time developing new test cases and working on the exciting future of software, consider which test case reduction techniques are the most practical in your context to allow you to test legacy features more efficiently.

1.0 Test Case Bloat

1.1 Why is Test Case Bloat a Problem?

Ideally, you would wrap up a software project, and head right into a new project with your focus firmly on the future. That is much easier to do when you aren't interrupted by the need to test a security patch, a dot release, and a bug fix for a critical account. Even if we did manage to ship perfect software, we still can't predict future environmental changes. A new operating system, browser, or other software our product relies on could change, requiring rework and retesting. While it is impossible to test everything dealing effectively with the past is what can allow us more time to focus on the future. This paper is about one aspect of dealing with legacy software, specifically how to reduce test case bloat.

1.2 What is a Test Case?

Clinically defined a test case is an input and an expected result.³ For my purposes it doesn't matter if a test case is automated or manual so long as it is a planned test. For the purpose of reducing test case bloat, I'd go further and say that it is a test you plan to execute a minimum of once in the product lifecycle.

1.3 What is Bloat?

This is absolutely the million dollar question and quite a tricky one. Test case bloat is any test where the cost of keeping the test is higher than the cost of getting rid of the test for any reason. Even if the reason is simply that the area isn't very visible or used in the software.

Many years ago I may have told you that test case bloat was simply running any test that didn't result in a likely bug. My opinion has changed vastly since those early days of my career. I very much hope to be running critical tests that do not find bugs. As part of the overall software development process, when some sanity tests never fail, that simply means that the stability of existing code has improved upstream. It does not necessarily mean that those cases are "bloat". If failure of the test would be a test blocking bug and catastrophic failure for the software, it is critical to continue running the test and only lower the cost of running it often and earlier.

There are some test cases which result in bugs that I now consider bloat and would cut entirely. In the past I would have kept them simply because I believed that any test case that revealed a bug was vital. However, finding bugs that a user is unlikely to encounter

and is not going to be fixed means effort was expended for no quality improvement. How expensive is it to create, execute, and maintain that test? Is that cost greater than the cost if the test fails and it is only later discovered by users?

There are times when what you cut may not be bloat. There are some situations where the decisions are the equivalent of “Do we cut off the arm or the head?” Well, a person can live without an arm. If you are in a situation where you are so time constrained that critical areas will be untested, you can still communicate the risk, be transparent and use a strategy to test the most important areas first. It is possible to plan for and do testing for a very time constrained project.

On my test team after releasing CS3, the largest product in the history of Adobe at the time, our company had a reorg to align to new initiatives. While we were very lucky to have minimal layoffs, our team went from 16 down to 4 testers as our people were transferred to other projects. Rather than working on the entire Suite, they would be working on Suite components and technologies, leaving just a few of us with a focus on the Suite itself. While this made sense for the changes at the company, it seemed a very daunting task to perform testing with less than 1/3rd the size of the test team remaining to test a collection of products that had grown from 4 to 16 products during the last project. My immediate reaction was despair and a serious concern that we would fail. I did realize with time that while we could not test as much as we tested before with so few people, but we certainly could accomplish important testing. In these situations it helps to take emotion out of the picture, and remember that by making an informed test strategy and executing on it, that no matter what the resource constraint is, you will be directly having a positive impact on product quality. It is even more important to get the priorities right and the bloat out of the planned testing, or critical testing could happen too late, or not at all.

In some software there are vital tests which must always be run so that testers are up to date on the status of these critical features.

Some examples of existing test cases excluded from being possible bloat:

- external regulations an application must adhere to
- most software tests which cover security
- safety
- legal matters
- any test which would be catastrophic and highly visible if it were to fail

Basic build sanity tests are the type of test case that are generally not test case bloat even if they are not finding bugs often. At a base level, to be testable, you need a certain number of test cases to pass. While these aren't the most complicated or esoteric and clever tests you can create, they are critical and must pass for others to rely on a consistent base level of quality to start testing from. Other than the list of exceptions started above, the rest of the existing tests cases, automated, tool assisted, or human executed, are possible candidates to be cut.

By asking the following questions, we were able to change the focus of what we were testing and we ended up not only completing the testing we agreed to execute, but we also managed to run some tests for another team. When are legacy test cases run? How often are they run? What is the cost of maintaining this test? What is the risk of not running this test? Can or should someone else run this?

1.4 When is Bloat reduced?

No matter what software development method is being used, it is possible to reduce test case bloat whenever test planning happens. If that is every week, every month, once per product cycle, or at each product milestone, whenever you are creating test cases, planning for when you are going to test, or preparing a set of tests to run for any milestone, sprint, or certification, it is a good time to think about what percentage of your time will be dedicated to testing legacy code and features, and how you are going to focus on those test cases which will have maximum return. What will you want to consider again for this product, and what are you willing to cut forever and why?

There are many ways to reduce test case bloat, but rather than drawing straws, using the magic 8 ball, or a random number generator to delete test cases via a lottery, I'll cover eight ways that I know of to reduce the total number of tests you are planning to run. With this list to get you started, you can add your own ideas and come up with the best possible combination of preserving the best legacy test cases that still apply.

2.0 Methods for Reducing Test Case Bloat

2.1 Mathematical Combinations

All-Pairs Testing, Pairwise Testing, and orthogonal array, are all variations on the idea that you can use math to find the best combinations of inputs and expected results to create an efficient subset of tests to cover functionality. While many of us use equivalency cases when creating test cases, it can be helpful to use them when reviewing existing test cases or planning to run legacy test cases. Make sure that you are running only the minimum set possible using math by using only the combinations suggested by the tool you run instead of a full set if presented with a full coverage matrix or set of test cases designed to cover all combinations of inputs. This isn't perfect and there is some risk that you could miss.

As a bonus, lots of engineers just love math and will be very pleased that there is a concrete reason for this set of tests based on numbers, so this can be a low pain way to reduce the number of test cases in some contexts. There are some free tools to help, as well as available commercial tools.

In practice this wasn't as useful as I'd hoped in reducing testing scope for my particular project, but because the most math oriented people love it, it was worthwhile to use where possible. The reason math based equivalency may not result in significant bloat

reduction is that equivalency pairs generally apply only to functional test coverage, which is a small percentage of the total tests which are run on most software. It is my experience that you can eliminate some legacy test cases safely this way depending on the design of your legacy test cases.⁴

I think it is a useful way to reduce test case bloat in the instances where someone expects full test coverage of all inputs in some UI. Only a small percentage of the kind of legacy testing that teams at Adobe are tasked with fit this description. Even if it was an entirely perfect best practice, it would not address a majority of the testing challenges we face.

Math alone doesn't solve much of my testing problem, so getting a free tool to do reduction for me is how I include this method in the small percentage of cases it applies in my current work. I wouldn't claim to have a full understanding of it, nor would I be interested in a debate for or against it as a useful practice. For those of you who are interested in the arguments against, check out the article, [Pairwise Testing: A Best Practice that Isn't](#), written by James Bach from PNSQC 2004 which points out some of the flaws in blindly applying tools and math without considering the context.⁵

If you are looking for mathematical programs to help you reduce tests based on math, please see the excellent collection of easy to understand combinatorial testing papers at <http://www.combinatorialtesting.com/clear-introductions-1>.⁶

2.2 Tiered by Importance

One method of dealing with bloat is by organizing the test cases you have into more manageable sized pieces so that time can be allocated in a more strategic manner. While organizing by importance to the project is pretty common, usually it is done as a way to organize test cases rather than to reduce bloat. The idea is that when your test cases are created, you add an extra step where you create a tiered system that works with your method of test case storage. I've seen systems based on when a feature is expected to work and the tiers marked as Alpha, Beta, FC, Milestone 2, Stage 4, M4, July, and Sprint 4. It doesn't really matter how long the iteration or what the marker is called, if it is based on time it is a generally a tiered system for availability, but not always tiered for importance.

While it is vital to know when something is expected to be testable, for test cases I'd suggest an importance tier as well which is pretty standard. So long as the categories reflect how important knowing the result of that test is to the project, it is tiered by importance. The advantage of knowing the importance of your test cases is that you have a built in contingency plan if you must cut tests due to time or resource constraints. You can get agreement and be transparent about your test plans and you can be more confident that what you see as critical to the quality of the project is consistent.

Importance is going to mean different things depending on the surrounding factors. In many cases importance will vary depending on which stakeholders are most critical to the success of the project.

2.3 Stakeholders

- End users-When the results of the test case would impact a large number of users. For some software you can evaluate usage data for legacy tests to more accurately prioritize the impact the test would have for users.
- Internal Customers-When teams within a company rely on the results of this test case. This is very common when components are integrated or shared. If a test case is covering the ability to integrate the importance raises with the number of expected integrations.
- Test-When a failure would block other testing.
- Shareholders-The result of this test represents a major loss of revenue to the company. Many security tests may have more importance for this reason. For some software this means that ad revenue or license revenue depends on the result of this test case.
- Marketing-Product demos could fail or be delayed if this test case fails. A public demo failure is likely to have a different priority than the same result in a test lab.
- Certifications-When any external certification (Logo, operating system, device) could be denied based on the results of the test case.
- Legal-There could be a lawsuit or any undesirable legal result associated with this test case.
- Other-In your specific situation there will quite likely be stakeholders not listed here that need to be considered.

As part of your plan to reduce bloat, it can be helpful to state your assumptions about who is important and where you are placing testing priority and why. When reducing test case bloat you are taking a calculated risk. You are weighing the risk of being unable to test new features by insisting on testing every legacy case against the risk of purposefully not running some tests. When you share your starting assumptions with your stakeholders you offer them the chance to counter with their own assumptions and often you can clarify the boundaries of your testing this way to avoid gaps in testing or duplication.

2.3.1 Test Case Reserve

Some tests are so fundamental to the success of a project that they must be run before any release to customers and in some cases on every build. When reducing test case bloat it can be helpful to know which tests are entirely off limits. I've seen different approaches to this case. On one project there was an automated Build Verification Test called "minimal" and each developer had to run the test suite before they could check in their code. However, in order to improve product quality, "minimal" kept growing. Each

release, more and more features were added to “minimal” and it morphed into what should have been called “maximal”. Not only did this bloat the builds, but it slowed development. Had the bar been higher for the tests allowed in to “minimal”, not only would the developers feel it was more helpful, but the time spent on adding more and more tests could have instead been used to make the validation more effective, the tests faster, and the results more accurate and easy to isolate. This is the ultimate goal of reducing test case bloat, to test with more focus, and rather than testing more, to test what is more important.

When organizing test cases it can be helpful to track which cases fall into this category and for what reason. Then in your test planning you can focus your strategy on moving these tests upstream in the development process, using these tests to measure development progress, and working towards faster notification and resolution of breakages in these critical areas. Reducing the number of tests that fall into this category can allow you to spend more time improving the tests which really add the most value to the project.

Does this mean that some important tests do not get run that should be? Yes. It is well known that it is impossible to test everything. Is it irresponsible to know of a test case and decide not to run it? We know of test cases and decide not to run them every time we design test cases, when we define the scope of our testing, use math to reduce cases, or remove support for a configuration. I challenge you to think about how wrong it is to steal time from the great new test cases that are more important because you are unwilling to let go of test cases which have lost their relevance or are no longer adding significant quality improvement or customer value. It is easy to think about the risk of what we do, but it is difficult to consider the risk of what we do not do. Running the same set of tests because they “must be run” without considering why is risky. I am not suggesting that you throw out all of your legacy test cases, only that you consider the risk and decide what works best for your test project.

If you take the list above under Stakeholders and find a good reason why that test case must be run often (for example, on every build or before every release to a customer), add it to the test case reserve, or collection of tests which are not to be considered for cutting unless the reasons they are critical should change. When collaborating with other testers or handing off a test area to other personnel, it is helpful to understand why a percentage of tests are considered critical to the project.

2.4. Change Based

It's widely accepted that with each code change there is a risk of introducing bugs. For this reason there are a number of ways to use code changes to predict “hot spots” or areas of the software under test which are high risk, especially when testing legacy software. Using information from a change list is a basic example of change based testing that most of us use regularly to guide our testing. In the case of a dot release you can look at the bugs fixed and have some guidance as to which areas are most likely to be impacted.

If you have your test cases categorized by feature area, you can focus the majority of your testing on the areas of change first, only reserving a small percentage of last minute tests for areas of lower risk and less change. This reduces bloat by addressing the “test and retest” cycle that can happen on software projects when every test is considered critical to run at all points during the project.

One other technique which is based on change is to plan major testing events around points of major change. One example of timing testing around changes is that after major integrations we will run an automated suite of sanity tests, and if those pass we will then run broad collaborative user workflow test exercises to get broad coverage and this gives us information on the user experience across multiple products complete with stability information.

While there are many barriers to implementing a visual map showing all changes across multiple products and platforms, I’ve often dreamt of a useful map of all changes per week across products, showing all component integration, bug fixes, and feature check-ins for all products. Even having this information for one product would be helpful to a tester. I have not seen a code check-in tool create such a visual map that is human readable, but I can imagine that one day such a feature or tool might exist and guide test professionals to more efficient focus on the areas of highest risk. I see this as the “laser guided missile” and right now, at least on my team, we don’t have that clear a picture of the code changes going in to the software under test. We can use the data we do have to send out unmanned drones.

We can also use all of the change data we are able to compile to show us areas of low change. These areas are more likely to be stable than areas undergoing major change, so we can look for test cases in these lower risk areas as candidates to test less often, or even not at all if that amount of risk is acceptable for the project.

2.5. Timing Based

While it generally doesn’t sound so great to test managers, strategic procrastination really can be an excellent plan. If you know for a fact that an upcoming change will invalidate the results of certain test cases, it may be better to hold off testing them until after that change has occurred. When you are waiting for a code change, a feature to be testable, or another test to be run, it can be helpful to label the set of tests clearly with what you are waiting for such as a milestone, a feature hand-off, or a calendar date.

I recommend labeling a set of test cases using the exact feature or milestone I’m waiting for rather than the expected date to start running the test cases because it is often easier to maintain. The test triggering reason remains relevant even if the schedule, ownership of the feature, or even the owner of the test case changes.

There are some situations where it is the relevance of a test case is time sensitive and a test case should expire. If your test cases are ranked by priority and you have reached

point in the project where failure of a test case will not possibly result in a bug fix, the test case has far more value if results are reported earlier. It is best to note the expiration date, run the tests when the results can have a positive impact on the software quality, stop testing, and retire those test cases for the rest of the cycle.

2.6. Test Case Find Results

In any set of tests, we usually have test cases which result often in bugs and then we have those duds which have never found a bug of importance. With some exceptions, you may want to consider a results based restructuring of your test cases. Top performing test cases get run first, and possibly more often. Those lowest yielding tests should possibly be put on the bench of “backup” tests. If these tests continue to take time and energy, but do not generate bugs or protect a key area which would be a ship stopping bug if the test ever failed, put them into consideration for retirement based on their contribution to overall quality. In our test case database, we have an “inactive” status which means a test case is not being run currently, but is not deleted and can still be viewed. The past data and the test case itself is tracked, including who moved it to the status of inactive and when, but it is no longer taking up tester time and money to be maintained.

Be careful when considering test cases that you don’t accidentally throw out a test case which exists only because failure would have a huge customer impact. What we really are evaluating the test cases for is “Return on Investment”, and there are some cases which would be so expensive if they ever failed that even if they haven’t ever failed in the past, the risk that they could is too great to take. Past results is only one criterion to use when deciding which legacy test cases receive focus. The impact of failure of that test case happening post ship without your test team knowing the results in advance should be a key consideration.

2.7. Selective Combinations

How many code paths does this test case cover? When designing automated test cases, it is often most efficient for isolation purposes and flexibility to run tests in parallel to methodically test one thing at a time. This can make maintaining the tests and fixing the bugs simpler, but it often will miss bugs that happen when a combination of factors are in play or a series of steps causes a bug rather than one action in isolation. In addition to the properly isolated tests which are designed to cover functionality, I like to compliment that set with a complex scenario which provides broad coverage. The advantage to having a few complex scenario tests which represent a large more detailed set of test cases is that the test can be run by a human to give thoughtful evaluation. Once the more isolated and scripted tests can be successfully automated, having the test suite pass can give you confidence that nothing has broken behind you. However, even very well designed automated validation will have gaps in coverage, so having a few broad tests which combine features in a complex and user focused way can efficiently add to your confidence without duplicating tests already covered by automation.

For example, at one point I was testing the integration of XML parsing core technology to ensure proper import and export of UTF-16 encoded characters. In order to cover the basic functionality, I wanted to ensure we supported every character that the spec supported.

2.7.1 Example

Automated Test Pass-New

Import XML files into InDesign for each character supported in UTF-8 and UTF-16 both in content and tag name.

Export the file and Diff.

Expected: The XML file should be the same.

In this example, each character is processed in a separate file because this makes it very easy to isolate any parsing problem instantly, and also makes the automation more reliable, as if one file fails, the rest of the tests still will continue running.

Automated Test Pass-Legacy

Once all of these files passed, it was very unlikely they would break behind us, yet with each integration I continued to run these tests. Why not? They still counted as “test cases run” and that should give us some confidence.

Instead of running each file through one at a time, I combined all of the files into one XML file for verification of encoding. What this did was gave me just one “Pass or Fail” result summary. If it ever were to fail, I could run the more isolated cases to easily find out why, but I no longer got a pass result for each character. Instead, I asked the question after integration, “Does parsing and exporting UTF-16 encoded characters still work?” For the rest of that release the answer was either “Yes”, or “Total failure of all characters either with importing or exporting.” When you are maintaining legacy code, sometimes a summary is better than a detailed question. In addition to combining the encoding test, I was able to get some wonderful customer content, to test for parsing, test for performance, transformations on import, results of merging and appending with existing content, and the application of style mapping by making one summary test to run to look at the legacy functionality. The advantage of testing a legacy feature is that you have customer feedback, usability data, and often times real world files to test with. Combining the functional tests of the past with some user content can create a useful summary scenario.

You may not need to do a thing with your tests themselves in order to summarize your results. Some test case management systems allow you to customize collections of tests and get only a pass/fail for a specified group. If you are working with such a system, you may just need to organize your tests in a way that makes sense for the

maturity of the code you are working on so that you are notified only when there is a change. People don't use software one feature at a time in an isolated and methodical way, so combining tests often more closely matches the user experience. When reducing test case bloat, look for duplication. Are any of your simple tests already part of your build validation?

Human evaluated summary test scenarios can be to stack one test on the previous test, emulating a more natural user workflow, and that will make running the tests faster, more efficient, and also much more realistic for evaluating the user experience.

We use a custom test case management system at Adobe, which means we can ask for custom fields for tracking test cases for each product. It is possible to create a broad summary case which contains smaller and more isolated test cases within, yet report results against the entire set of cases per test run. If subsets and grouping aren't possible with the tracking tool you are using, you may be able to use a simple keyword per scenario to mark which isolated test cases are covered in the summary scenario. Despite the administrative overhead you may have to endure if your test case management methods do not include bulk updating, you still may save time over running each of the tests in isolation by creating a realistic summary scenario.

2.8. Model Based

Models are being used in software design, automation design, test planning and even results reporting. There are some widely model types used, such as UML models of machine states and user scenario diagrams, as well as some less known models, such as large flow-charts which combine state and user data, and even some artistic cartoon diagrams with stick figures. Since attending a presentation and follow up discussion on Model Based Testing⁷, by Harry Robinson at StarWest 2006, I've used models to help with some aspects of my overall test effort. While I have not yet created my own robot army based on Harry Robinson's ideas, I did use some models to help reduce the number of legacy test cases we were covering for XML export in 2006, and again in 2008. For more reading on Model Based Testing, see http://www.geocities.com/model_based_testing/online_papers.htm

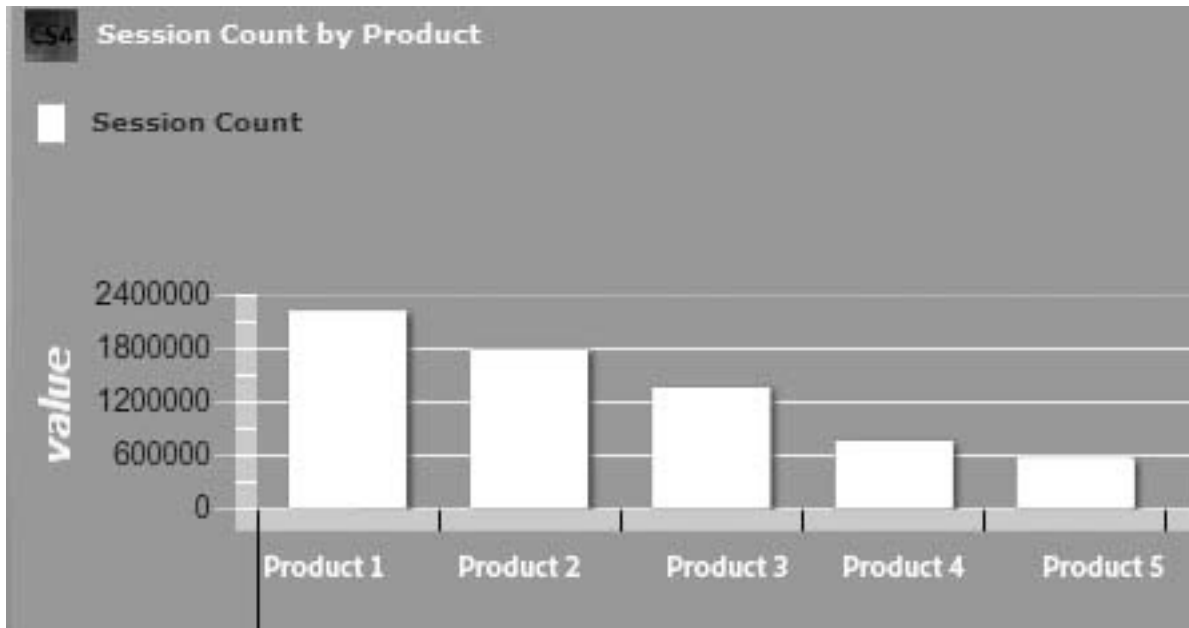
When reducing test case bloat, consider using the models that you have to evaluate your test coverage and examine where you could possibly take more risk, avoid duplication, leverage existing testing based on what is close, or what you could easily "summarize" as explained in section 2.7 based on stability of the area. If you do not currently have any models, check out the above list of online papers and consider which type of models might be helpful in your situation to visualize your overall testing in order to plan where you are going to focus and also where to cut bloat. My experience with models currently includes mainly mapping user workflows⁸ through multiple applications which is much more like making a map from Point A to Point B with scenic viewpoints along the way than it is creating an entire topographical map of a previously unexplored area. Regardless of the type of model, when you have a visual summary of a feature,

you can identify areas of overlap, areas of risk, and possibly areas of interaction where you might want to look for duplicative test cases.

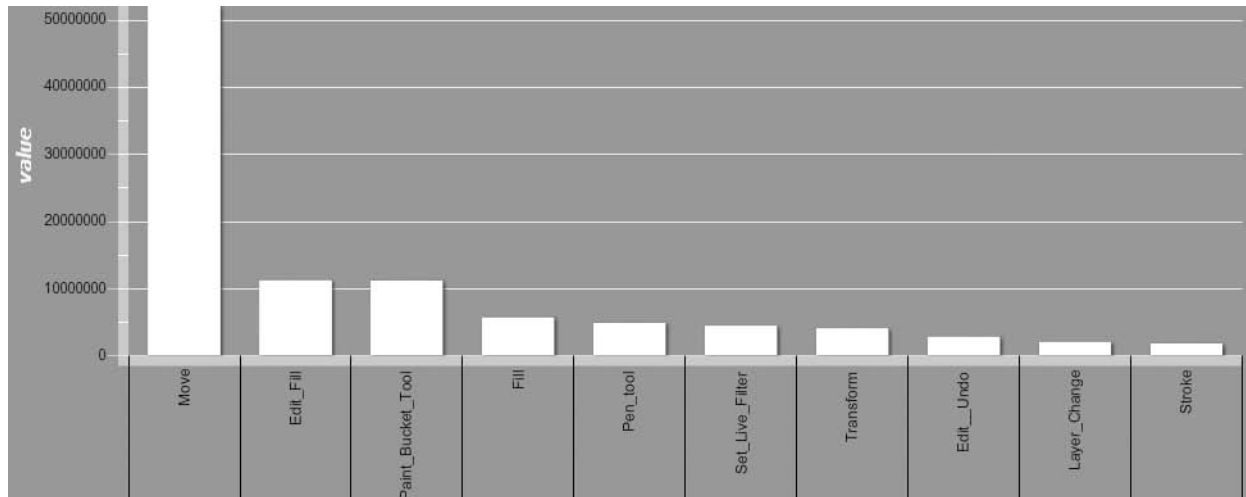
2.9. Customer Workflow Based

The most effective way that currently reduces our legacy test cases is based on customer impact. We combine usage data coming in from our customers allowing us insight into which products they use most commonly, how long each session lasted on average, as well as which were the top ten features used in each product. Future features must also be considered as they significantly impact legacy features as existing functionality changes, and new abilities create previously impossible workflows.

Having objective data confirm what we could only assume previously gave us the courage to reduce our test coverage in some legacy areas so that we could focus our energy on the high impact features, including the new features, areas undergoing major change, and very visible features that impact a large percentage of users. In addition to reducing test case bloat, for the features we discovered were more widely used it was easier to justify the time to automate that test and add it as a build verification test. With the top 5 features covered to some level before the build even passes acceptance, we can focus on the way that the important legacy features interact with the new functionality being added which gives us more time to evaluate the user experience further upstream. The sooner we can offer feedback on new features as well as the impact they have when they are inconsistent with existing workflows, the more cost effective it is to make changes.



Product Usage



Top 10 Features Used in Product

Summary

Any test case that you decide not to run, or to run less often introduces some level of risk that you will miss a bug. However, each legacy test case comes with a cost to run which introduces expense to the project. Evaluating your legacy test cases to reduce redundancy, prioritize, and summarize can allow you to move forward to testing new functionality.

Prioritizing tests can help save time by running the same legacy tests, just less often. If the test doesn't fall into one of the most used workflows or one of the top priority experiences that the software is delivering, it isn't in the top tier. A test being in the top tier means that the test failure will stop shipment of the product. The next tier of tests should be run until Release Candidate submits. Bugs in this category most likely will be fixed if the test fails so long as it can be fixed without jeopardizing the product schedule. The final tier is any test case which must be run during the product cycle which doesn't fit into the first two tiers.

Set out in your test plan what legacy workflows you are protecting and what you are consciously deciding not to test, and why that decision was made. When you have the guts to not test the low priority areas and get stakeholders to sign off on it, you are recognizing and avoiding a larger risk. Adding more and more test cases without ever taking any away is an even bigger risk. If every legacy test is of equal importance, with every new version you need either an exponentially growing QE staff, a schedule that gets longer for testing each time, or you reduce risk by taking on less ambitious new functionality.

References

1. Bellman, R. and Hall, M. *Combinatorial Analysis*. Amer. Math. Soc., 1979., <http://mathworld.wolfram.com/Combinatorics.html>
 2. Combinatorialtesting.com overview, Bolton, Michael Pairwise Testing, DevelopSense©2004
- Videos at <http://www.combinatorialtesting.com/videos>
3. IEEE (1998). *IEEE standard for software test documentation*. New York: IEEE. ISBN 0-7381-1443-X.
 4. P. Black, "My Life with Bugs, or Why I Believe in Combinatorial Testing", American Society for Quality, Gaithersburg, MD Oct. 30, 2007.
 5. James Bach, Pairwise Testing: A Best Practice that Isn't, PNSQC, <http://www.testingeducation.org/>, 2004.
 6. Robinson, Harry, http://www.geocities.com/model_based_testing/
 7. http://en.wikipedia.org/wiki/Model-based_testing
 8. Creamer, Lanette, Testing for the User Experience, PNSQC, 2008.