Defining data pointer for software testing efficiency measurement

Vittalkumar Mirajkar VitalkumarrMirajkar@gmail.comm
Sneha Mirajkar SMirajka@cisco.com
Narayan Naik Narayan_Naik@mcafee.com

Abstract

A well-designed software solution needs a well-designed test approach to ensure quality. Measuring software quality has never been an easy equation to solve. When new software is in the architecture phase, its associated test set is designed assuming certain user patterns and use cases. Relying on the test set to cover the assumed use cases is an inadequate indicator for appropriate software test coverage measurement. You never know when a module is over-tested or under-tested.

There are measurement techniques in place, such as Code Coverage, Unit Coverage, and conditional coverage which help gauge the percentage of software invoked by executing the associated test cases. However, these are not adequate to bring in confidence that post-release no production bugs will be reported. Besides, code coverage measurement needs advanced tools and special builds and set up, to measure them. For software that is already in production, how do we know which modules need immediate attention? There is no simple to use measurement technique that can be used to gauge current software testing efficiency, leading to error-prone tracking of testing effort which leads to an inconsistent effort to outcome mapping.

In this paper, we outline the measurement technique we developed to measure Quality Volatility which helps us gauge product stability and its anticipated performance in upcoming releases. We also propose how an individual test case efficiency can be calculated, which helps in the timely review of test cases for efficacy. This is based on the detection efficiency aging model we have developed.

Biography

Vittalkumar Mirajkar is a Sr Manager at Skyhigh Security, with 16+ years of testing experience ranging from device driver testing, application testing and server testing. He specializes in testing security products. His area of interest is performance testing, soak testing, data analysis and exploratory testing.

Sneha Mirajkar is a Software Engineer at Cisco, with 15+ years of experience in software testing and extensive hands-on in test automation using PYTHON, Selenium, PERL, QTP, VBscript, web services testing and functional testing. She has expertise in cloud testing (SAAS) and IAAS, AWS applications

Narayan Naik is a Software Engineer at Trellix, with 14+ years of experience in exploratory testing and performance testing. He holds an expertise in providing consultation to enterprise customers for features and compatibility of various security products and security solutions deployed. His areas of interest are inter-compatibility test areas, performance testing and encryption product lines.

1. Introduction

Measuring software quality based on established measurement techniques does not assure no bugs will be reported from production. One of the quality measurement techniques is built on the basis of breaking the software into as small as possible testable code and then defining test cases for those units. When a new software is released, no matter how well tested it is and how detailed test measurement used, engineering teams spend the initial days post release, waiting in anticipation on what will be reported from production as a bug. Every production bug reported, exposes a gap in our understanding of the end customer's use cases.

2. Software Measurement Techniques

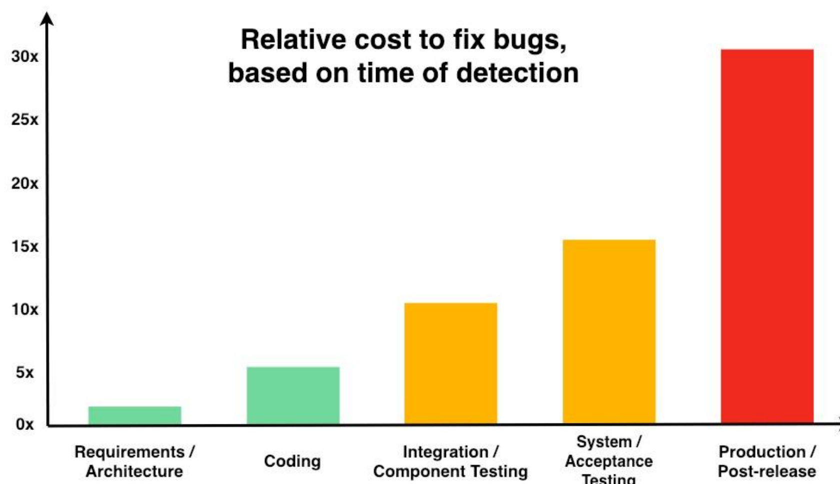
Some of the popular and well-established Quality Measurement metrics are, (Pal Kienitz 2019)

- Unit test coverage (the amount of software code that is covered by unit tests)
- Path coverage (how many linearly independent paths of the program, the test covers)
- Requirements coverage
- Number of defects
- Percentage of automated test coverage (against the total test coverage which includes manual testing)
- Percentage of broken builds, etc

There are many other measurements and test techniques (Pal Kienitz 2019) (Satyabrata Jena 2021) (Sealights.io 2022) which are employed during a software development life cycle. All of these are followed as release check list in anticipation to cover vast majority of software test concerns. However, despite religiously following the quality metrics, production bugs are a living reality.

Example: The most efficient, well tested, almost production bug free software ever released is of NASA shuttle program (Airbrake 2017) (Fishman 1996). 11 release version and only 17 errors report. This requires eliminating almost all variables; commercial software never has this freedom. Every customer is unique and has potential to expose an untested area.

Below figure gives the relative cost of fixing bugs at different stages of the SLDC.



Current measurements does not give real time data as to what is the impact of uncovering potential productions bugs. To assess the cost of production bugs, the bug needs to be uncovered in production and only after that the subsequent impact can be assessed. To measure production bugs real impact, there is a need for post release impact measurement, which is a lagging indicator and not real time.

2.2 Why measurement is required in software testing

Primary motivation and need for measurement, specifically software bug measurement, is to have a constant radar directing the software testing team and guide the efforts in the right direction.

Can we use the same technique for both, a new software under development and software that is under sustenance releases?

1. New software:
   For software, which is under development, a sprint over sprint measurement can be used as baseline
   - If a feature X is developed in a sprint N, sub sequent sprint (> N) treat feature X as pseudo production release and any bug detected for feature X to be treated as production bug.
2. Software in sustenance releases:
   Post a release, any customer reported bugs can directly be used as a measurement and also QA teams uncovering bugs, post release can also be treated a production bug

4 Quality Volatility

Volatility as a trend indicator has been used extensively in Stock Trading and almost always gives the prediction of where the stock market range exists.

In Derivative trading (Kotak Securities 2022), PUT and CALL (Downey 2022) open interests for either Index or a specific stock determines what is Put to Call Ratio (aka PCR) (Summa 2022). PCR is widely used to forecast market direction with Put/Call ratios. On a normal expiry week, PCR ratio between 0.10 to 10 determines the market Index range.

Taking inspiration from this well tested and established Trading strategy, we define QA Volatility which helps determine which direction the overall quality of a product is heading. With historic data being readily available, a clear uptrend or a down trend can be established. Despite all the improvement initiatives which are put in place for quality improvement, if the inflow of customer bugs does not show a decline, the initiatives are not in the right direction and Quality Volatility is the easiest approach to measure it.

4.1 Measuring Quality Volatility (QV)

In House Bugs (IHB):

Bugs uncovered during the development cycle by both Dev and QA can be called as In House Bugs (IHB). As soon as the product releases, the window to uncover IHB is complete.

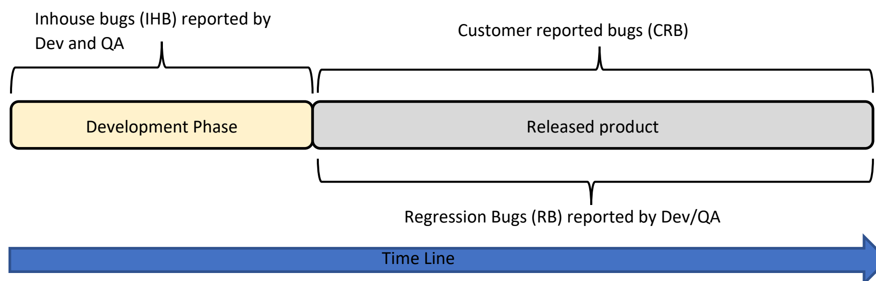
Customer Reported Bugs (CRB):

Once the product is released, any bugs reported by the customer from field is recorded under Customer Reported Bugs (CRB)

Production / Regression bugs:

Post release, if a bug is uncovered by Dev / QA team, these are to be categorized as Regression Bugs. These could be as a part of the code fix regression for an already reported bug by customer, and during regression testing new bugs were uncovered. However, the reason these being uncovered is because the In-House Testing phase did miss them.

Below figure depicts the timeline and the respective section of bugs reported.



Simple Quality Volatility can be defined as

$$Quality\ Volatility\ =\ \frac{Total\ number\ Customer\ Reported\ Bugs}{Total\ number\ of\ \ In\ House\ Bugs}$$

Here, the total count of all the reported bugs is considered. Irrespective of the priority of the bug, the weight associated with each bug is the same. A priority 5 (P5) bug is treated as same weight as priority 1 (P1) bug.

To make the Quality volatility more realistic, Weight needs to be associated with the respective priority of the bug reported. Weights are $\{P1 = 5, P2 = 4, P3 = 3, P4 = 2, P5 = 1\}$, where P1 is Priority 1 bug and so on.

With this approach, the Weighted Quality Volatility can be defined with the below equation.

$$Weighted\ Quality\ Volatility\ =\ \frac{CRB\ \{(5 *\ P1\ count) + (4 * P2\ count)\ + (3 * P3\ count)\ + (2 * P2\ count)\ + (1 * P5\ count)\}}{IHB\ \{(5 *\ P1\ count) + (4 * P2\ count)\ + (3 * P3\ count)\ + (2 * P2\ count)\ + (1 * P5\ count)\}}$$

5. How to use Quality Volatility to improve quality

Quality volatility is a ratio. This ratio represents stability of the product for the specific version for which it is measured. A series of these measurements help up build a near accurate trend of the overall quality of the product.

Quality volatility is lagging indicator (The Investopedia Team 2021). This ratio can be found when both data points i.e CRB and IHB numbers are available. IHB are recorded based on the bugs reported during development phase. For CRB, product must be released and we wait for a fair amount of time before the count of CRB is measured. As a thumb rule, we wait for the same time duration as much as the product was in development phase (Example: If the overall development was of 6 weeks, post release to measure CRB, ideal wait should be 6 weeks).

When we have a series of Quality Volatility ratios from multiple releases, it is possible to extrapolate the next value in the series, which will indicate the potential customer bugs, which may get logged once the product is released to market.

Below table helps us interpret Quality Volatility ratios and one of the possible corrective actions that could be taken. We have presented the corrective suggestion, we started with to improve the QV ratio.

Ratio	Status	What does it mean	Suggested recommendation
0 - 0.2	Green	Every 10 bugs found in house, after production we can expect 2 bugs	• Add more test cases • Increase test coverage
0.2 - 0.5	Yellow	Every 2 bugs logged in house, 1 bug was logged by customer.	• Review existing test cases • Add new test cases
0.5 - 1	Red	Every bug found in house; customer is matching the count	• Identify testing gaps • Increase coverage • Make customer logged bugs as part of regression testing • Increase manual as well as Automation coverage
> 1	Black	Customer is logging more bugs than in house.	• Current testing has shortcomings • Do architecture review of the features built and customer expectation • Identify customer user cases and build test scenarios for effective testing

When we applied the following model to track the Quality progress of the 5 different projects, we were able to see clear trend on how the quality was progressing and the risk areas. This becomes highly effective technique in simultaneous multi-product release cycles and aids Quality Leader to concentrate improvement areas in specific troublesome products.

Below is the representation of a Simple Quality Volatility Index. In this table all bugs have same weight

Release	Product 1			Product 2			Product 3			Product 4			Product 5		
	IHB	CRB	QA Volatility	IHB	CRB	QA Volatility	IHB	CRB	QA Volatility	IHB	CRB	QA Volatility	IHB	CRB	QA Volatility
N+4	20	7	0.35	25	9	0.36	10	4	0.40	4	1	0.25	10	4	0.40
N+3	12	6	0.5	15	8	0.53	9	4	0.44	12	1	0.08	13	2	0.15
N+2	15	16	1.07	17	22	1.29	4	8	2.00	3	8	2.67	10	7	0.70
N+1	15	13	0.87	26	24	0.92	7	9	1.29	18	10	0.56	14	8	0.57
N	25	16	0.64	20	23	1.15	9	15	1.67	7	19	2.71	18	10	0.56

In this case, release "N+4" is the latest release. Based on this model, measuring Quality Volatility for N, N+1, N+2, we observe there is a quality concern that needs to be addressed.

Below is the representation of a Weighted Quality Volatility Index, where bugs have weight assigned

Release	Product 1									Product 2								
	IHB				CRB				QA	IHB				CRB				QA
	Bl	Cr	Ma	Total	Bl	Cr	Ma	Total	Volatility	Bl	Cr	Ma	Total	Bl	Cr	Ma	Total	Volatility
N+4	5	7	8	20	2	2	3	7	0.35	1	9	15	25	3	2	4	9	0.41
N+3	5	3	4	12	1	3	2	6	0.47	2	4	9	15	1	4	3	8	0.57
N+2	2	7	6	15	3	8	5	16	1.11	0	4	13	17	2	7	13	22	1.40
N+1	4	5	6	15	2	8	3	13	0.88	3	4	19	26	5	16	3	24	1.11
N	5	11	9	25	5	6	5	16	0.67	2	3	15	20	4	17	2	23	1.40

Release	Product 3									Product 4								
	IHB				CRB				QA	IHB				CRB				QA Volatility
	Bl	Cr	Ma	Total	Bl	Cr	Ma	Total	Volatility	Bl	Cr	Ma	Total	Bl	Cr	Ma	Total	
N+4	2	1	7	10	1	1	2	4	0.43	1	2	1	4	0	1	0	1	0.25
N+3	2	2	5	9	1	2	1	4	0.48	0	0	12	12	0	1	0	1	0.11
N+2	2	1	1	4	2	6	0	8	2.00	0	0	3	3	2	5	1	8	3.67
N+1	1	2	4	7	1	7	1	9	1.44	0	2	16	18	1	9	0	10	0.73
N	6	1	2	9	1	13	1	15	1.50	0	3	4	7	6	12	1	19	3.38

Release	Product 5								
	IHB				CRB				QA
	Bl	Cr	Ma	Total	Bl	Cr	Ma	Total	Volatility
N+4	2	4	4	10	1	2	1	4	0.42
N+3	5	6	2	13	0	2	0	2	0.15
N+2	4	5	1	10	2	3	2	7	0.65
N+1	2	4	8	14	1	6	1	8	0.64
N	1	1	16	18	0	4	6	10	0.60

*The weights are Blocker = 5, Critical = 4, Major = 3

** Our observation, between Simple QV and Weighted QV, there is a very minor change in Volatility

5.2 Test case effectiveness

Test case(s) are designed to execute a specific code flow. During the test case execution, if this unique code flow is broken, the test case has uncovered a bug. The test case's potential to uncover a bug can be described as "Bug detection potential value factor".

Once the code is fixed for the bug uncovered, the code has become immune to the condition introduced by the test case and hence on subsequent execution of the test case, post fix, it is unlikely to break same code flow. Subsequent runs of the test case are regression run to ensure the code path is unbroken.

5.2.1 Defining Test case efficiency:

Every test case has an "Bug detection potential value factor". Over time, release over release if we do not uncover a bug, this value reduces. Every passing release (irrespective of how many times this test has been run within a given release, lets treat it at per release level), the value is deprecated. This follows a half-life pattern, which is very similar to that of a radioactive decay half-life equation (Cuemath 2022).

$$N_{(t)} = N_{(0)} * \left(\frac{1}{2}\right)^{\frac{t}{t_{\frac{1}{2}}}}$$

Where:

$N_{(0)}$ = Potential value of test case to find a bug during test design phase, where it is not even executed even once. The Value = "1" as each test case can find 1 bug for the code flow it is/was designed to test.

$N_{(t)}$ = Effectiveness retained of a test case at the end of "t" release execution.

t = Number of releases / executions that have been completed. In this case we refer to it number of releases that this specific test case has run.

$t_{(1/2)}$ = Time taken for test case to reach half its effectiveness. In software testing this is "1" as each release test case potential effectiveness is reduced by half.

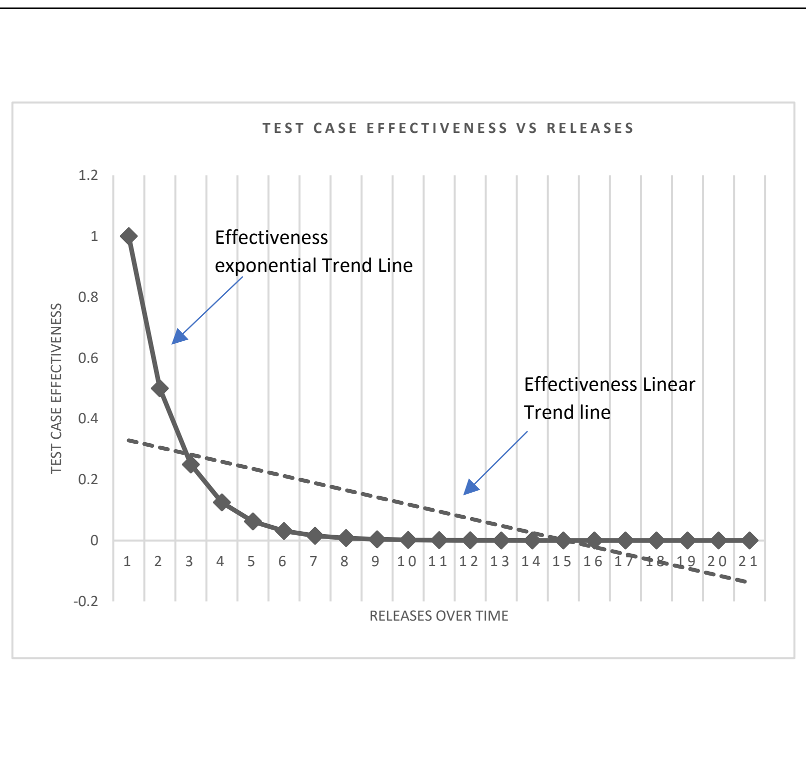
Using the values of: $N_{(0)} = 1$, t = 1 to n (where n is current release number), $t_{(1/2)} = 1$ in

$$N_{(t)} = N_{(0)} * \left(\frac{1}{2}\right)^{\frac{t}{t_{\frac{1}{2}}}}$$

*Assumption made: During maintenance release, no major code overhaul is done (under most circumstances, existing codes remains almost similar to previous release, regression testing is done to maintain quality status quo).

Below table shows calculated effectiveness of test case release over release when it fails to detect a bug. On the Right-hand side, you see the plot of the trend lines for both Exponential effectiveness and Linear Trend line of effectiveness

N(t) = Every subsequent release	Test case Effective ness	Probability of detection %
0	1.000000	100.000000
1	0.500000	50.000000
2	0.250000	25.000000
3	0.125000	12.500000
4	0.062500	6.250000
5	0.031250	3.125000
6	0.015625	1.562500
7	0.007813	0.781250
8	0.003906	0.390625
9	0.001953	0.195313
10	0.000977	0.097656
11	0.000488	0.048828
12	0.000244	0.024414
13	0.000122	0.012207
14	0.000061	0.006104
15	0.000031	0.003052
16	0.000015	0.001526
17	0.000008	0.000763
18	0.000004	0.000381
19	0.000002	0.000191
20	0.000001	0.000095



Observations:

- If a test case has not uncovered a bug in 10 releases, probability of it uncovering a bug is < 0.01%
- If a linear trend of effectiveness is followed, by 15 releases, a test cases which has not detected any bug in last 15 release, its potential effectiveness has fallen to 0.
- Best time to review a test case is after 10 release cycles. After 15 release cycles its already in RED zone.

Exception to the rule:

If a test case detects a bug in any subsequent runs, reset the value of test case equal to the number of releases it was run, and no bug was detected

Example: If a test case uncovers a bug in 10th release, whereas in previous 9 release it did not detect it. Change the "test case effectiveness" value from current value 0.000977 to 10.

The late-stage detection of a bug is a strong indicator a stable piece of code has been tweaked and has greater potential to break portions of the code which have remained stable in current / previous releases, potential quality RED flag.

*If tracking testcase effectiveness at a test case level becomes time consuming, this could be done "test case group"/ module level as well with same effectiveness.

5.2 How to be bring back the ratio < 0.2

Frist step in managing QA Volatility is understanding our current test approach and how to improve it.

If the ratio of QA Volatility is > 0.5, it is indicator of two major possibility what the high ratio can suggest.

- If it is a new project, high QA Volatility ratio indicates there are no adequate test cases developed yet
- For a project in sustenance mode, high QA Volatility indicate current test cases are ineffective and code is immune to current test set.

Following steps can be adopted to quickly introduce corrective actions in brining QA Volatility under control.

- Incorporate Exploratory testing (Softwaretestinghelp 2022) as part of regular test cycles. Exploratory testing is one of the quickest ways to uncover bugs
- Based on the "Test case effectiveness" technique described in previous section, identify test cases which need review
- Design review by QA: The best place to detect a bug is even before it is coded. Feature implementation discussion between Dev and QA helps a great deal to making this happen and helps sync Dev and QA to the same page about the feature under development
- For sustenance projects, last 3-6 release customer bugs should be integrated into regression cycles as Priority Set

Exception to the rule:

For a project under sustenance, if a module is stable and no field / customer bugs are reported for couple of releases. This indicates, QA teams' current approach has already covered all possible customer use cases. In such case, do not try to fix which is not broken.

6. Applying Quality Volatility for stabilizing projects which were stressed.

We have effectively used Quality Volatility ratio to help us identify products where quality was a concern and using some of the corrective methods described in the section above, we have been successful in reducing Quality Volatility ratio.

Below table presents data of IHB and CRB of 5 products Product 1 – Product 5, where based on the past trend of N, N+1 and N+2 release, these products were seeing an increase in customer reported bugs.

Using some of the corrective steps in the last two release i.e. N+3 and N+4 release, we have been able to reduce Quality Volatility < 0.5.

Release	Product 1			Product 2			Product 3			Product 4			Product 5		
	IHB	CRB	QA Volatility	IHB	CRB	QA Volatility	IHB	CRB	QA Volatility	IHB	CRB	QA Volatility	IHB	CRB	QA Volatility
N+4	20	7	0.35	25	9	0.36	10	4	0.40	4	1	0.25	10	4	0.40
N+3	12	6	0.50	15	8	0.53	9	4	0.44	12	1	0.08	13	2	0.15
N+2	15	16	1.07	17	22	1.29	4	8	2.00	3	8	2.67	10	7	0.70
N+1	15	13	0.87	26	24	0.92	7	9	1.29	18	10	0.56	14	8	0.57
N	25	16	0.64	20	23	1.15	9	15	1.67	7	19	2.71	18	10	0.56

With the corrective steps for N+3 and N+4 release, we are seeing higher IHB and there by better quality product delivered to customers which is directly proportional to reduced CRB, reducing the QA Volatility ratio.

7. Learning and key takeaways

Following are our key takeaways are learning

- To measure QA effectiveness, we do not need complex measurements to track and measure quality improvement
- Customer reported bug is the golden measurement to gauge quality of a product
- Code coverage may not be adequate to claim code is fully evaluated
- Exploratory testing and QA teams' involvement in feature design review are most cost-effective technique for early bug detection
- Code does become immune to testing over multiple releases
- Moving from "test to pass" to "test to break" testing mindset, is a must to uncovering the bugs which had high probability of being reported by the customer
- There is very minimum to negligible difference between Simple QV and Weight QV. For a start, we can begin with Simple QV to help identify concerning products

References

Airbrake. 2017. *Production Defects Are Not Inevitable.* June 16. Accessed April 1, 2022. https://airbrake.io/blog/devops/production-defects-are-not-inevitable.

Cuemath. 2022. *Half Life Formula.* Accessed June 10, 2022. https://www.cuemath.com/half-life-formula/.

Downey, Lucas. 2022. *Essential Options Trading Guide.* August. Accessed Aug 2022. https://www.investopedia.com/options-basics-tutorial-4583012#:~:text=Call%20and%20Put%20Options,-Options%20are%20a&text=If%20you%20buy%20an%20options,right%20to%20sell%20a%20stock.

Fishman, Charles. 1996. *They Write the Right Stuff.* December 12. Accessed April 5, 2022. https://www.fastcompany.com/28121/they-write-right-stuff.

Kotak Securities. 2022. *What Is Derivative Trading.* Accessed April 5, 2022. https://www.kotaksecurities.com/ksweb/Research/Investment-Knowledge-Bank/what-is-derivative-trading.

Pal Kienitz, One Beyond. 2019. *https://www.one-beyond.com/how-we-measure-software-quality/.* December 24. Accessed March 2, 2022.

Satyabrata Jena. 2021. *Measuring Software Quality using Quality Metrics.* August 12. Accessed March 2022, 4. https://www.geeksforgeeks.org/measuring-software-quality-using-quality-metrics/.

Sealights.io. 2022. *Measuring Software Quality: A Practical Guide.* Accessed March 5, 2022. https://www.sealights.io/software-quality/measuring-software-quality-a-practical-guide/.

Softwaretestinghelp. 2022. *What Is Exploratory Testing In Software Testing (A Complete Guide).* August 7. Accessed Aug 10, 2022. https://www.softwaretestinghelp.com/what-is-exploratory-testing/ .

Summa, John. 2022. *Forecasting Market Direction With Put/Call Ratios.* May 15. Accessed April 15, 2022. https://www.investopedia.com/trading/forecasting-market-direction-with-put-call-ratios/.