Software Testing Configuration Drift Principles – An Analysis

*** Meenakshi Rathod, Assistant Professor, Dept of Computer Science, Govt First Grade College, Shapur.**

****Sampatkumari.M.Bandagar, Assistant Professor and HOD of Computer Science, Govt First Grade College, Humanabad.**

*****Dr.Basavaprasad.B. Assistant Professor and HOD of Computer Science, Govt First Grade College, Raichur.**

Abstract

This paper attempts to study s**oftware testing** that can be stated as the process of verifying and validating that a software or application is bug free, meets the technical requirements. At a high level, we need to make the distinction between manual and automated tests. Manual testing is done in person, by clicking through the application or interacting with the software and APIs with the appropriate tooling. This is very expensive as it requires someone to set up an environment and execute the tests themselves, and it can be prone to human error as the tester might make typos or omit steps in the test script.Automated tests, on the other hand, are performed by a machine that executes a test script that has been written in advance. These tests can vary a lot in complexity, from checking a single method in a class to making sure that performing a sequence of complex actions in the UI leads to the same results. It's much more robust and reliable than automated tests – but the quality of your automated tests depends on how well your test scripts have been written. Automated testing is a key component of continuous integration and continuous delivery and it's a great way to scale your QA process as you add new features to your application. But there's still value in doing some manual testing with what is called exploratory testing as we will see in this guide. Unit tests

Key words: End-to-end testing, verification, validation, QA process, Automated testing

Introduction

Much of this wasted time often happens in one specific scenario that most development teams should be familiar with. A bug is discovered in production and developers find that they don't have the right information in their log files to diagnose the issue. Teams go through an iterative cycle of pushing new builds with new log lines and eventually opt to deploy the current production build to a remote test or staging environment with hopes of being able to more easily reproduce the issue. Sometimes this might work and other times the environment is just too different to adequately reproduce the issue and it goes unresolved. In the rest of this article, we'll take a look at a few reasons why you shouldn't reproduce issues in remote environments. It verifies that various user flows work as expected and can be as simple as loading a web page or logging in or much more complex scenarios verifying email notifications, online payments, etc. End-to-end tests are very useful, but they're expensive to perform and can be hard to maintain when they're automated. It is recommended to have a few key end-to-end tests and rely more on lower level types of testing (unit and integration tests) to be able to quickly identify breaking changes. Acceptance tests are formal tests executed to verify if a system satisfies its business requirements. They require the entire application to be up and running and focus on replicating user behaviors. But they can also go further and measure the performance of the system and reject changes if certain goals are not met. Performance tests check the behaviors of the system when it is under significant load. These tests are non-functional and can have the various form to understand the reliability, stability, and availability of the platform. For instance, it can be observing response times when executing a high number of requests, or seeing how the system behaves with a significant of data. Performance tests are by their nature quite costly to implement and run, but they can help you understand if new changes are going to degrade your system. Smoke tests are basic tests that check basic functionality of the application. They are meant to be quick to execute, and their goal is to give you the assurance that the major features of your system are working as expected. Smoke tests can be useful right after a new build is made to decide whether or not you can run more expensive tests, or right after a deployment to make sure that they application is running properly in the newly deployed environment.

Objective:

This paper intends to explore and analyze **Software Testing** ; process of evaluating the functionality of a software application to find any software bugs. It checks whether the developed application is as per elicited software requirement specification

Software Testing Configuration Drift

As much as we try to keep development and test environments perfectly aligned with the same configuration as production, configuration drift almost always happens. Most organizations today realize that processes such as infrastructure as code and continuous deployment can cut configuration drift down tremendously, but even with strict practices in place configuration drift is often unavoidable. Even things as small as package updates to a server, differences in memory and CPU configurations, or manual changes made by developers or testers on a server can cause drift.

Instead of spinning up and deploying applications to remote test environments in order to identify the root cause of a defect, it's much easier to inspect the state of the application while it's running in its native environment in order to gather clues that can lead to resolving the issue. Production debugging tools allow teams to resolve defects faster, where they happen without having to worry about reproducibility in a separate environment.

Inefficient and Time Consuming

Today many organizations have automated build and deployment processes that allow for automated creation or tear down of environments, but many still do not. Regardless of which side of the fence you're on, spinning up new environments can be time consuming. When development teams need to stop what they're doing and wait for new environments to be spun up, context switching happens which results in reduced productivity. Deploying new builds into these environments can take anywhere from minutes to hours, or even days if approvals are required and multiple teams need to get involved.

In addition to the time needed to spin up these environments, they can also be costly in terms of cloud or infrastructure spend required to run these environments for extended periods of time. In general, most organizations aren't spinning up exact duplicates of production environments due to cost constraints which can mean certain hard to reproduce environment specific bugs may still go unresolved in these remote environments.

Those Hard to Reproduce Bugs

We've probably all encountered them at one point or another. You discover a bug in production, gather as many details from the logs as you can, try to reproduce the issue in a test environment but at the end of the day come up empty handed. The defect gets marked as "not reproducible". We hear it all the time from customers. Some bugs go unresolved due to the fact that you can't get the information you need while the application is running in production and when trying to reproduce the issue in test environments, the bug doesn't reproduce.

It's not uncommon for software development organizations to have bugs in their production environments that go unresolved for months or even years due to the inability to reproduce them. For these scenarios it's critical to have proper tooling which allows you to debug applications on demand without requiring code changes or redeployments of your application.

Data Constraints

When attempting to reproduce an issue across multiple environments, one area that teams must have solid processes around is test data management. Test data can be critical in the reproduction of bugs in that if you don't have the right test data in your environment, the bug may not be reproducible. Due to the sheer size of production data sets, teams must often work with subsets of that data across test environments. The holy grail of test data management processes is to allow teams to easily quickly subset production data based on the data needed to reproduce an issue.

In practice, things don't always work out so easily. It's hard to know what attributes of your test data may be influencing a specific bug. In addition, data security when dealing with PII data can be a major challenge when subsets of data are used across environments. Teams need to ensure that they are in compliance with corporate data privacy standards by masking or generating new relevant data sets.

Many times it takes lots of logging and hands on investigation to uncover how data discrepancies can cause those hard to find bugs. If you cannot easily manage and set up test data on demand, teams will suffer the consequences when it comes to trying to reproduce bugs in remote environments.

The Right Tools for the Job

They say when your only tool is a hammer, all problems start looking like nails. Thinking outside of the box and having the right tools for the job tends to make life a little bit easier. When attempting to debug a production issue, the standard tool set that comes to mind for most organizations are APM, logging, and exception management tools. While these tools are all useful and have their time and place to be used, when developers want to dive into their code running in production these tools just don't provide the level of detail needed to get to the bottom of those hard to find bugs. Production debugging tools allow developers to get to a deeper level of detail that APM tools just don't allow capturing. They also allow collection of on demand logs or snapshots of data from applications but unlike traditional logging solutions, it can be done all without writing a line of code or redeploying the application. Having a production debugging solution in place can often allow developers to find a defect right when and where it's happening without having to go through the extra effort of spinning up a new environment

Principles of Testing:-

(i) All the test should meet the customer requirements
(ii) To make our software testing should be performed by third party
(iii) Exhaustive testing is not possible.As we need the optimal amount of testing based on the risk assessment of the application.
(iv) All the test to be conducted should be planned before implementing it
(v) It follows pareto rule(80/20 rule) which states that 80% of errors comes from 20% of program components.
(vi) Start testing with small parts and extend it to large parts.

Types of Testing:-

1. Unit Testing

It focuses on smallest unit of software design. In this we test an individual unit or group of inter related units.It is often done by programmer by using sample input and observing its corresponding outputs. **Example:**

a) In a program we are checking if loop, method or

function is working fine

b) Misunderstood or incorrect, arithmetic precedence.

c) Incorrect initialization

2. Integration Testing

The objective is to take unit tested components and build a program structure that has been dictated by design.Integration testing is testing in which a group of components are combined to produce output.

Integration testing are of two types: (i) Top down (ii) Bottom up **Example**

(a) Black Box testing:- it is used for validation.

In this we ignores internal working mechanism and

focuses on output.

(b) White Box testing:- it is used for verification.

In this we focus on output and ignores on internal

mechanism

3. Regression Testing

Every time new module is added leads to changes in program. This type of testing make sure that whole component works properly even after adding components to the complete program. Example

In school record suppose we have module staff, students and finance combining these modules and checking if on integration these module works fine is regression testing

4. Smoke Testing

This test is done to make sure that software under testing is ready or stable for further testing It is called smoke test as testing initial pass is done to check if it did not catch the fire or smoked in the initial switch on.

Example:

If project has 2 modules so before going to module

make sure that module 1 works properly

5. Alpha Testing

This is a type of validation testing.It is a type of *acceptance testing* which is done before the product is released to customers. It is typically done by QA people.

Example:

When software testing is performed internally within

the organization

6. Beta Testing

The beta test is conducted at one or more customer sites by the end-user of the software. This version is released for the limited number of users for testing in real time environment

Example:

When software testing is performed for the limited

number of people

7. System Testing

In this software is tested such that it works fine for different operating system.It is covered under the black box testing technique. In this we just focus on required input and output without focusing on internal working. In this we have security testing, recovery testing , stress testing and performance testing Example:

This include functional as well as non functional testing

8. Stress Testing

In this we gives unfavorable conditions to the system and check how they perform in those condition. Example:

(a) Test cases that require maximum memory or other

resources are executed

(b) Test cases that may cause thrashing in a virtual

operating system

(c) Test cases that may cause excessive disk requirement

9. Performance Testing

It is designed to test the run-time performance of software within the context of an integrated system.It is used to test speed and effectiveness of program.
Example:

Checking number of processor cycles. One common misconception is that JUnit is exclusively a Unit Testing Tool.

JUnit is used for Unit Testing, because it is a 'code execution framework with built in assertions and supporting features for data driven iteration'.

- **code execution framework**, meaning that we don't have to 'build' an application and the @Test code has access to any of the code instantiated as it executes, e.g. classes we write. When used for Unit Testing, these are the classes in the application itself. When used for Integration Testing we use the classes we've written as Abstraction Layers, Domain Objects, and the libraries we have imported to support our execution (WebDriver, REST Assured etc.). But the 'thing' we are executing against is a separate application, rather than an object in memory.
- **built in assertions**, meaning we write code that 'fails' if the condition we expect to be present is not met during the execution
- **features for data driven iteration**, meaning we can separate variant data from the invariant path encoded in the @Test and reuse the @Test rather than create a 'loop' that reads data and feeds it into a set of sequential code JUnit, TestNG, and even Cucumber, can be viewed as execution frameworks which we harness to automate code and applications effectively. We choose what is most appropriate for the process, tooling, abstraction layer, automation approach, and experience we have. The tooling does not dictate what we do with it.

Codeless Software Testing

- **Save time and resources**: Codeless testing means that test cases can be created in minimum time and executed quickly. This means that time previously assigned to your QA testers to code test scripts can be reallocated towards other important tasks.

- **Improve time to market**: With codeless QA testing, testers now have plenty of time to focus on the outcome of testing rather than conducting test practices repeatedly. The best codeless test automation tools feature excellent integration abilities designed to simplify test creation and maintenance, further ensuring no delays in product deployments.

- **Easily create test cases**: Because codeless automation testing tools don't require coding skills to create test scripts, the process of creating test cases is approachable for any QA tester. These tools often showcase an intuitive user interface without the intimidation a script development environment presents.

- **Decrease your testing bottleneck**: Since codeless automated testing speeds up the process of creating test cases, QA teams can run through test cases and deliver results at a more accelerated rate. No longer are developers delayed in resolving application issues because of lag in testing.

- **Simple to maintain and manage test cases**: A codeless automation framework simplifies the process of updating test scripts whenever requirements change or whenever product enhancements are enabled. The best codeless test automation tools follow a well-defined structure and architecture so that your framework is properly managed over time.

- Quick access to testing reports: Collecting test performance data is a must for any test automation checklist. However, writing custom scripts to create these reports takes up time and resources. Fortunately, codeless automation testing tools often feature customized reporting that's easy to gather, understand and explain to stakeholders.

Conclusion

To begin with, conducting tests in an unrealistic testing environment would be the most common recurring problem we have seen. If we're interested in the performance of a certain system that should be deployed to multiple powerful machines, we cannot make reasonable assumptions about the resulting setup from a performance test done on a local laptop. Then, when the system is deployed in production, the results are completely different. As funny as it sounds, people tend to do that more often than one would think. The above-mentioned Software Testing Types are just a part of testing. However, there is still a list of more than 100+ types of testing, but all testing types are not used in all types of projects. So I have covered some common Types of Software Testing which are mostly used in the testing life cycle. Also, there are alternative definitions or processes used in different organizations, but the basic concept is the same everywhere. These testing types, processes, and their implementation methods keep changing as and when the project, requirements, and scope changes.

References

1. "An Introduction to Dew Computing: Definition, Concept and Implications - IEEE Journals & Magazine".
2. Montazerolghaem, Ahmadreza; Yaghmaee, Mohammad Hossein; Leon-Garcia, Alberto (September 2020). "Green Cloud Multimedia Networking: NFV/SDN Based Energy-Efficient Resource Allocation". IEEE Transactions on Green Communications and Networking. 4 (3): 873–889. doi:10.1109/TGCN.2020.2982821. ISSN 2473-2400.
3. The NIST Definition of Cloud Computing NIST
4. Wang (2012). "Enterprise cloud service architectures". Information Technology and Management. 13 (4): 445–454. doi:10.1007/s10799-012-0139-4. S2CID 8251298.
5. "What is Cloud Computing?". Amazon Web Services. 2013-03-19. Retrieved 2013-03-20.
6. Baburajan, Rajani (2011-08-24). "The Rising Cloud Storage Market Opportunity Strengthens Vendors". It.tmcnet.com. Retrieved 2011-12-02.
7. Oestreich, Ken (2010-11-15). "Converged Infrastructure". CTO Forum. Thectoforum.com. Archived from the original on 2012-01-13. Retrieved 2011-12-02.
8. Ted Simpson, Jason Novak, Hands on Virtual Computing, 2017, ISBN 1337515744, p. 451
9. "Where's The Rub: Cloud Computing's Hidden Costs". 2014-02-27. Retrieved 2014-07-14.
10. Steven Levy (April 1994). "Bill and Andy's Excellent Adventure II". Wired.
11. White, J.E. "Network Specifications for Remote Job Entry and Remote Job Output Retrieval at UCSB". tools.ietf.org. Retrieved 2016-03-21.
12. Griffin, Ry'mone (2018-11-20). Internet Governance. Scientific e-Resources. ISBN 978-1-83947-395-1.
13. "July, 1993 meeting report from the IP over ATM working group of the IETF". CH: Switch. Archived from the original on 2012-07-10. Retrieved 2010-08-22.
14. Corbató, Fernando J. "An Experimental Time-Sharing System". SJCC Proceedings. MIT. Archived from the original on 6 September 2009. Retrieved 3 July 2012.
15. Levy, Steven (April 1994). "Bill and Andy's Excellent Adventure II". Wired.
16. Levy, Steven (2014-05-23). "Tech Time Warp of the Week: Watch AT&T Invent Cloud Computing in 1994". Wired. AT&T and the film's director, David Hoffman, pulled out the cloud metaphor–something that had long been used among networking and telecom types. [...]