Volume 5, Issue 5, May 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Automated Software Testing using Test Orchestration System Based on Engineering Pipeline

Rashmi Singh[*]
IT Department, W.C.E., Sangli
India

S. P. Sonavane
IT Department, W.C.E., Sangli
India

Samir Inamdar
TIBCO Software Inc.
India

Abstract— In today's highly competitive software application market, it is crucial for a software vendor to respond very quickly to customers frequently changing needs. This has led to the adoption of iterative software development methodologies using frequent customer feedback. Without rigorous testing, this agility of development could leave the software vulnerable to defects. Testing by manual processes is infeasible in the above scenario because of the large time and cost requirements. Thus, test automation is the only viable recourse to achieving software quality in a time bound manner. This paper describes a methodology to achieve continuous testing during the software development process that provides immediate feedback to the software developer. The Test Orchestration System (TOrch) provides for a pipeline of test stages that perform the full complement of various test types. At each test stage, Torch first runs the specified tests, analyzes the test output and generates reports. A software quality analysis tool, namely SonarQube is also integrated with TOrch to measure the quality of the software being produced by comparing the product against predefined coding standards and rules.

Keywords— Continuous Integration, Engineering Pipeline, SonarQube, Test Orchestration.

I. Introduction

In today's highly competitive software market, it is crucially important to provide an out-of-the-box high quality product with minimal defects. Not only is it important to introduce high quality products but also to maintain its quality through the successive agile release cycles, under the constraints of limited time and resources. Established and widely prevalent practices still rely significantly on manual testing, which is slow and expensive. Test automation provides the only viable method to achieve this goal.

With traditional test automation a few automated tests do exist, but they are often poorly maintained and out-of-date and require supplementing with extensive manual testing. This results in defects detection much later in the product life cycle or they end up getting discovered by end users. With new techniques like test driven development it is possible for test automation to closely follow the software development cycle. Now it becomes possible to follow an end to end continuous testing methodology, wherein various test types, from unit tests, to integration tests, to system tests can all be executed, analysed and reported in an integrated manner. The Engineering Pipeline specifies and realizes this concept and changes the very nature of the software test automation process.

This paper proposes the idea of the Test Orchestration system (TOrch) that sequences and coordinates various stages of the automated testing process. In addition this system provides enhanced notifications of test results, significantly reducing the conventional communication overhead between development and test teams. Each stage of the pipeline executes various functions namely test execution, test results analysis, reporting and notifications. Another important purpose of TOrch is to provide visibility to all stakeholders of the software application. Also, by providing context sensitive information in case of test failures, TOrch enables easier debugging of the code. Additionally, it provides on going visibility into the current level of product software quality through the integrated use of SonarQube.

Enterprise statistics demonstrate that on average, 80% of the cost of software is spent on maintenance. Hence we can say that internal quality is a key component for the future cost of the software. Therefore, inspecting quality of software applications has become a major concern for any company that is involved in building software. Traditional approach to managing code quality is to test the code once in a while, basically at the end of a development phase. In the best case this approach leads to delays and re-work while in the worst case, this approach leads to the shipment of low-quality, expensive–to-maintain software. Continuous Inspection of code is a new approach in code quality management that clearly transfers ownership of the code quality back to the developer; one that accentuates quality throughout the development phase and has a shorter feedback loop to ascertain rapid resolution of quality problems. The main idea in continuous inspection is finding issues early-when fixing them is quiet inexpensive and easy.

Continuous Inspection using Test Orchestration is an approach that builds in quality from the start, rather than considering it after the fact. Quality tools integrated with TOrch, overcome the quality issues like unexpected delays, unplanned rework or missing features by continuous inspection of code from the start that are caused by traditional approaches. In software companies, these are the main causes that reduce the team's productivity. Such code inspection tools provide ongoing guidance and reminders to the developer to improve code quality and follow prescribed best coding practices. This reduces quality issues that typically emerge during the software development life cycle.

SonarQube [3] is internally integrated in TOrch and checks the internal quality of the code continuously against seven Axes of Quality and gives moment-in-time snapshots of code related issues. Not only does it show what's going wrong, also provides quality-management tools to actively help put it right. It is now very easy for the developer involved in building software to see several reports simultaneously in a consolidated screen. Examples of quality tools include open source products such as PMD, Check style, FindBugs and SonarQube for code analysis and test coverage. TOrch uses Jacoco for test coverage analysis.

This paper is organized as follows: Section I introduces Test Orchestration. Section II discusses the limitations of manual testing and contains a literature survey. Section III gives a brief introduction of the proposed system and its design. Section IV describes results from sample test runs conducted using TOrch. Section V presents the conclusion and provides directions for future work.

II. RELATED WORK

Automation, in the field of testing and software deployment has gained new significance in the context of fast iteration and agile software development methodology. It integrates testing into the development process to provide the developer with fast feedback on the quality of his output. Agile teams use a "whole-team" approach to "bake in quality" into the software product. This approach allows testers to cooperate actively with the development team, giving them a capability to identify issues and guide the design/code corrections.

Current research in the field of test automation is focusing on making the testing directed, efficient and more intelligent. The various types of automated testing has been discussed in several research works. Jez Humble and David Farley [2] has discussed the Continuous Integration techniques, Extreme programming, Engineering Pipeline technique and its effect on design and development process of software product.

G.Ann Campbell and Patroklos P. Papapetrou [3] published a book SonarQube IN ACTION in 2014, which described how to use SonarQube quality platform and how to help development team to continuously improve their code quality.

Karhu et al. [4] said that "It is possible to reduce the cost of software testing when it performs in automated manner. Automated software testing can also improve quality because of more testing in less time, but it includes extra efforts in, eg implementation, execution, maintenance, and training". The team also explains that "automated testing systems consist of hardware and software and suffer from the same issues as any other systems", tragically it is not explained on what these issues are; with the exception of saying lack of quality problems.

Berner et al. [5] study and analyze several projects and report issues with test environment use and test environment design. During the findings process faced with absence of documentation, absence of test framework and a general lack of understanding that a test automation organization deliver a piece of software that needs to be verified itself and maintained.

Gelperin and Hayashi [6] said, in a paper explaining the "Testability Maturity Model" (TMM) that, the main issue with test system is reuse and that to empower reuse. One must support the test framework by configuration and management, modularization and independence. It is also suggested that the test environment should be automated, adaptable and simple to utilize.

Karlstrom at al. [7] proposed a test practice framework valuable for small emerging organizations, and stresses the accessibility of the test environment when it is required. The paper draws from various case-studies that demonstrates the significance of test environment taking care of. One success factor that is brought up is a quick test environment with a known and determined design. Another important perception is that the test environment itself must be verified upon setup and also that this is the hardest piece to get right because the complexity of this is typically underestimated by developers.

"TAIM Test Automation Improvement Model" [8] by Sigrid and Ericson et al. talks about ten key areas that include Test management, Test Requirements, Test specification, Test code, Test Automation Process, Test Execution, Test Environment, Test Tools and Defect Handling. The key areas address what the test product does and how to handle the test product. In addition this framework also explains other relevant ideas of traceability and defined measurements such as efficiency, effectiveness and costs.

To summarize, the literature survey found a lack of clear, detailed, and easy-to-use guidelines on the design, implementation, and maintenance of an automated test execution system. None of the surveyed test automation frameworks provided means to keep the aggregated technical debt to a satisfactory level. In addition, the surveyed test frameworks were found to be difficult to maintain and required supplementing with extensive manual testing.

III. PROPOSED APPROACH

TOrch is based on the idea of a software testing Engineering Pipeline. Basic Engineering Pipeline is shown in figure 3.1. The pipeline consists of a number of connected stages, with each stage responsible for all process related to a test type. Within a stage the various processes associated with test execution are arranged in a logical sequence for execution. The first stage of the pipeline compiles the code, run unit tests, performs code examination, creates installers, produces unit test outcomes and sends detailed report to the team. In TOrch, there is an orchestrator, whose work is to establish the effective coordination between the processing elements at phase level as well as stage level.

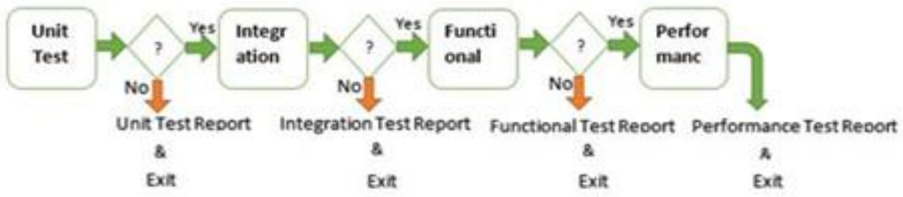

Unit Test
?
Yes
No
Integr ation
Functi onal
Perfor manc
Unit Test Report & Exit
Integration Test Report & Exit
Functional Test Report & Exit
Performance Test Report & Exit

Fig 3.1: Simple Engineering Pipeline

A. DESINE

In TOrch, testing can be triggered in three different ways:

- Svn poller.
- TOrch Dashboard
- Eclipse plug-in

In first case, process starts with the developers commenting changes into their repository. Orchestrator reacts to the commit by triggering a fresh instance of Engineering Pipeline. Svn poller which is a java application. It takes svn log by querying svn, creates unique id along with information like user name, comment and file name. Then svn poller puts the id in test queue and all relevant information in database. The UI and eclipse plug-in are under development.

TOrch uses dedicated machine for each type of tests. It also keep track of idle and occupied machines. To execute particular type of test like Junit, TOrch first search for the availability of the machine. If related test machine is idle then id is retrieved from queue. Based on that id, orchestrator queries database to get all details for that id and puts the id in the test job queue. In provisioning, svn update and set up preparation steps are performed on respective machines. These functionalities are called through REST calls. After test execution, log analyser will analyse logs and all tests results are stored in database. The report summary mail is then sent to respective developer along with the detailed report link.

Torch uses TIBCO Business Work (BW) for orchestration, which is the key component in TOrch. Orchestrator (BW) decides the notion of test pipeline. Additionally it control the execution of different tests like Unit test, Integration test, functional test and performance test.

If first stage of the test pipeline executed successfully and code is behind to scratch then only it will be forwarded to the next stage. The second stage is for integration test. This stage will be trigger after successful execution of first stage.

The third and fourth stage will be triggered in similar ways. At the end of this stage team are able to see the report of current stage as well as the status of the previous too stages. At the end of pipeline mail notification are send to all team members. In TOrch, each change will propagate through the pipeline instantly as shown in Figure 3.2.

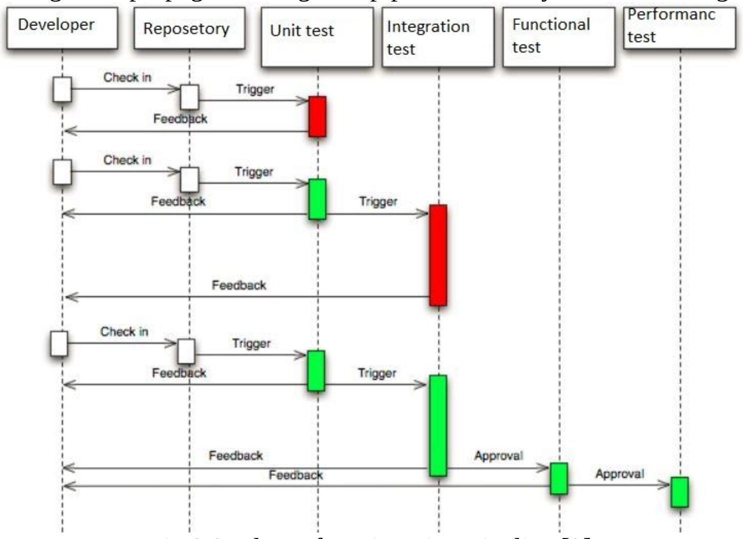


Fig 3.2: Flow of Engineering Pipeline [2]

TABLE 3.1: COMPARISONS WITH SONAR

	FindBugs[9]	**PMD[10]**	**Checkstyle[11]**	**Sonar[4]**
License	Lesser GNU Public License	BSD-style license	Lesser General Public License	GNU Lesser Public License
Purpose	Potential Bugs: Find bugs in Java byte code.	Bad Practices: Looks for potential problems, possible bugs, unused and sub-optimal code and over- complicated expressions in the Java source code	Conventions: Scans source code and looks for coding standards, e.g. Sun Code Conventions, JavaDoc	Duplication Scan: source code and looks for copy paste.

Strengths	1. Finds often real defects 2. Low false detected rates 3. Fast because byte code 4. Less than 50% false positive	1. Finds occasionally real defects 2. Finds bad practices	1. Finds violations of coding conventions	1. Find duplicate code across file level and project level. 2. Many more plugins is available
Weaknesse s	1. It is not aware of the sources. 2. It needs compiled code	- slow duplicate code detector	1. It can't find real bugs	1. Very few rules
Number of rules	408	234	132	113

B. CONTINUOUS INSPECTION

In the absence of better code analysis tool, the quality of code degrades over a period of time, which in turn makes software maintenance very expensive. Also, traditional approach to code quality control is to test the code manually at the end of development phase. In best case, this methodology prompts rework and delay. While in most pessimistic scenario, it prompts the production of low quality code.

This Test Orchestration system is designed keeping in mind above facts also. Static code quality improvement tools like PMD, Checkstyle, SonarQube and FindBugs are widely used in this system. Integration of each individual quality tools for their own specific purpose. Each has an own merits and demerits. Important features of tools are describe in table 3.1.The most excellent way to use all these tools together is to use it along with sonar. SonarQube is the main component to manage code quality. It also offering visual reporting on and across projects. These tools scan the source code for anti-patterns against seven axes of quality as shown in Figure 3.3 and report each instance as an issue. All issues are not bugs, but every issue needs further consideration. Code analyzer analyses source code based on set of rules.

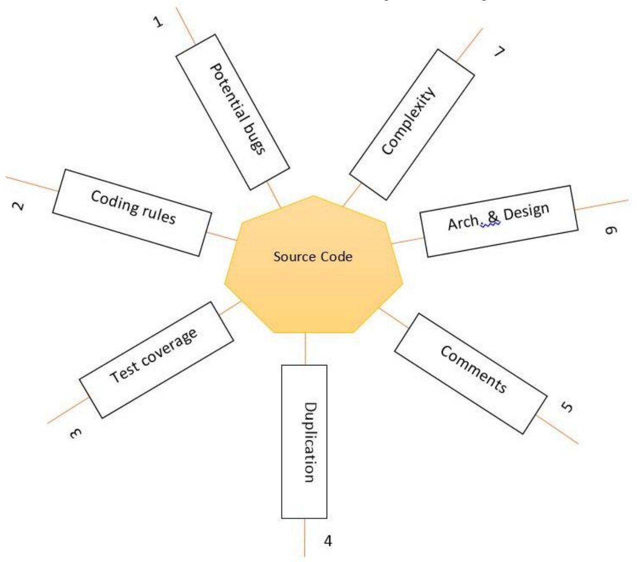


Fig 3.3: Seven quality axis

Most of the rules are inherited from some other open source quality tools like FindBugs, Checkstyle, PMD, SonarQube and Jacoco. These rules are packed into profiles and for each language it performs analysis. Code analysis is triggered by Jenkins job but internally this job prefers sonar-runner and a basic properties file. The flow of code analysis is shown in Figure 3.4. Flow starts with the code check-ins by developers.

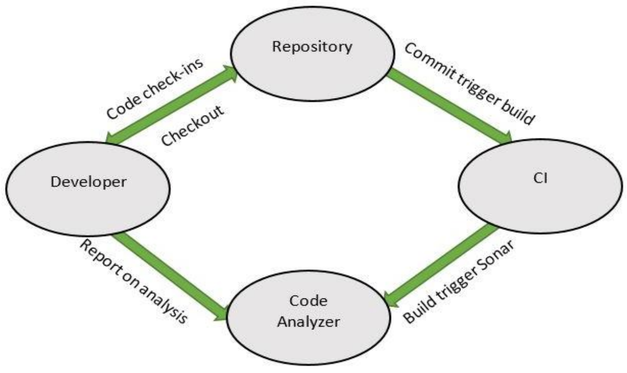


Fig 3.4: Flow of code analysis

IV. RESULTS

This section contains results of test execution and code analysis. Figure 4.1 shows the execution of test stages in pipeline fashion. Main process in each stage having various sub process. There is a timer associated with the pipeline which automatically initiate the execution of pipeline after fix time interval. For example; somebody want to execute this pipeline after every three hours of intervals in a day. For this, simply they have to set the timer value three and it will trigger the pipeline on every three hours. Successful execution of one stage of pipeline will trigger the next stage immediately without any external intervention.

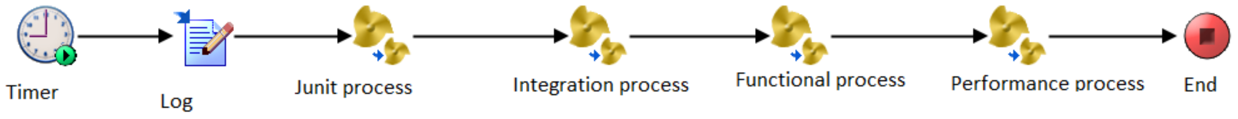
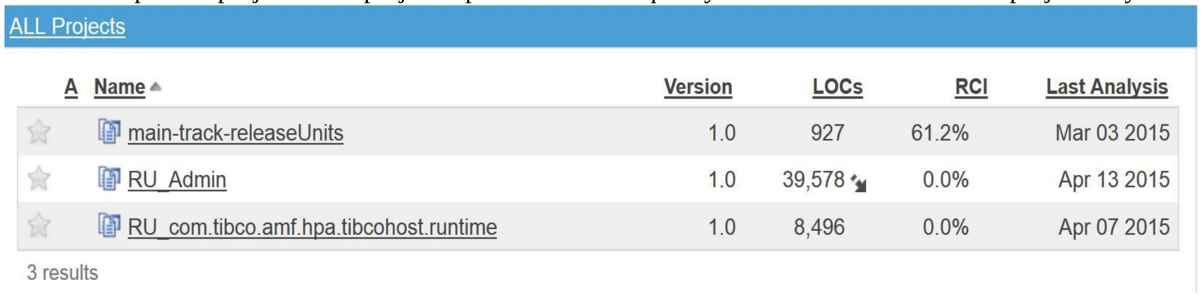


Fig 4.1: Flow of test execution

Figure 4.2 shows main dashboard of the code analyzer. It contains list of project under analysis with some basic information like number of LOC, version number, and last analysis date and time. Detailed view of the report is visible upon click on respective projects. The project report contains all quality information related to that project only.



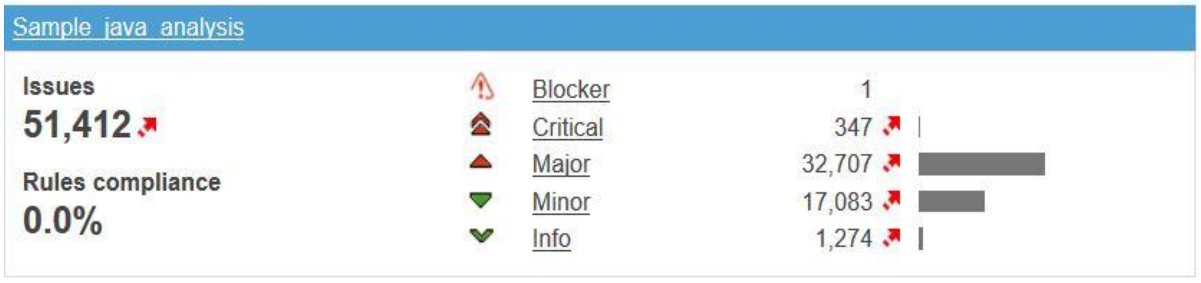
ALL Projects

A	Name	Version	LOCs	RCI	Last Analysis
☆	main-track-releaseUnits	1.0	927	61.2%	Mar 03 2015
☆	RU_Admin	1.0	39,578	0.0%	Apr 13 2015
☆	RU_com.tibco.amf.hpa.tibcohost.runtime	1.0	8,496	0.0%	Apr 07 2015

3 results

Fig 4.2: Main dashboard

Figure 4.3 shows issues reports of code analysis. Based on violation of Checkstyle, PMD, FindBugs and Sonar rules, number of issues is identified. These issues are categorizes as blocker, critical, major, minor and info.
All issues are not bugs but critical and blocker issues needs attention like file open, null values.



Sample java analysis

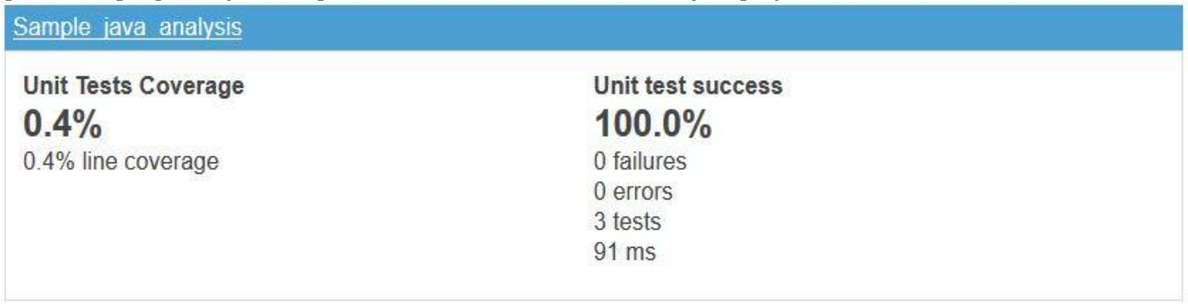
Issues
51,412

Rules compliance
0.0%

Blocker	1
Critical	347
Major	32,707
Minor	17,083
Info	1,274

Fig 4.3: Issues report

Figure 4.4 shows a unit test coverage report which tells unit test coverage percentage. It also shows number of test cases, number of failed cases, errors and time taken to execute the all test cases. Percentage in line coverage explain how many lines of code is covered by test program and which line is uncovered or untouched. So that it is possible to write appropriate test program by developer that will test whole functionality of project.



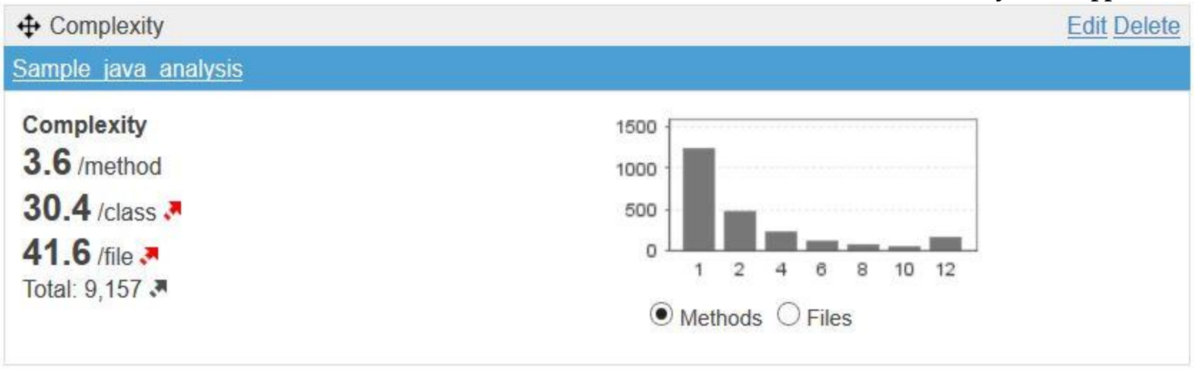
Sample java analysis

Unit Tests Coverage
0.4%
0.4% line coverage

Unit test success
100.0%
0 failures
0 errors
3 tests
91 ms

Fig 4.4: Coverage report

Figure 4.5 shows complexity report. The report tells about the distribution of complexity of the program that is being analyzed. Complexity is evaluated at method level, class level, and file level. It also shows overall complexity of the project. In ideal scenarios this bar graph should be weighted to left. Left weighted bar graph shows that the code is well maintained. The more complex a program is, the more difficult it becomes to maintain.



Complexity
3.6 /method
30.4 /class
41.6 /file
Total: 9,157

[● Methods] ○ Files

Fig 4.5: Complexity report

V. Conclusions and Future Work

Continuous delivery of a software in an industry requires frequent testing of different types. Also to detect problems early in development there is need for the new approach that speedup testing. Test Orchestration is an attempt to go beyond traditional test automation. In Test Orchestration there is notion of Engineering Pipeline where different types of tests executes one after another. Static code analysis tools are tightly integrated with the Test Orchestration system which runs with the pipeline as a backend process and produce all quality information related to code and test-coverage. At the end of test execution and code analysis Test Orchestration system will send the report link to the respective developer. These types of test automation and code analysis is required for any software organization which guide team throughout the development to produce quality software. On successful execution of test phases Test Orchestration System produce detailed report of each and every phase separately and send the link of report to the respective development team.

Today Test Orchestration system uses static machine for test execution. It may be the future plan to make Test Orchestration system cloud based, so that resources can be utilized in best possible way. Once this system move on to cloud, spawning, provisioning and shutdown of VM’s can be done at runtime and per need basis.

References

[1] S. Aditya, P. Mathur “Foundation of Software Testing”, First Edition, Pearson Education, 2007.
[2] Humble, Jez, and David Farley. Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010.
[3] Campbell, G., and Patroklos P. Papapetrou. SonarQube in Action. Manning Publications Co., 2014.
[4] Karhu, Katja, et al. "Empirical observations on software testing automation." Software Testing Verification and Validation, 2009. ICST'09. International Conference on. IEEE, 2009.
[5] Berner, Stefan, Roland Weber, and Rudolf K. Keller. "Observations and lessons learned from automated testing." Proceedings of the 27th international conference on Software engineering. ACM, 2005.
[6] D. Gelperin, “How to support better software testing,”Application Development Trends, no. May, 1996.
[7] Karlström, Daniel, Per Runeson, and Sara Norden. "A minimal test practice framework for emerging software organizations." Software Testing, Verification and Reliability 15.3 (2005), pp. 145-166.
[8] Eldh, Sigrid, et al. "Towards a Test Automation Improvement Model (TAIM)." Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on. IEEE, 2014.
[9] http://findbugs.sourceforge.net.
[10] http://checkstyle.sourceforge.net/index.html.
[11] http://pmd.sourceforge.net.