Achieving Right Automation Balance in Agile Projects

Vijayagopal Narayanan

Vijayagopal.n@cognizant.com

Abstract

When is testing complete and How much testing is sufficient is a fundamental questions that has been asked for a long time. A convincing answer to this question will help in optimizing the effort spent by Development and QA teams in testing the systems that they build.

Multiple approaches have been proposed as a silver bullet to achieve high quality, from TDD, BDD, ATDD, Service layer testing etc. To traditional requirements based testing. One view is to breakdown the system into its architectural layers and use tools to verify what is being build in each layers. In such a scenario some of key questions that arise are, how do I know testing at these layers brings value? How can I be sure that there are no redundancies? And where should I invest more? An attempt has been made in this paper to come up with a framework which provides a unified platform on which testing across these layers can be integrated. In addition, a set of metrics are defined to measure the effectiveness of testing in various layers and their contribution to overall quality.

Biography

Vijay Narayanan is a Senior Manager at Cognizant, currently working at Chennai, India. He has carried out multiple consulting and delivery engagements in Testing and Test Automation for customers in different industries. The engagements covered various dimensions like Automation Strategy, Testing Tool Strategy, Test Process Assessment, and Setting up of Testing Center of Excellence. He has strategized and deployed Testing Transformation Program for Large Testing engagements.

He also led development and deployment of solution accelerators like Automation frameworks, and QC integration tools. He led the Automation and SOA COEs and was responsible for new service offering, partnership management and business development in this area.

He has 13+ years of experience in software services industry, out of which 10+ years in Testing and Test Automation.

Vijay has an M.E in Computer Integrated Manufacturing and B.E Mechanical Engineering from Bharathiar University.

1. Introduction

Different viewpoints have evolved to the question of "How much testing is sufficient?". The answers are provided from one of the two perspectives, one is to verify the smaller parts which make up the whole (i.e. unit testing, code coverage etc.), and the other is to look at the behavior of the emergent system as a whole (i.e. a black-box point of view). The development in test automation has also followed on the similar lines with different tools and frameworks addressing each of these.

In the bottom-up view of building-in quality the system is broken-down into its architectural layers and testing tools are used to verify what is being built in each layer. In this scenario one of the key questions is, how do I know testing at these layers brings value? In addition a top-down method is also adopted typically during later stages of testing. In which case, How can I be sure that there are no redundancies? And where should I invest more – at each layer or at system level? In this paper an attempt has been made to come up with a framework which provides a unified platform on which testing across these layers can be integrated. In addition, a set of metrics are defined to measure the effectiveness of testing in various layers and their contribution to overall quality by leveraging and extending static code analytics tools, and analytics. It helps in painting an integrated view of various testing and how effective they are in exercising the system.

2. Testing Multitier Architecture

Most of the systems built in today enterprises follow a multi-layer (N-Tiered) architecture. The tiers typically have a GUI layer to render the user interface, a business layer which encapsulates business rules and orchestration, a services layer which implements business rules\functions and a data layer for persistence. Even with logical separation of responsibilities, the complexity involved in testing such a system can be very high.

Organizations often employ a pure black-box testing strategy to validate such systems. A requirement based testing approach is used and traditional GUI test automation tools are employed to bring efficiency to test execution. In more matured organizations a grey-box test strategy is adopted using a combination of GUI and Service testing tools and frameworks, thereby providing feedback at two of the layers (GUI & Business Process) involved.

In cases of Agile projects teams may employ techniques like unit testing, GUI automation for behavioral and acceptance testing and API\Service layer testing. Though a combination of these efforts may improve the quality of the system being developed, it is very difficult to balance these test approaches and decide on how much testing is required at different levels.

3. Limitations with Existing Solutions

In this context it is essential to put in place a feedback mechanism so that the test efforts are optimized. Some of the mechanisms that are already available are,

A) Static Testing and Code coverage tools: Static Testing tools can automate the process of code review for standards and best practices. It reveals sections of code which can cause errors and supports implementation of white-box testing. Code coverage tools provide information on the parts of code executed as part of black-box testing. When a project team makes a refactor or implements a new feature code coverage tools can be used to understand whether the refactored or added code has been covered by testing or not.

B) Behavior Driven Development (BDD): Behavior driven development is an Agile development methodology that focus on user needs captured as behaviors in ubiquitous language and form an executable acceptance criteria. Business, development and quality teams collaborate to capture behaviors that are to be met by the system under development. These are then tied together in a validation script and run using BDD frameworks like JBehave. When these scripts are run feedback on the system is provided at behavior level.

What we need here in addition to the above methods is a mechanism which can verify the effectiveness of coverage at different architectural layers. This mechanism needs to be light weight and also should be able to automatically collect data and analyze the feedback that is received from across the layers. It must also provide a view to integrate the business domain models with system model, which consists of implementation units like Classes and Web services.

4. Integrated Test Effectiveness View

4.1. Why we need an Integrated Test Effectiveness View

If we look at the feedback mechanisms available from the perspective of the multitier architecture, we see that code coverage and behavior coverage address the opposite ends of the spectrum. The code coverage tools provide very granular information and helps in understanding the inner working of the system, while the BDD helps in focusing on the bigger picture, as shown in the Figure 1 : Integrated Test Effectiveness given below. Coverage effectiveness measures for other layers are either not available or not in common use. It will be interesting to have feed-back mechanism for other layers. A framework that can integrate feed-back data to provide a seamless in an out-side-in or inside-out fashion will be valuable from the quality assurance stand-point. It can also help in identification of gaps in testing and to highlight the layers\areas to be focused based on the change is being implemented.

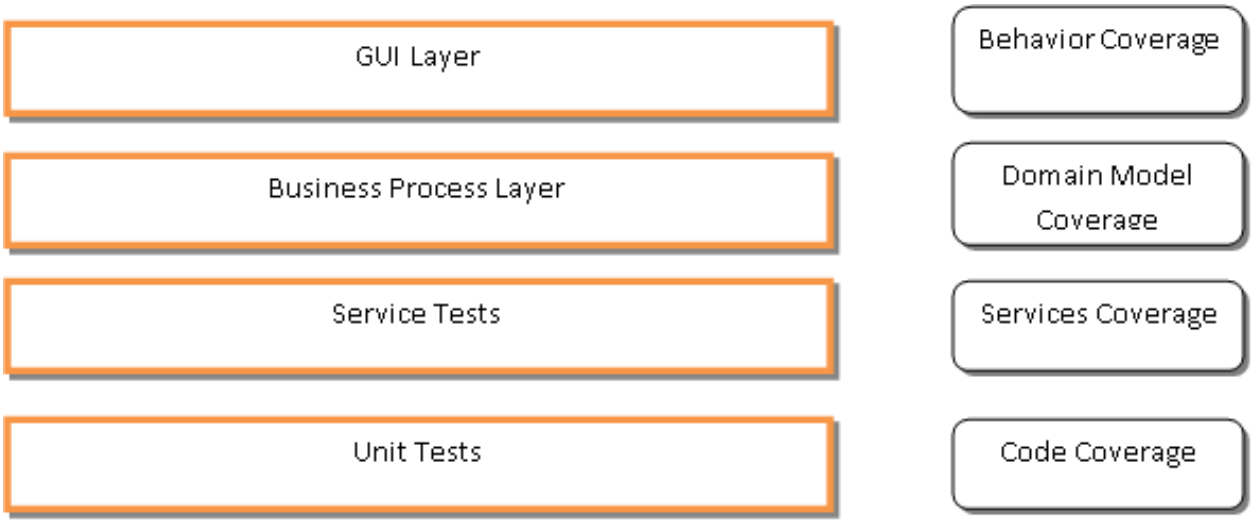

Figure 1 : Integrated Test Effectiveness

We are at an interesting point of time where the developments in testing concepts and technology field provides us various tools which can be used to arrive at a better feedback mechanism.

4.1.1 BDD approach

BDD frameworks allow testers to hook the test scripts at deferent levels of the architectural layers, while at the same time keeping focus on behavior of the system. At the GUI layer the step definitions can be implemented using one of the GUI\Browser drivers like Selenium, WatiN etc. Again at the services layer SOAP UI or other such tools can be used to invoke the services and check for the system behavior. Similarly for the API\Class\Method layer.

4.1.2 Model driven engineering

Model driven engineering approach focuses on domain models and deriving systems designs\patterns from there. This provides tools for testers to understand the underlying systems and compare coverage between the actual and modeled system behavior. It even allows automation of the test design processes using techniques like state-machines to derive scenarios for the system.

4.1.3 Analytics

This is a technology which has not been leveraged optimally to understand the testing coverage, progress and other interesting attributes. As systems become increasingly complex, it will require more of analytics to bridge the gap between the domain-model and application model. There by, helping us to understand emergent behavior is produced through interactions between the system components. e.g.: How do hundreds of unit test scripts map to a service and how do they in-turn map to a behavior of my domain model.

4.2. Conceptual Model of Proposed Integrated Test Effectiveness View

By leveraging the above mentioned advances an approach is proposed to provide the testing team a valuable Information Radiator to address the key shortcomings in the current approaches. This can be used in real-time to understand the quantum of testing carried out by various team and vector the test efforts on a real-time basis.

It is assumed that the project team has implemented static testing and code coverage tools, service and business process layer testing as well as traditional functional testing at GUI layer.

The conceptual components of these solutions are explained in Figure 2: Conceptual Model below.

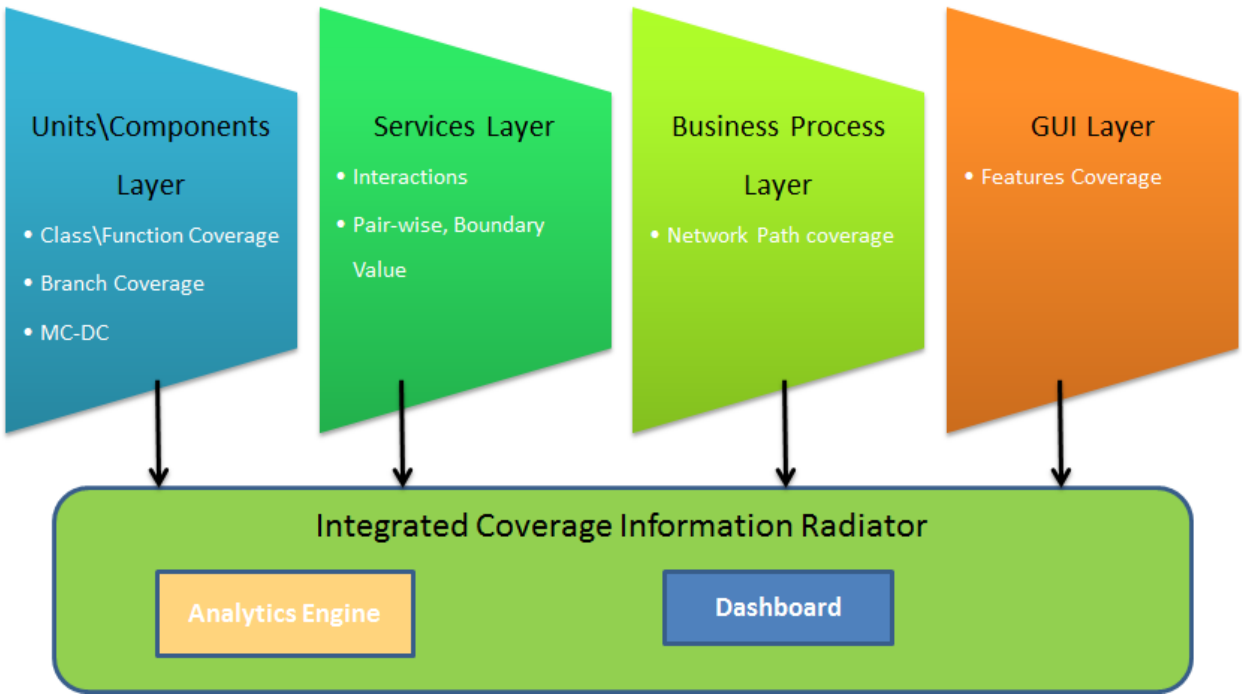

Figure 2: Conceptual Model

4.2.1 Unit and Components Layer

This component extracts the code coverage and static code quality data for the various packages that are part of the system being tested.

4.2.2 Business Process Layer

A component will analyze the business processes that are covered by testing. One of the approaches that can be used is to, leverage the middleware\BPM component. Typically the business process orchestration happens through a middleware component or Business process engine. A BPEL model is prepared and used to define the orchestration mechanism. This information can be matched with actual business process orchestration data generated during the testing.

The other approach is to leverage service virtualization tools to stub all systems other the System under test (SUT). By analyzing the invocation records from the virtualization tool, information could be gathered to trace the systems that are called as part of a business process.

Both the approaches can provide high-level as well as granular data on the coverage achieved through testing at the level of business process layer.

4.2.3 Services Layer

Services level coverage data can be generated by extending the capabilities of service testing tools and gather information on the atomic and composite services that are tested at the courser level. Detailed coverage at data element level like boundary value\pair-wise coverage can also be achieved.

4.2.4 Functional\Story Coverage

The next element is the coverage at system behavior or feature level. Projects adopting any one of the BDD frameworks can gather this information. This will show at a business level which features are being tested.

4.2.5 Analytics engine

This is the key component that will run algorithms to integrate and piece together a system view from the coverage data that is made available from all the layers. All the information that is gathered at each of the layers, as mentioned above, will be integrated and a complete picture of the solution will be constructed in the Analytics engine. This will include performing analysis, such as A) which class and services render a particular behavior, B) Which services are orchestrated during a business process etc. We envisage integration with Business modeling and BPM tools to compose this integrated picture of the system.

4.2.6 Dashboard

The insights produced by the analytics engine are provisioned through the dashboard which will serve as an information radiator for the teams. For this purpose BI tools like Informatica or light weight implementations likes Google Analytics APIs can also be used.

5. Implementation Approach

The above conceptual model can be realized by employing tools\frameworks like SONAR, SOAP UI, Selenium and JUnit for multilayer testing. Reports from these tools like test coverage, test failure rate etc. will be fed into the Integrated Test Effectiveness Synthesizer, which can be a Java based component. This component will have necessary logic to integrate the feedback from each layers and the resultant dataset will be stored in a database like MySQL. A dashboard generator component will query and format the data in JSON to feed the Dashboard. Dashboard rendering engine can be based on Google Analytics API along with an agile project management tool.

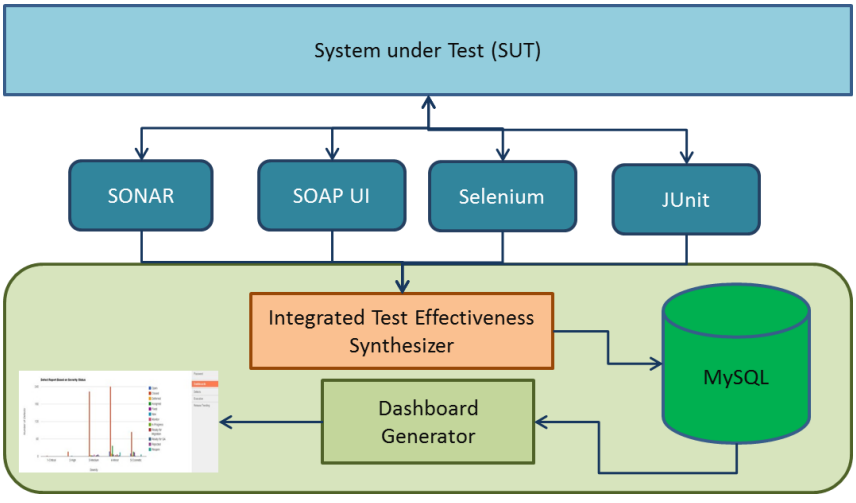

Figure 3 : Implementation Model

A template reporting format for integrated test effectiveness dashboard is shown in Figure 4: Template Report given below. In this view the failure of test scripts at GUI, Service and Unit level are shown in red. They can be aggregated at the level of features, web service methods and classes respectively for Acceptance, Service and Unit test levels.

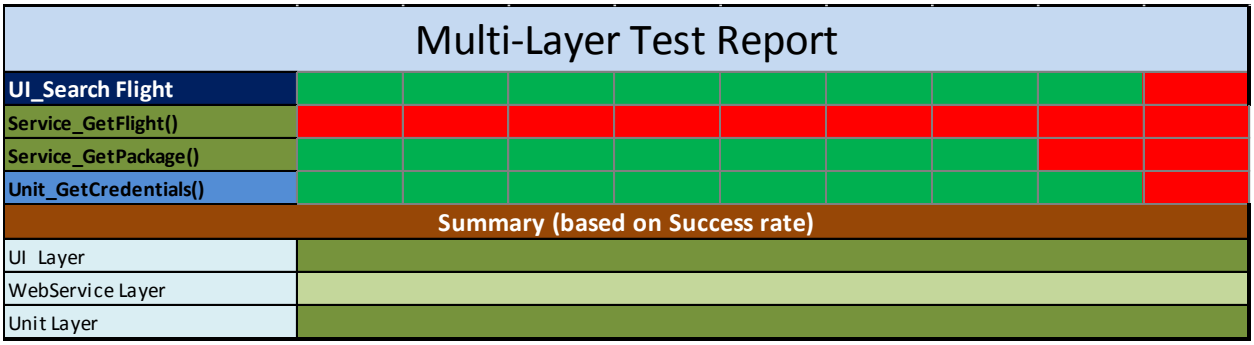

Figure 4: Template Report

6. Conclusion

The challenge of doing the right amount of testing at right time increases due to system complexity and multiplicity of test approaches that are adopted; there is a necessity for a "Crystal Ball" of sorts to view the big picture. In the current approach different testing techniques are used validate various layers of the system. Still there is no mechanism available to verify the cumulative effectiveness of these testing efforts. This paper proposes an Integrated Test Coverage Radiator that can integrate with testing tools used for multi-layer testing and provide a synthesized view into system behavior at these layers. This approach can be adopted for situations where the applications being built are complex, or connect to multiple dependent systems to keep the testing on track and thereby to achieve the right balance in automation.

References

Larsen, Michael.2012. "Get the Balance Right: Acceptance Test Driven Development, GUI Automation and Exploratory Testing", *Excerpt from PNSQC 2012 Proceedings.*

Bach, James.1998. "A Framework for Good Enough Testing." *Computer, IEEE Computer Society (Oct):124-24.*