Agile and DevOps in a Multi-Vendor Business Transformation Project
Sharon Elgarat, Solution Architect
Amdocs Testing Services

Traditional agile methodology includes several principals, including a scrum team made up of business, testers and developers, working together to determine scope and develop small units of work named stories. These principal fail to exist in a multi vendor environment.

**PROPOSED METHODOLOGY**

Agile methodology developed by Amdocs Testing Services:
- The definition of Minimal Market Features (MMF) is replaced by Minimal Testable Features (MTF)
- MTFs are sent to all vendors to develop within internal scrum teams
- Delivery dates of MTFs are aligned for all vendors
- An MTS is defined by: Business value, testability
- Added considerations to define an MTF: Autonomic, minimal scope, end-to-end, complete, product oriented
Implementing the slime mold intelligence in software testing
By Aviel B. and Evgeny P.
Software Engineering Department, Sami Shamoon College of Engineering

What is the Slime Mould?

The slime mould is basically an enormous amount of single cells with thousands of nuclei. They are formed when individual flagellated cells swarm together and fuse. The slime mould has some interesting features, it continuously grows in search for food, it creates a network of sensors around it to get information about food, it marks the path that it walks in search for food.

What does it have to do with bug resolution?

The slime mould uses a technique called “spatial” memory to find food. In its continuous work of finding food the slime lives a trail of a thick mat of nonliving, translucent, extracellular slime which will be used in the future to acknowledge which areas it visited. Many bugs appearing in software testing can be resolved to a “root cause” affecting several bugs appeared in early testing stages. If we can model early appeared software bugs as food found by the slime mould, then maybe there is a way to find the root cause as the central part of the slime.

Develop

During the development, each function will be a little crumble of food, a set of functions will be a food source.

Design

In the test scenario we put some sources of food.

Analyze

If some bug/mistake happened in the program the slime mold cuts its path to the food source or the crumble of food.
Improve Your Unit Tests by Using Chemical Bonds

By Aviel B., Daniel C., Evgeny P. and Adir N.
Software Engineering Department, Sami Shamoon College of Engineering.

In software engineering, a critical part of the development is to test your product. The most low level of testing is Unit Test, in which individual units of source code. One of the main goal of Unit Test is to find if there is a problem at an early stage, before production stage, but there is a catch: which Unit Test to write, and how many? How do we cover all the depths of code?

In order to achieve this goal, we suggest to look at Unit Tests, at the code being tested as a chemical organism which made of chemical bonds, which in turns allowing us to split and refactor Unit Tests into the logical atom.

Choosing core unit tests  
Causing it to fail and writing down the results  
Classifying the chemical bonds by the results

For more information about chemical bonds, unit tests types and this research please scan this QR code.
Socialize Automated Testing

Ofir Vaknin
Tamir Kratz
Shamoon College of Engineering (SCE)
Mentor: Mr. Dani Almog

Software Engineering Final Project

Project highlights
RDTA - an approach to the buildup process of test automation infrastructure which proposes reuse of testing artifacts as a fundamental principle for the creation of test automation.

We were looking for answers to two questions:
• Can testing automation artifacts be reused?
• If so, how?
The term software repository here refers to a storage location from which software packages or artifacts may be retrieved for reuse in other systems or software products, preferably - as is.

Exploring different aspects of the reuse of software test automation artifacts and elaborating on several practical implications and changes that arise from the implementation of this new paradigm in a software development organization.

Our goal >
• Implement the principals of Repository Driven Test Automation (RDTA).
• Research and Develop Software Reuse platform.

Our Product >
• Developing a sharable directory of software artifacts.
• Share and Use code easily: Scripts, Data Structures, Data Bases, Networking Environments and more!

Socialize you’re Automation today >
• Join the growing community of automation testing software developers!

If you are using our socialize automation tool, you can save time and money in your Software Development process >

Automation Testing

Crowd Sourcing

Software Reuse

Cost

Manual
Automated

Times Run

SAT

1
TESTING YOUR SOFTWARE TESTERS

Ori Bendet | Inbound Product Manager, Application Delivery Management

ABSTRACT

In QA organizations today, a tester must have technical know-how, good communication skills, and attention to detail. We know that a tester's main responsibility is to test the software that developers develop to ensure that the product meets the quality standards expected of today's applications. But apart from that, it's difficult to measure what exactly makes a good tester.

QA managers and their team members are constantly under pressure to test faster and more efficiently, and deliver software with fewer defects. The role and importance of QA in today's R&D teams is evolving from simply finding defects to protecting the corporate image. As a result, your testers have to be more productive and more efficient, and change their mindset to think about quality over quantity. It's not just about finding bugs; it's about continuing to measure and improve, and finding the right bugs to make the end-user experience better.

In this lecture I will share with you some of the key performance indicators (KPIs) that we use to measure our own testing efforts: Percentage of high/critical, escaped defects, Time to test, Defect resolution time, Percentage of rejected defects and what we've learned from each of them, and how our team improved its efficiency and productivity as a result.

THE METRICS THAT REALLY MATTER

ESCAPED DEFECTS are those that are missed by the tester and found by the end user. The key is making sure the tester is focused on what really matters to the customer. This KPI ensures that you're focusing on the most important defects and areas that are important to the end user.

TIME TO TEST is the time it takes a backlog item to get from 'in-testing' to 'done.' We found this to be an important KPI for measuring the efficiency of the tester and for determining how slowly or quickly they are completing tasks.

DEFECT RESOLUTION TIME This KPI measures the tester's responsibility and ownership for their bugs. The tester's responsibility is to track all of the resolution time, and not just the time it takes to find the bug or verify the fix. We need to make sure that the bug is found, fixed, and verified in a reasonable time to satisfy users.

PERCENTAGE OF REJECTED DEFECTS These are defects that are found in the product but not accepted by the developer as defects. A large number of rejected defects indicates whether or not the developer and tester are on the same page about the feature's functionality and its purpose.

THE METRICS THAT APPEAR TO MATTER

TOTAL NUMBER OF DEFECTS how many defects does your application currently have that are open. The quantity of defects is no longer as relevant as it once was. You may prefer that a single bug in your application critical path will be found and solved quickly on the spot rather than the other 40 that get logged.

DIRECT COVERAGE This is the automated and/or manual test coverage of a feature in agile, what's done is done, there is no possibility of missed coverage. If any feature is not covered by tests in a particular sprint, it will not be considered done; and will be moved to the next sprint until it's completed.

UNIT TESTING COVERAGE Measuring unit testing coverage doesn't necessarily contribute to the quality of the product. Even if you do have high unit test coverage, you're still not guaranteed high quality in the final product, as the majority of defects are detected during integration testing and end-to-end testing, which is done after the unit tests have run.
Understanding the User’s Privacy Preferences In Smartwatch Interaction - an Empirical Approach to Requirements Engineering

Roman Plotkin
Software Engineering Department, SCE

Smartwatches available since the early 80’s, but they are still not so popular, why?...
Understanding the user perceptions regarding the interaction with the smartphone is important, particularly from privacy and usability perspectives. We test empirically the users’ preferences and needs regarding their interaction with the smartwatch at different locations by using an automated tool. By applying this approach it is possible to involve the end-user in the requirements engineering and improve the process of requirements validation (before UAT), user satisfaction and application quality. For example, a user may prefer to block phone calls in a public area, but still get a notification at these places. Design a reconfigurable menu can support this.

Proposed methodology
An Android surveys application for Smartphones, including integration with Smartwatch, that collects user answers about their preferences of using a smartwatch or smartphone on basic operations (like receive phone calls, messages, alerts, etc.), in addition to the answers user location and the filling time are recorded to DB, using a PHP web server.

Initial Results
☐ Users prefer to receive a notification about phone call in public places rather than the call itself (1)
☐ Users prefer receiving a private phone call in public places (2)
☐ User would rather send a text message, then receive a phone calls in public places (3)

Insights can help software designers to accommodate their applications already at early stages of the development life cycle to the users’ privacy preferences and thus improving the software quality.

Future research and implementations
- Insights can be implemented to design a location based privacy setting system.
- It is possible to apply this approach by integrating tools to validate/test users’ needs at early stages as part of the development life cycle.
UNIT TEST
The Unit Behind The Test

UNIT TEST
VS
TEST UNIT

- Units should be defined during the design phase, not pursued during testing.
- Quality is injected in code and verified through tests.
- Units of code should be tantamount to the units under test.

BY THE DEVELOPER
“Unit testing is performed by the programmer who writes the program unit because the programmer is intimately familiar with the internal details of the unit” (Naik, Tripathy, and Naik, 2007)

“Unit testing is done without the knowledge of the encapsulating software application” (Ammann and Offutt, 2008)

ONLY PUBLICLY KNOWN APIs

DURING DEVELOPMENT
“Each component should be tested while it is being implemented… this is termed unit testing” (Schach, 2004)

“Unit test: Specify and test one point of the contract of single method of a class. This should have a very narrow and well defined scope” (StackOverflow, 2016)

ISOLATION, INDEPENDENT, ASYNCHRONOUS

“Unit tests quickly become the proverbial hammer that makes everything look like a nail.” (Hunter, 2012)

“There is no consensus on the definition of a unit.” (Naik, Tripathy, and Naik, 2007)