Final Paper

Automation in Agile Testing

Vijay Kumar - Senior Software Engineer - Testing
CenturyLink Technologies
Vijay.Kumar@CenturyLink.Com

Abstract

In any Agile Development methodology, automated testing is a core activity. Any successful test automation strategy assumes a continuous delivery model with multiple agile teams. In this article we will learn the features which a successful automation strategy must possess. We'll learn about automation packs, inverted test automation pyramid and pitfalls in Test automation to be avoided to achieve and maintain a successful test automation streak in each agile project you might get to handle.

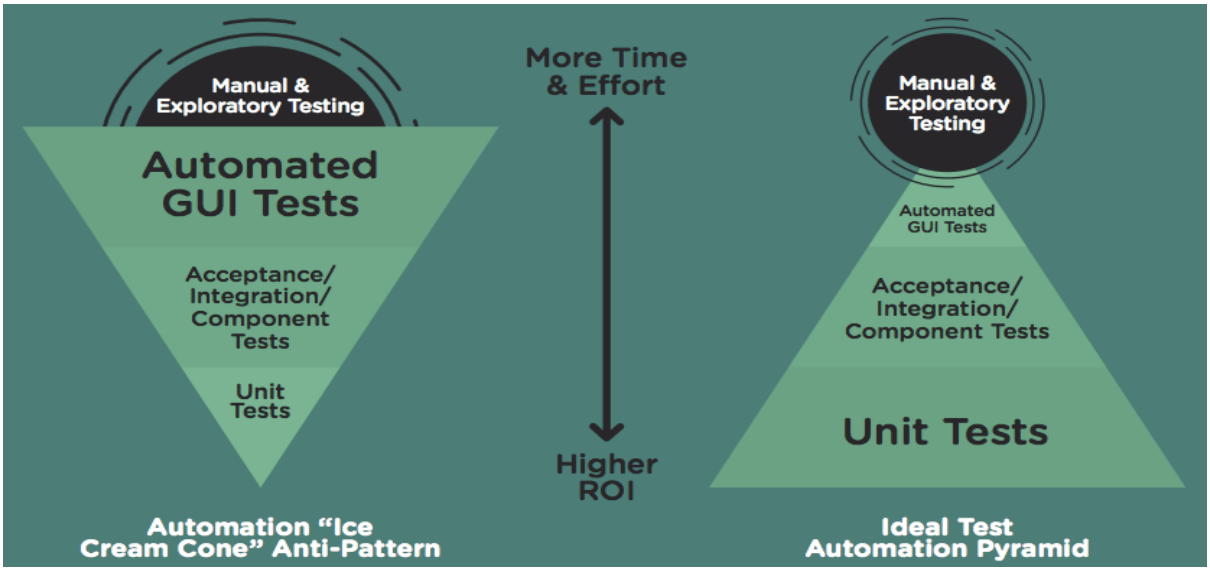
Getting Started

In order to get quick feedback, automated tests need to be executed continuously, should be fast and test results should be consistent and reliable. This can only be achieved consistently if development and testing become a coherent activity ensuring quality is baked in right from the start and making sure what is being developed works and doesn't break existing functionality.

This can be achieved by "Inverting the Test Automation Pyramid".

How can we invert the Test Automation Pyramid?

A. By scaling down the GUI tests that usually take more time to execute.
B. Increase automation at the Base layer – Unit Tests
C. Increase automation at the Mid –layer : API/Integration/Component Tests



Test Automation Strategy

Prevention rather than Detection:

Every effort should be focused on preventing the introduction of defects in the first place. We'll discuss about those methods and techniques a little later but first we will go over the methodologies that allow quick detection of bugs when they are introduced into the system and quick feedback to development.

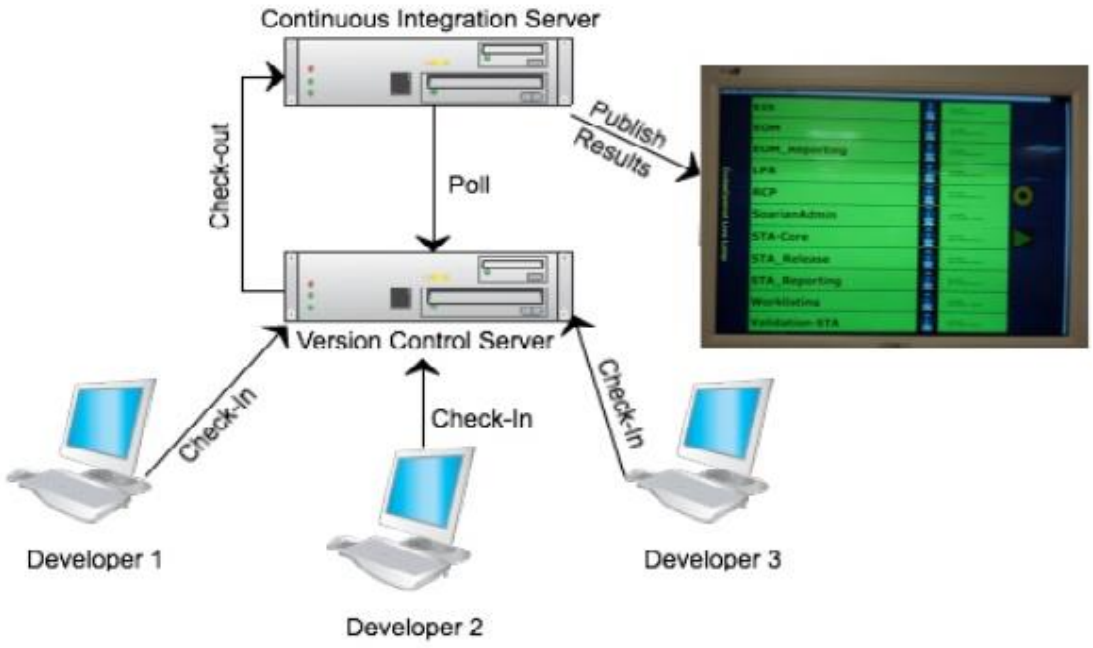
Quality should be favored over quantity. In most cases it is better to release with one feature that is rock solid rather than multiple features that are flaky. As a minimum release criterion, any newly developed feature should not have introduced any regression defects.

A quick feedback on the health of the application is of immense importance to support continuous delivery. Hence, a process/mechanism to obtain quick feedback is essential to empower us and to keep the road ahead from becoming bumpy.

1. One way of getting quick feedback is by increasing the number of unit tests, integration tests and API tests. These low level tests will provide a safety net to ensure the code is working as intended and help prevent defects escaping in other layers of testing.

Unit Tests form the foundations for test automation at higher levels.

2. The second element of improvement is running the regression tests more frequently and aligned with the process of Continuous Integration. Automation Testing should not be seen as an isolated task, but rather as a coherent activity embedded in the SDLC.



Regression Packs

Automated regression tests are the core of the Test Automation Strategy.

Smoke Regression Pack:

Sanity checks to ensure that the application can be loaded/accessed.
A few key scenarios should also be run to make sure that the application is functional.

Aim:

The aim of the smoke test pack is to capture the most obvious issues such as- application not loading or common user flow not executing.
For these reasons, the smoke pack tests should last no longer than five (5) minutes to give quick feedback in case something major is not working.

The smoke test pack should run on every deployment and can be a mixture of API and/or GUI tests.

Functional Regression Packs

They are meant to check the functionality of the application. If there are multiple teams handling different sections of the application then ideally multiple functional regression packs should exist.
These packs should be able to run in any environment as and when required and may be multiple times as the case demands.

Aim:

The purpose is to ensure that the features remain consistent in multiple environments wherever the application is tested. As functional tests are more detailed, they will definitely take longer to run.
Hence, it is important to have majority of functional tests at the API layer.
An ideal functional regression pack should last no longer than fifteen to thirty minutes (15-30 min).

End to End Regression Pack:

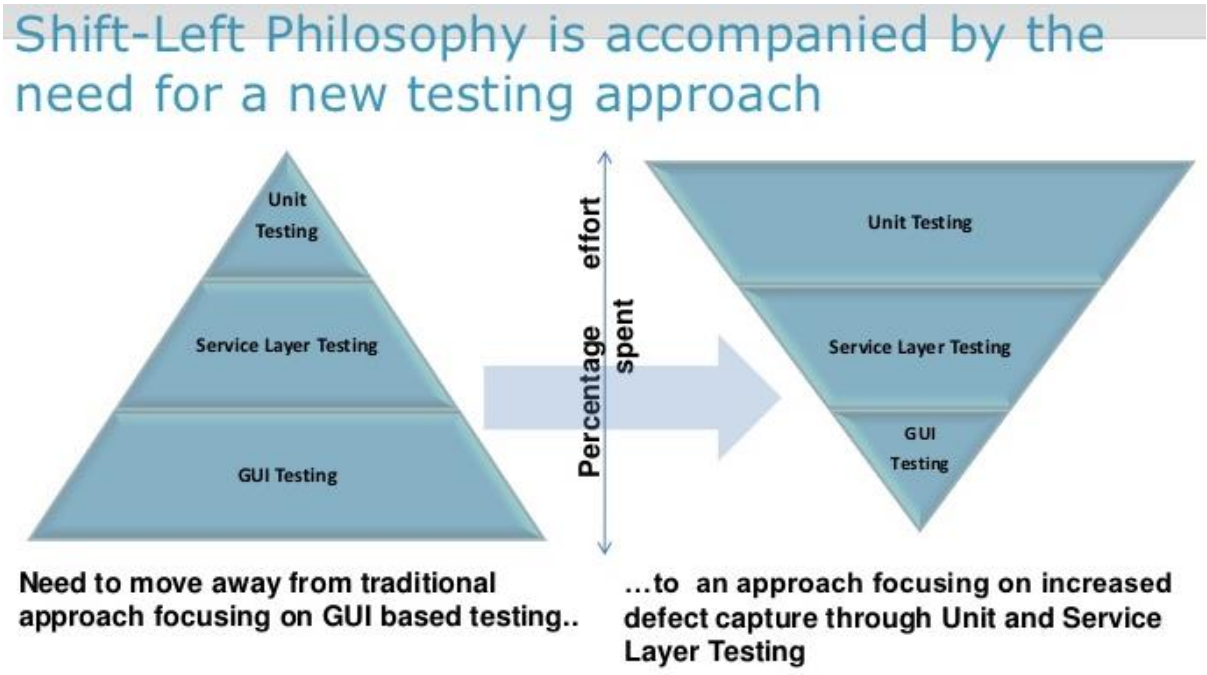
They are meant to test the application as a whole. The aim of these tests is to ensure that application interaction with various databases or third party applications work as intended.

Aim:

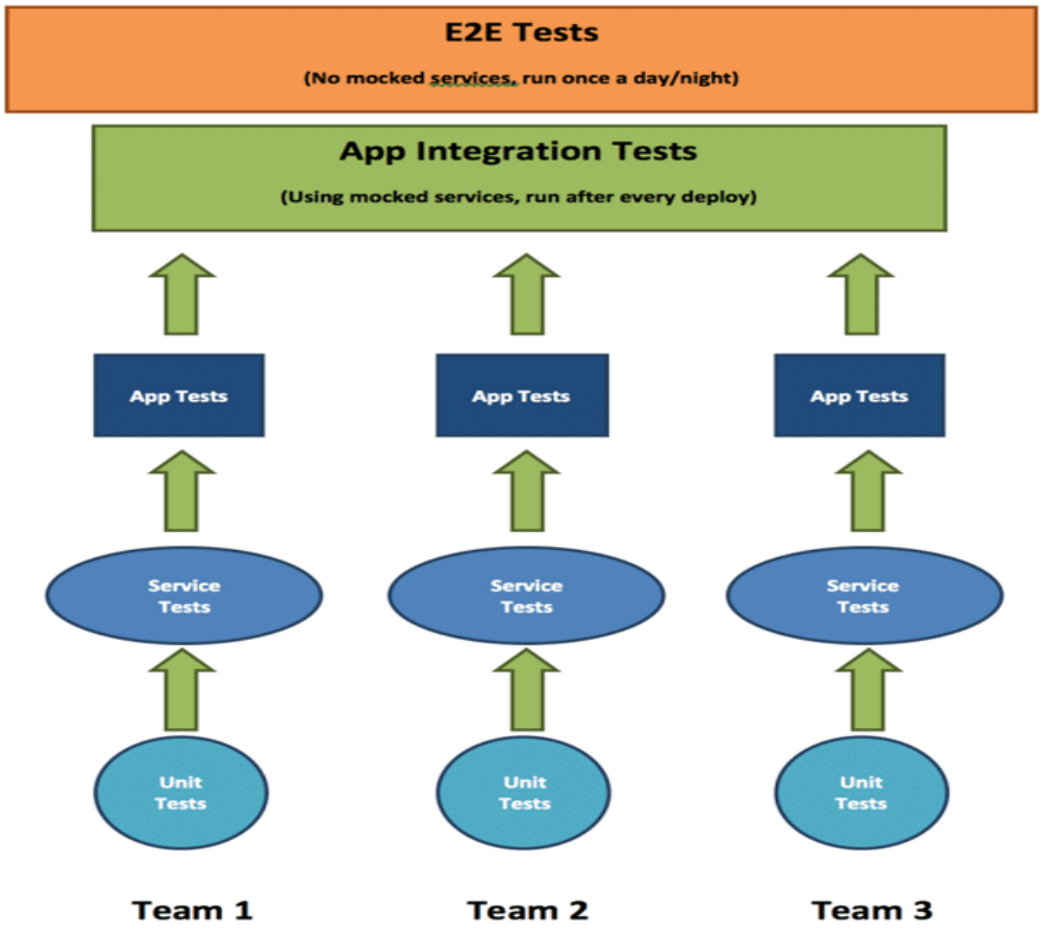
The End to End tests should not be meant to test all of the functionality as those must be handled in functional regression packs. These tests should be able to test the application as a whole involving testing the transition from one state to another plus some of the most important scenarios or user journeys.
Ideally and End to End regression pack should be run once a day or night.

Test Automation Strategy for multiple agile teams

A successful test automation strategy can be devised if we understand clearly that automation is critical and there needs to be a change in approach.



An ideal test strategy will look like the one below:



"A highly automated test suite is considered mandatory by Scrum teams; it is considered a luxury by traditional teams" – Mike Cohn

The key take-away of a successful test automation strategy is:

- It matters what you automate and when you automate it
    - The earlier the better for small tests (i.e. Unit Tests)
    - The later the better (or at least once stable) for UI-facing automation
- Automate to free up time for Exploratory Testing

Automated Unit Tests

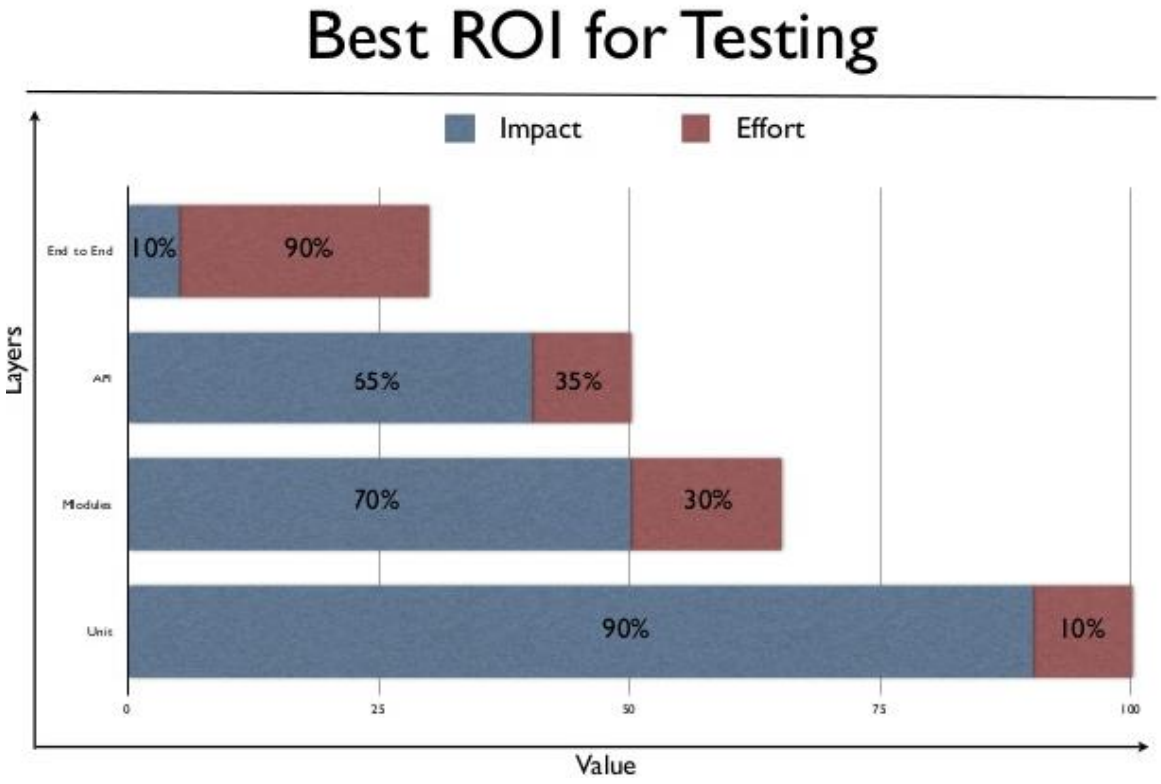
In a successful test automation strategy, the test automation will start at Unit level. Unit tests should be written for any new feature that is developed.

The Unit tests form the foundation of a larger automation practice that go all the way up to the System GUI tests.

The development team must ensure that for every new feature that is developed, a set of coherent and solid unit tests should be written to prove that the code works as intended and meets the requirement.

Unit tests provide the most ROI to the team as:

- They are quick to run.
- Easy to maintain (as there are no/minimum dependencies)
- When there are errors in the code, the feedback to the developer is quick.



Unit tests are run on developer's system as well as CI environment.

Automated Integration / API or Service Tests

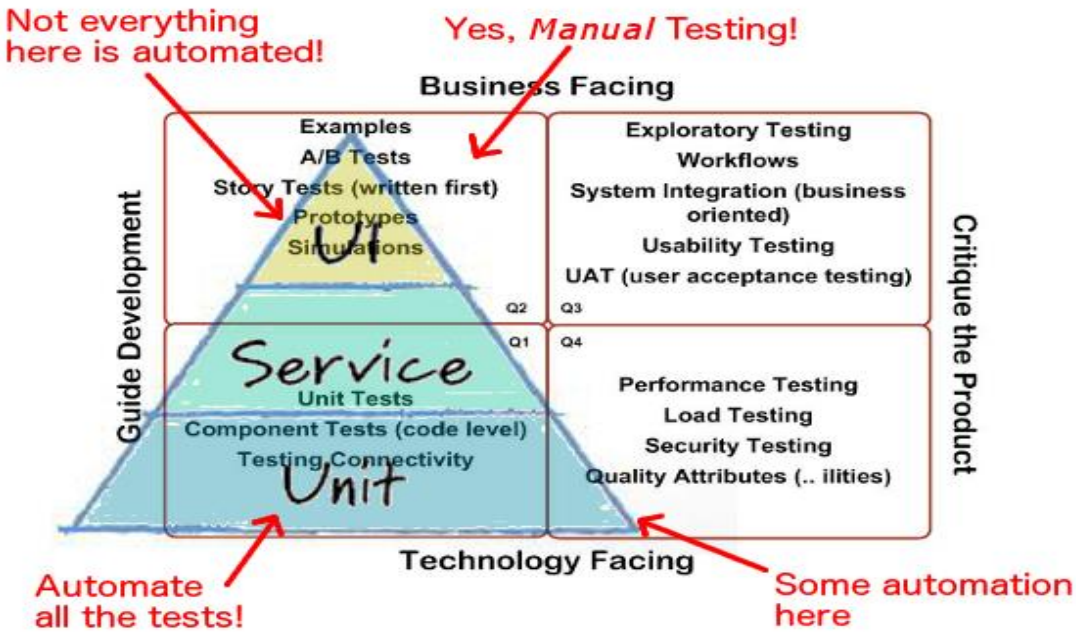
Integration Tests come one level up from Unit Tests to test collectively what makes-up the component to deliver a piece of functionality. These tests are executed only when the Unit Tests have run successfully and passed.

Service Tests are run at API layer without the intervention of the GUI web interface; hence these tests verify functionality in a pure form and because the tests talk directly to the components, they are fast to execute and are ideally part of the build.

Wherever necessary, mock services/wiremocks are used to factor out the dependency of third party systems or when downstream systems are not available to provide the relevant data for testing.

Integration Tests and/or Service Tests run on the developer's system as well as can be part of the build. However, if they take long time, then it is best to run on the CI environment.

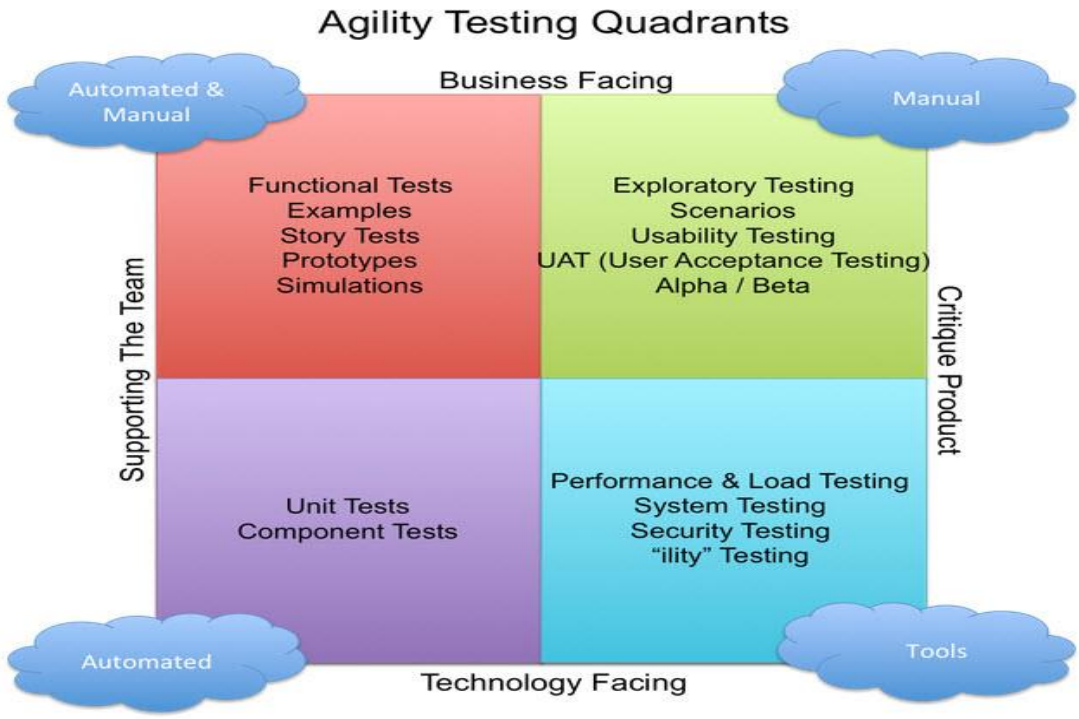
Tools such as SoapUI, Parasoft's SOATest can be used for Service Tests.



Application Testing

Application tests typically require an interface to interact with different components, hence it is anticipated that the tests are run via a browser on the GUI.
As an example, a typical big e-commerce application can be divided into different user journeys providing different functionalities or in other words different applications providing different functionalities. The concept of "App testing" comes where a group of tests that test the desired functionality of the app are organized and run.
These packs are useful in cases where a team wants to release a specific individual application and would like to know if it is functioning as desired.

The purpose of app testing is to ensure that the features of the application are functionally correct. These tests are also called "Vertical Tests", since they execute down a particular path/app. These tests are thorough and coverage is large.
Selenium Webdriver is most commonly used to run the automated tests against different browsers since it provides a rich API to allow complex verifications.



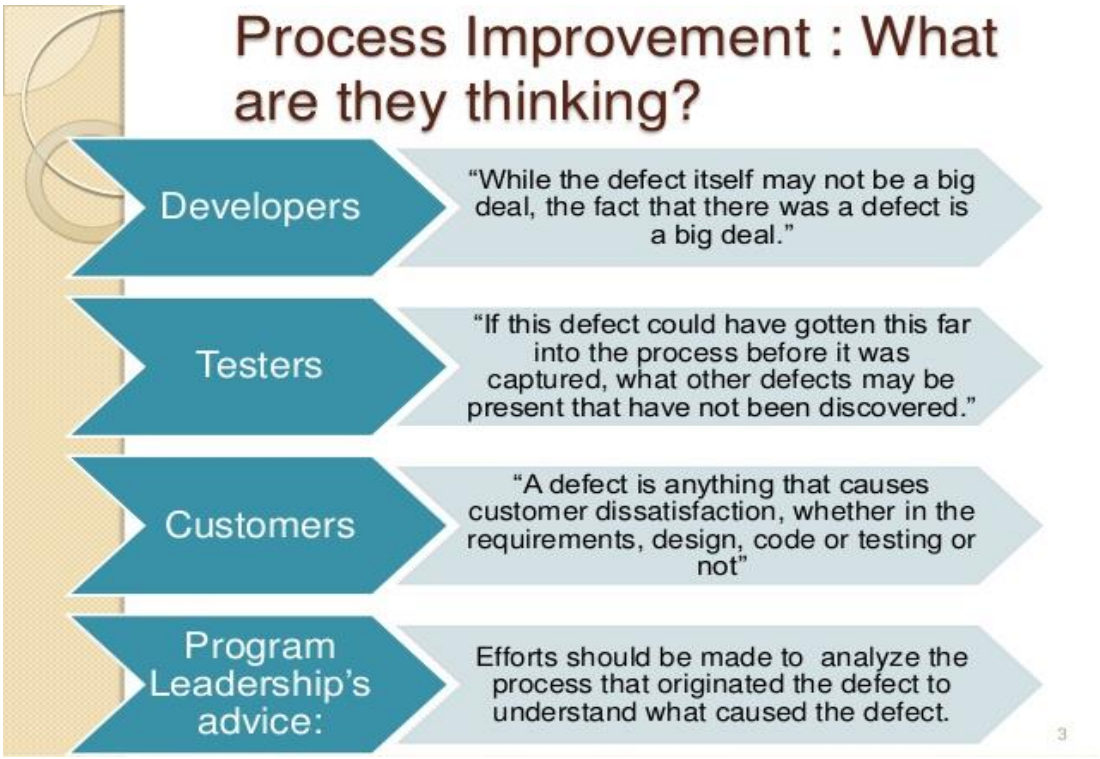
End-to-End Scenario Tests

The automated GUI tests which run against the system are typical user flows, user journeys or end-to-end scenarios. The end-to-end scenarios are usually included in nightly packs.
These tests should be kept to a minimum due to the inherent issues associated with them as explained next.

Inverting the Test Automation Pyramid

As part of a successful test automation strategy, we need to ensure to minimize the number of automated tests that are run at GUI layer.
Although running automated tests at GUI level provide good leverage in terms of simulating user's interaction with the application but it is prone to many issues as listed below:
Brittle: – As these tests rely on HTML locators to identify web elements, as soon as an id etc associated with a web element is changed, the tests fail. Therefore they bear a lot of maintainability costs.
Limited Testing: – An automated GUI could confine the tester's ability to fully verify a feature as the automated GUI may not contain all the details from the web response to allow validation and extensive coding at GUI level will make validation complex, have maintainability issues. Also it might not be feasible or profitable to automate every user journey.
Slow: – As the tests are executed through GUI, the implicit and explicit wait times for page loads or other scenarios can substantially increase the overall testing time.
Least ROI: – Due to the reasons mentioned above, the automated tests at GUI level provide the least ROI.
Thus, browser automation tests should be kept at a minimum and should be used to simulate user's behavior incorporating common user flows and end-to-end scenarios where system is tested as a whole.

Hence, the focus should be on process improvement and thinking out of the box to prevent the defects in the first place from occurring than finding them later.



Target for Automation: – Testing or Checking?

As an experienced tester will say it, profitable automation is feasible when the application is stable and the tester knows the correct expected outcome to automate for.

Thus, target for automation is definitely "Checking".

But can we replace one with the other?

It is a definite no and has been wonderfully explained by James Bach and Michael Bolton.

Testing VS. Checking

Testing is explorative, probing and learning oriented.

Checking is confirmative (verification and validation of what we already know). The outcome of a check is simply a pass or fail result; the outcome doesn't require human interpretation. Hence checking should be the first target for automation.

James Bach points out that checking does require some element of testing; to create a check requires an act of test design, and to act upon the result of a check requires test result interpretation and learning. But its important to distinguish between the two because when people refer to Testing they really mean Checking.

Why is this distinction important?

Michael Bolton explains it really well: "A development strategy that emphasizes checking at the expense of testing is one in which we'll emphasize confirmation of existing knowledge over discovery of new knowledge. That might be okay. A few checks might constitute sufficient testing for some purpose; no testing and no checking at all might even be sufficient for some purposes. But the more we emphasize checking over testing, and the less testing we do generally, the more we leave ourselves vulnerable to the Black Swan."

To put it in one line "Checking involves confirmation of existing knowledge whereas Testing involves discovery of new knowledge".

Continuing further, to be successful one must also know the pit-falls to avoid. Similarly, as a strategy is a journey in itself towards realizing a particular goal, a strategist must be aware of the pit-falls he needs to avoid for becoming successful.

Pitfalls to avoid:

A. Flawed comparison of Manual Testing and Automation

A flawed comparison:

a. Assumes that automation can replace manual testing effort.
b. Automation generally doesn't find new defects.

Testing is not merely a sequence of repeatable actions (Automation).
Testing requires thought and learning.
Other automation targets should be Regression, automation of test support activities like (data generation), big data problems.

Suggested Approach

a. Avoid comparison between manual and automated testing – both are needed.
b. Distinguish between the automation and the process that is being automated.

B. Over indulging on commercial test tools

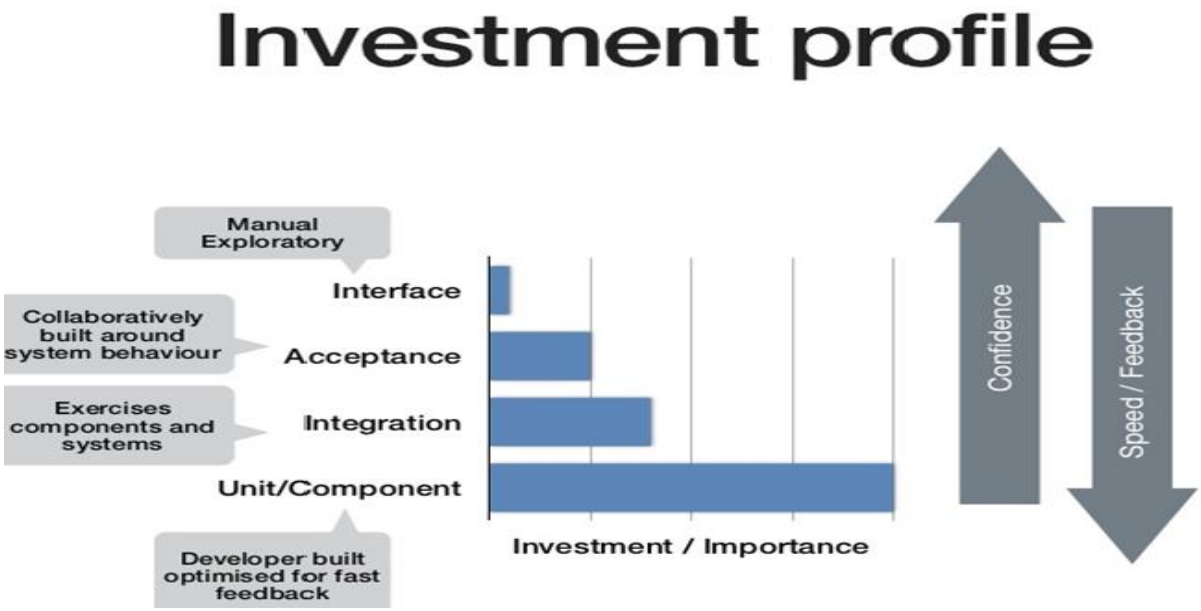
a. Vendors sell automation tools as capture replay of manual testing process.
b. Showcase tools as miracles that solve all testing problems.
c. Commercial licenses restrict the usage.
d. Sometimes incompatible technology renders commercial tools as an overhead due to integration issues.

Suggested Approach

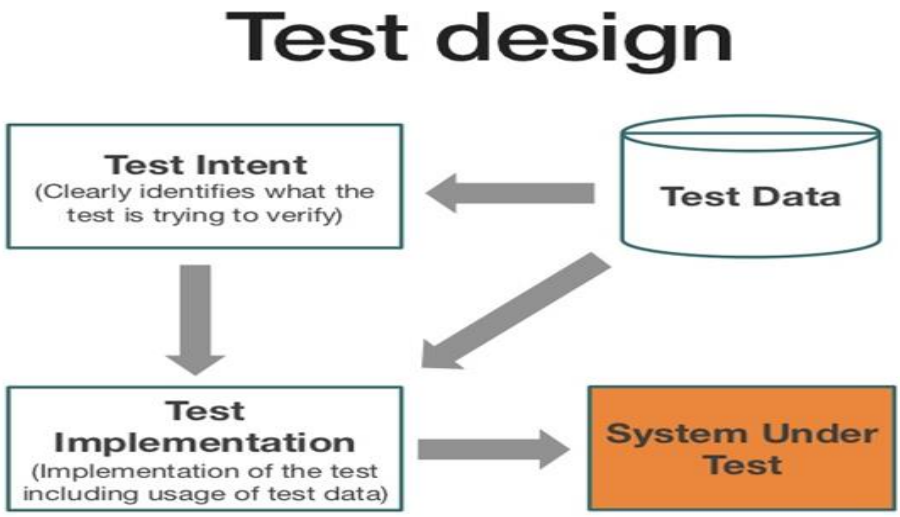
a. Using open source software tools whenever, wherever possible.
b. Choose tools that integrate well with an existing development tool chain or promise higher ROI in the long run.

C. User interface forming the basis of all testing

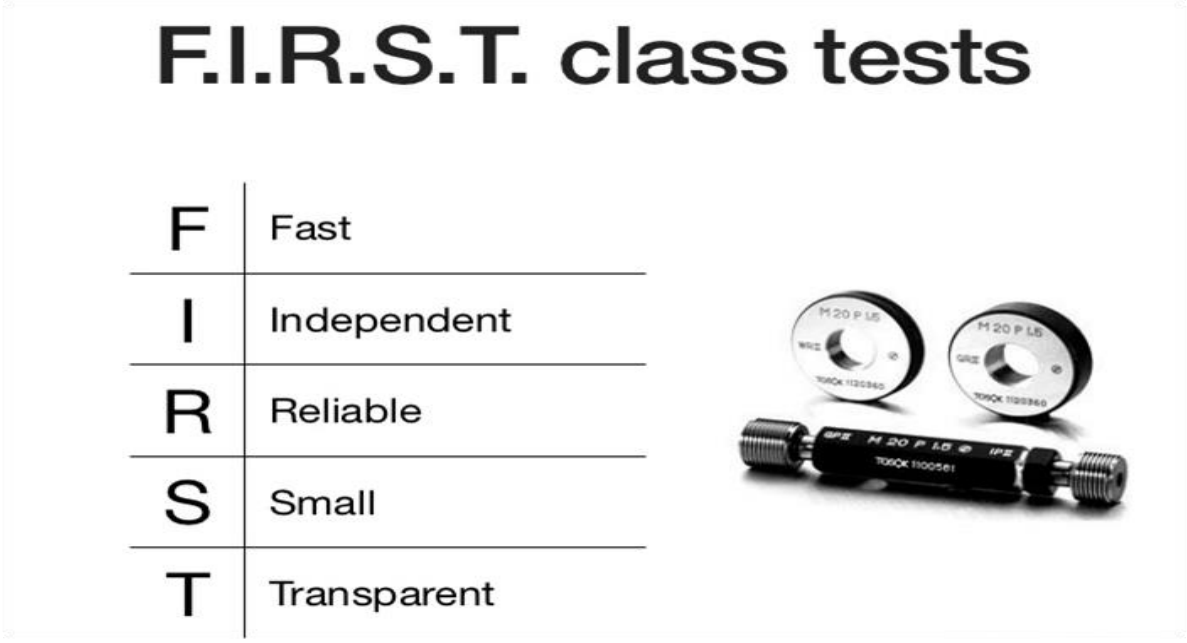
a. Testing through the GUI – Non technical testers usually approach testing through the user interface.
b. Ignoring underlying system/architecture results in tests that are slow and brittle.
c. It becomes difficult to set up test contexts resulting in sequence dependent scripts.



Understanding application and system architecture improves test design. Functionality can be tested at the right level.



Following the correct approach will result in FIRST class tests.



Suggested Approach

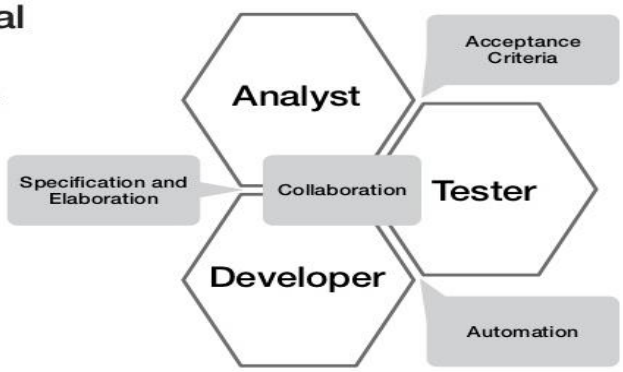
a. Limit the number of automated tests that are run through the user interface.
b. Collaborate with developers.
c. Focus on investment at lowest possible level with clear test intent.
d. Ensure that automation gives quick feedback.

D. Too proud to collaborate when creating tests

a. Automating tests in isolation
b. Automating too much and setting performance goals based around test cases automated.
c. Poor collaboration
d. Maintainability and compatibility issues.

Good collaboration

- Cross-functional teams built better software
- Collaboration improves definition and verification



Suggested Approach

a. Collaborate to create good tests and avoid duplication.
b. Limit investment in UI based automated tests.
c. Collaborate with developers to ensure good technical practices (encapsulation, abstraction, reusability.. etc)

E. Being too lazy to maintain automated tests

a. Unless automated tests are maintained, their value is slowly eroded.
b. Example – A test suite has not been run for long and its state is unknown.
c. Continuous Integration history shows consistent failures following development changes/releases.
d. Duplication within automation code.
e. Small changes trigger a cascade of failures.

Suggested Approach

a. Ensure automated tests are executed using a continuous integration environment.
b. Ensure tests are always running even if system is not being actively developed.
c. Ensure collaboration between testers and developers.
d. Ensure to make test results visible to maintain transparency about system health.

F. Frustration with slow, brittle or unreliable automated tests

Slow automation is due to a variety of reasons like:

- Large datasets,
- Unnecessary integrations,
- Reliance on GUI based tests etc.

Tests become brittle due to dependencies like:

- Having time bound data.
- Rely on sequence of execution.
- Based on production data or environments

Frustration comes up when tests become unreliable.

- Like False positives
- Wastes time in investigating.
- Failures start being ignored
- Create uncertainty about system health.
- Workarounds are created.

Suggested Approach

a. Treat automated tests with same importance as production code.
b. Review, re-factor and improve.
c. Eliminate unreliable tests.
d. Ensure collaboration with developers
e. Up-skill /pair testers.

G. Trying to cut costs through automation

a. Believing vendors who try to calculate ROI based on saving labor.
b. Unreliable analysis that undervalues the importance of testing.

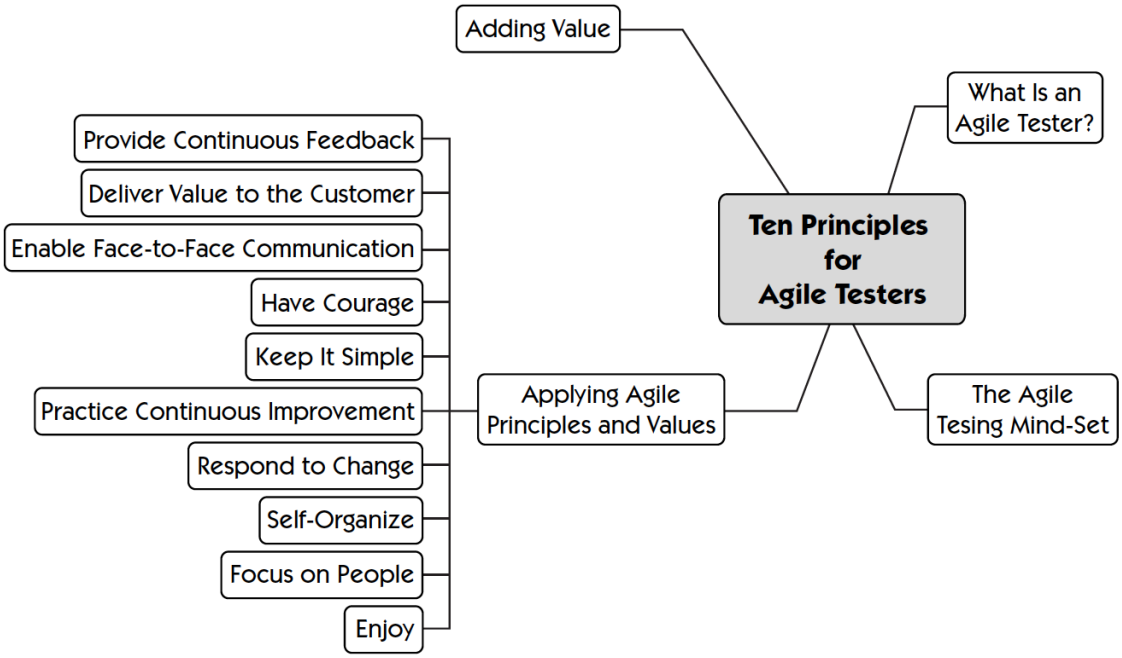
Automation is not cheap.

- Adopting test automation tools and techniques require significant investment.
- Investment in skills and collaboration.
- Investment on maintenance.

Suggested Approach

a. Ensure that the reasons for automation are clear and not based solely on reducing resources.
b. Ensure business case for automation includes costs for ongoing maintenance.

On a closing note, here are the ten principles for an Agile Tester to follow:



These guidelines for an automation strategy can be definitely tailored to an Organization's needs. Most importantly we now have an insight as how to approach automation in Agile Testing and what strategies we can apply to deliver a perfectly workable solution.

References & Appendix

http://www.drdobbs.com
https://www.atlassian.com
https://en.wikipedia.org/wiki/Agile_testing

THANK YOU!