Hewlett Packard Enterprise

Business white paper

Fast and high-quality modern software testing framework

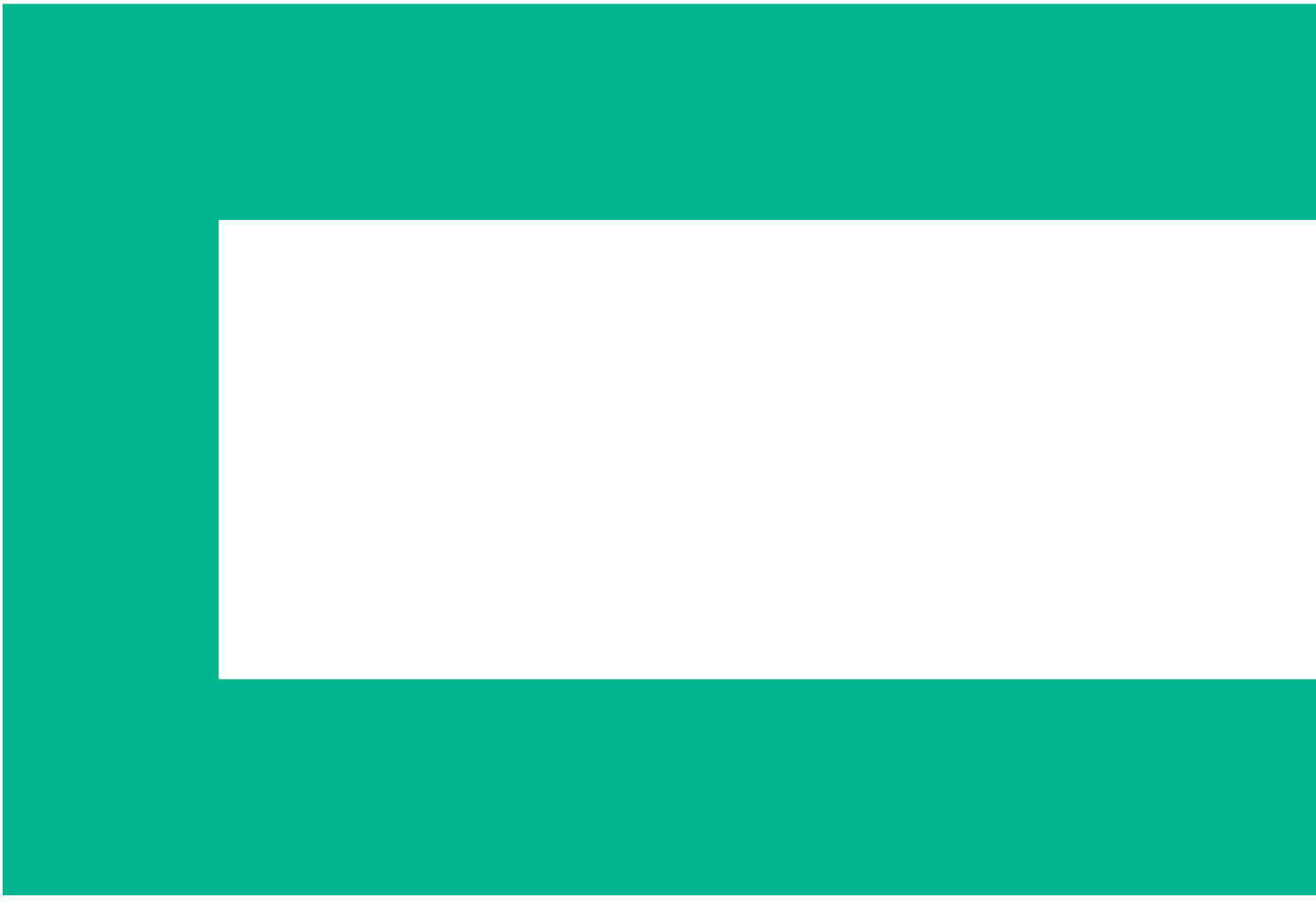

Table of contents

Introduction

Modern software development needs testing and delivery tools that can create reusable tests, and many organizations are applying test automation to agile projects. Yet 15 years after agile was first introduced to software development lifecycles, QA teams still struggle with this.

In today's era of fast-paced and dynamic application development, your end users expect to see continuous enhancements that are aligned with their requests. Being able to respond quickly becomes a major competitive advantage. Previously, software applications were composed of two to four monolithic services , but today's applications have multiple layers composed of distributed microservices, which are often integrated with legacy systems. The result is a much more complex software stack that becomes challenging to test when the goal is high quality and rapid delivery.

The "shift left" movement

Software producers today are increasingly motivated to deliver software faster, while keeping costs down and quality high. To achieve these goals, organizations are adopting a "shift left" approach, which means applying the conceptual and methodological changes required to incorporate software testing throughout the development and delivery processes.

Incremental testing and repetitive tasks

In the past, developers would design and develop the software, and then hand it over to the QA teams for comprehensive functional testing. Nowadays, the software-testing process is very different. Features tend to be developed incrementally, meaning that the basics of a feature are designed, implemented, and tested in the first iteration, with subsequent iterations gradually adding and testing more functionality until the feature is complete.

What does this mean for quality engineering? First and foremost, testers need to validate that tests that passed in previous iterations still pass after incremental changes are made, ensuring that there are no regressions. These tests are run whenever a change is checked in to the source code repository so that regressions are detected as early as possible. Many organizations will also run a nightly build, which runs more extensive tests in order to cover more functionalities. This software-testing process means that many functional testing tasks need to be repeated multiple times for each incremental change. For example, consider a feature that allows you to order a book online. The first iteration of the feature lets you click on a book in order to retrieve it from the database. The second iteration adds search functionality according to the first three letters of the book's title. This is an incremental addition on top of the initial base feature, but you still need to test that what was delivered in the first iteration is still valid in the second iteration, along with the additional search functionality.

End-to-end flow: reusing test components

To illustrate end-to-end flow, let's use the example of an e-commerce user flow:

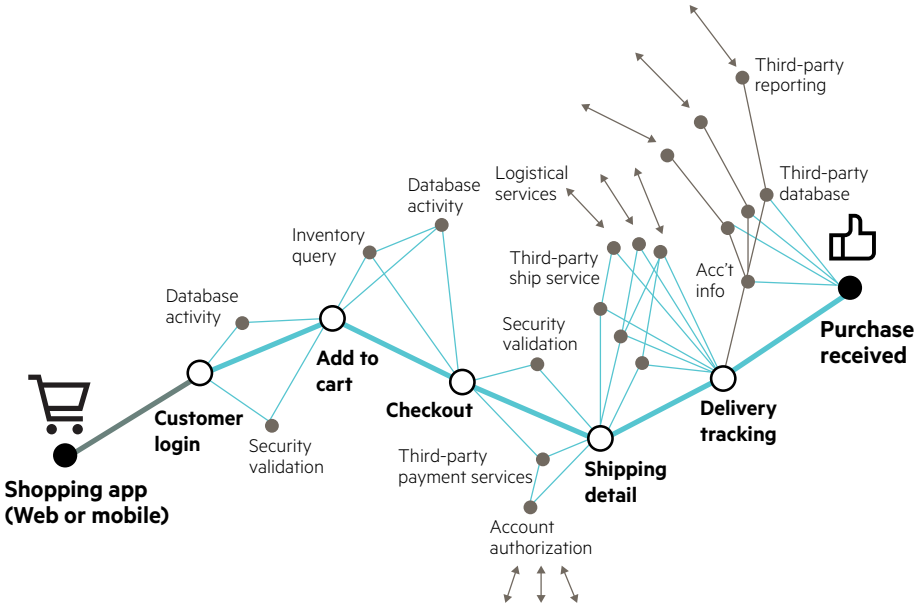

Figure 1: e-commerce user flow

From the user's perspective, there are six steps: Login, Add to cart, Checkout, Shipping detail, Delivery tracking, and Purchase received. From a system test perspective, each of the step represents a component that could potentially be reused. For example, the login functionality needs to be tested every time an end-to-end test begins.

Regardless of whether your test is manual or automated, it will have a sequence of steps that describe how to log in to the system. If there is a change to the login feature, the steps must be updated to reflect the change; otherwise, the test will fail. A typical example might be the first iteration of a login screen that asks the user to enter their username and password. In the second iteration, the user must now provide their domain as well. The test designer will need to revisit each and every one of the test cases and update its steps so that all of the tests can function as expected. However, if your login procedure is encapsulated as a component that is reused by each of the tests, you will only have to update the component, and each of the places where the "Login" component is used will automatically include the change.

In order to cope with these functional testing challenges, testers need to have the right framework and software-testing tools that facilitate effective and efficient testing to ensure that the software is delivered with high quality.

The modern framework: three core concepts

Now that we've established that a software-testing framework is the key to consistently achieving a high level of quality, the next step is to decide which testing framework to use. Here are the core concepts and capabilities that a successful framework should include:

1. **Componentization:** This is the process of defining a self-contained entity that will be considered a component and will perform a specific task, such as "Login." Each component will contain different properties such as its description, the test steps that are executed, and any parameters that are used. It must also have the ability to be included in multiple end-to-end test cases, as we described in the "Login" example previously.
2. **Parameters and iterations:** Parameters define the inputs and outputs of a component or test. Input parameters usually correspond to the user input (e.g., username and password for the "Login" component), while output parameters are the results of the execution of component. For example, a session ID might be the output parameter of a Login component. After receiving the values for the username and password through the input parameters, the Login component will perform the login, and receive the value of the session ID, which is assigned to the output parameter. The session ID can then be transferred between different components that will be executed later on in your test case. Note that input parameters can also receive the data from a previously executed component rather than a user.
3. **End-to-end test case implementation:** Once the components have been defined, along with their behavior and their input and output parameters, they are combined to make up end-to-end tests. Each test is defined by a sequence of components that transfer data to each other through their input and output parameters, and can range from having just one or two components, to highly complex tests with multiple components spanning several different systems. Going back to the e-commerce example in figure 1, we could have a test using the following components:

   Login → Add to cart → Checkout → Shipping detail → Delivery tracking → Purchase received

   The input and output parameters are used to pass data between the components. For example, the "Search item" component might have "Item ID" as its output parameter, which is passed into the input component of the "Add to cart" component.

Five capabilities of a functional testing framework

Now that we've described the core concepts, let's take a look at the key capabilities that add value to a functional testing framework.

1. **Component management:** Your testing framework should indicate which tests are using each component, and alert you to the implications of updating, deleting, or creating components.
2. **Parameters:** They represent test data, and the software-testing framework should enable you to associate several different data sets to each parameter. The data sets are essentially injected into the component's test parameters. This capability can greatly increase the test coverage of your application with very little effort by using the same test to exercise different scenarios that are defined by different data sets.
3. **Support multiple technologies:** Today's software is made up of many different technologies and environments. For example, with a book order, the system has to check that the book is in stock, and if it's not, it must automatically send an order to the supplier. This complex end-to-end test case involves multiple technologies in multiple layers, from a user interface layer to an application program interface (API) layer in conjunction with different platforms and frameworks—all of which must be supported by the system as a whole.
4. **Support various personas:** Testing teams can consist of many different personas, from technical test engineers to business experts. The test automation engineers usually create the components, and the business experts will define the end-to-end test cases and execute them. Your functional testing framework should accommodate different personas and allow them to contribute to test creation effectively and efficiently.

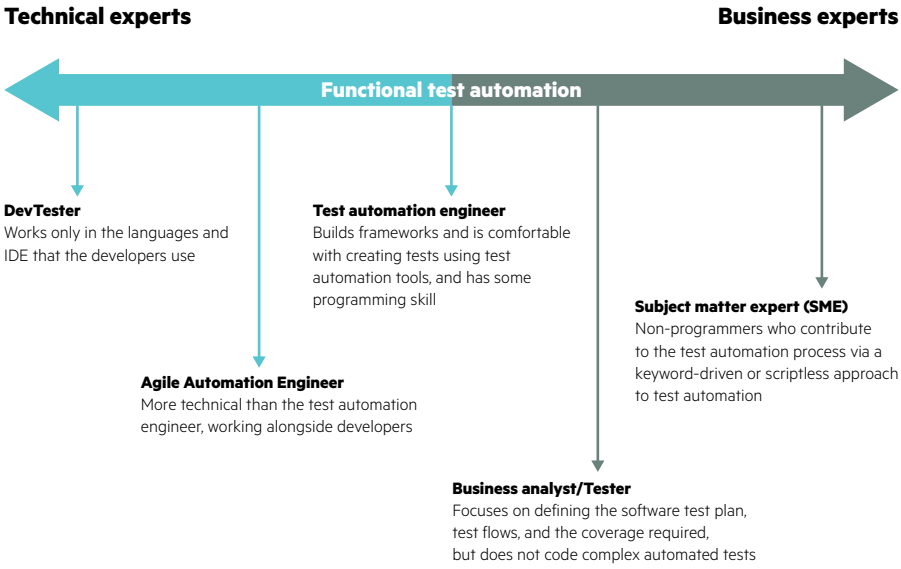

Figure 2: The different personas on a testing team

5. **Maintenance:** Modern agile development is done in iterations, which means that the software is continually changing. Consequently, the tests will need to be updated in order to reflect the changes in the software. The functional testing framework must make this maintenance painless, by identifying the areas that need to change, and allowing the tester to make the changes with as little effort as possible. In figure 1, the "Login" component will be used in almost every test case, and each of these test cases will be affected by the addition of the "Domain" field to the login process. The framework should be able to report which tests are using the "Login" component so that the new input parameter for the domain can be assigned a value in each test case. Similarly, removing a field from the original component will require the removal of the corresponding parameter and its data. Some frameworks can even automatically identify and implement any changes required to ensure that the test will continue to run.

Manual test scripts, or even automated functional testing tools without a framework, can't scale in today's ever-changing software development reality. A framework is a necessity, so make sure that you examine each of these five capabilities to ensure that you choose the correct one for your needs.

Measure the success of your testing framework

When you adopt a functional testing framework, you must ensure that your expectations are met either as a tester or as the manager of a software-testing team. Here are five additional criteria to assess when measuring the success of a framework:

1. **Return on investment (ROI)**—refers to the time and costs that your team members have invested in development vs. the time and costs saved in execution.
2. **Efficiency**—refers to how long it takes to adopt testing changes.
3. **Productivity**—refers to your component reuse rates.
4. **Optimization**—requires thinking about the exponential growth in test coverage that can be achieved and how it can be measured. This can be seen by having a single test with multiple data sets—measuring the number of different test scenarios from a single test.
5. **Maintenance**—can make up 30 to 40 percent of the time spent on daily activities. The less time you need to spend on maintenance, the more time you can spend writing and running more tests.

A software-testing framework is necessary to keep up with the speed of change and ensure that you and your team continue to deliver high-quality software to your users. Make sure you consider all of the factors that make up an effective testing framework to ensure that you choose the right one for your organization.

For more information
To read more about HPE Business Process Testing, go to:
www8.hp.com/il/en/software-solutions/test-framework/.

Learn more at
hpe.com/software/functionaltesting

Sign up for updates

★ Rate this document

Hewlett Packard Enterprise

4AA6-5193ENN, May 2016