Quality Assurance Challenges in the OutSystems Platform
- Industrial challenges -

Tiago L. Alves
OutSystems, Linda-a-Velha, Portugal
Email: tiago.alves@outsystems.com

OutSystems Platform is a software product to rapidly create and maintain industrial-grade web and mobile applications. The product consists of 5 main components that usually are jointly released in a one-year cycle. On the desktop there are 2 main components: *Service Studio* (integrated development environment) and *Integration Studio* (tool to extend the platform with custom, traditional code and integrate with existent applications/databases). On the server, there are 3 components: *Service Center* (operations management console), *Lifetime* (application lifecycle management tool), and *Platform Server* (services and libraries that are the backbone of the server components). The key differentiator of the *OutSystems Platform* is that all tasks can be accomplished visually with simple user interactions. As example the addition of a *if clause* in a business logic block, the creation of database table or web page, and the deployment of a full environment of applications must all be equally easy to perform by a non-expert user.

The visual simplicity is the top-priority concern when developing any new feature. Hence, a large part of the design and development phases deal in making the user-visible part intuitive and the main tasks easy to understand and accomplish. The *Service Studio* 1-CP (1 Click Publish) functionality, for example, is visible to the user as a single button that makes an application ready to use. Internally, this involves the following: *i)* serializing the model of a web application and sending it to *Service Center* to automatically make revisions of it; *ii)* the *Service Center* then orchestrates its code generation (either in C# or Java code) and its compilation (by calling the standard C# or Java compilers, respectively); *iii)* generation and execution of Data Description Language (DDL) scripts (to modify/create the necessary database structures); and, *iv)* deployment of the application into a web application server such as the Oracle WebLogic server. As further examples, the compilation itself makes use of data-flow analysis to ensure the output code is optimized and only the parts of the model that were affected by change are sent to the compiler. Not only the components internal of the *OutSystems Platform* are highly sophisticated, but also there is a myriad of sophisticated technology that helps developing and maintaining such components. For instance, the model that holds the details of a web application is shared among the *Service Studio* IDE and the OutSystems Compiler (a sub-component of the Platform Server). The definition of this model (meta-model) is stored in a file serving as input to *OutSystems* internal tools to automatically generate specific optimized versions of this model for *Service Studio* and for the code generator, and to automatically generate parts of the IDE itself.

The sophistication of the *OutSystems Platform* and its internal components pose many challenges to their quality assurance both at functional and non-functional levels. Thousands of tests were developed to automatically ensure each of these components work as specified with the introduction of new changes. These tests run for several supported versions of the *OutSystems Platform*, spanning different configurations of operating systems, databases, and web application servers. The current test infrastructure makes use of over one hundred machines that support building, testing (regression for different configurations, installer, and performance tests) and other quality assurance related tasks. Because of the focus on the user, most tests were created taking into account how the user interacts with the system. This natural process of thinking led to the creation of many system-level tests that have small differences between them and that have a high maintenance cost. Estimates within the company have pointed that around 30% of the total engineering time is spent just maintaining those tests. Based on the literature and good practices we are identifying ways to attack those problems.

By looking to our own test issues we laid out three provocative challenges for which we believe there is no appropriate scientific insight: *i)* capture the essence of a test; *ii)* identify the ROI of a test; *iii)* eliminate the overhead of the co-evolution of code and tests. Wrt. *i)* capturing the essence of a test, literature approaches it by introducing levels of testing. Focusing on functional testing, the most known levels of testing are unit testing, integration testing, and system testing. Although these levels are well known, identifying the testing level is not that clear from the moment that the underlying code depends of some library. The situation is even worse, when considering functional, acceptance or regression tests which in practice are concepts hard to match with the code under test. Creating tests for each level is recommended to address well-known risks. Unit tests address that the individual functionality works, integration tests address the interoperability of two or more components, and system testing address the functionality of the overall scenarios. However, creating and maintaining suites for each type of tests is a daunting task leading in many cases to not creating tests at all or creating tests at the wrong levels.

Wrt. *ii)* the ROI of test, literature approaches it using test coverage adequacy criteria. The most well known criteria are statement and branch coverage, falling under the category of structural testing. Other categories exist such as fault-based testing or error-based testing. In theory all of these criteria could be implemented, possibly, to the extent of test creation budget to be a magnitude higher than creating the code itself. The decision of which criterion to apply can be based on some notion of risk (more scientific) or good sense (not scientific), yet none of them provide a clear indication about the value of creating a new test or adding a new test to the current test suite. Wrt. *iii)* eliminate the overhead of the co-evolution of code and tests, this is addressed by putting the tests as close as possible to the code under test. In unit tests there are good practices to help achieving this such as, in Java, mimicking the package and class structure, and using in test method's name the method under test name. For integration or system tests it is more difficult to apply these good practices. Additionally, mimicking the code structure only solves part of the overall problem, since code changes can require the effort of maintaining the tests.

In this presentation, we will introduce the *OutSystems Platform*, and present a brief overview of the current quality assurance challenges. We will present a brief overview of the solutions we are putting in place to overcome those challenges. Based on those solutions we introduce three future challenges: *i)* capture the essence of a test; *ii)* identify the ROI of a test; *iii)* eliminate the overhead of the co-evolution of code and tests. For which future challenge we will present the industrial motivation behind it and foster the discussion to find new research directions.