Test Mocks for Low-Code Applications built with OutSystems

Alexandre Jacinto
FCT, Universidade NOVA de Lisboa / OutSystems
Lisbon, Portugal
agerardojacinto@gmail.com

Miguel Lourenço
OutSystems
Lisbon, Portugal
miguel.lourenco@outsystems.com

Carla Ferreira
NOVA LINCS, FCT, Universidade NOVA de Lisboa
Lisbon, Portugal
carla.ferreira@fct.unl.pt

ABSTRACT

Unit testing is a core component of continuous integration and delivery, which in turn is key to faster and more frequent delivery of solutions to customers. Testing at the unit level allows program components to be tested in complete isolation, therefore these tests can be carried out quicker thus reducing troubleshoot time. But to test at this level, dependencies between application components (*e.g.* a web service connection) need to be removed. There have been advances in mocking and stubbing techniques that remove these dependencies. However, these advances have been made for high-level programming languages, while low-code development technology has yet to take full advantage of these techniques. This paper presents a mocking solution prototype for the *OutSystems* low-code development platform. The proposed mocking mechanism removes dependencies to components that the developer wants to abstract a test from, as for instance web services or other pieces of logic of an application.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; *Software development methods.*

KEYWORDS

OutSystems platform, low-code, mocking, unit testing, software testing

ACM Reference Format:
Alexandre Jacinto, Miguel Lourenço, and Carla Ferreira. 2020. Test Mocks for Low-Code Applications built with OutSystems. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3417990.3420209

1 INTRODUCTION

Software used to be tested with manual testing and system-level automated testing techniques. Development teams spent large amounts of time on testing applications, which led to testing activities being the main bottlenecks of the development process [12]. Hence, software was produced less efficiently and at a lower rate, preventing customers from receiving software updates sooner. As a result, companies started investing in Continuous Integration and Continuous Delivery practices (CI/CD).

Unit testing [5] is a key component of CI/CD and of software testing, and when in place, changes to a program can be quickly tested, and errors or failures will be easier to troubleshoot. Unit testing also makes the execution of tests more predictable, as they are not susceptible to the interference of external systems that may be integrated into the application.

There have been advances in the industry to develop techniques that can remove unwanted dependencies when performing tests (*e.g.* connecting to a database) so that these can be made at the unit level. These techniques are mainly based around *test doubles*, which mocks are part of. A mock is a dynamic object that has the notion of state. It is used to replace another object, providing answers to calls based on that state. It should provide the same interface as the object it replaces while having a simpler implementation. However, the techniques available today apply to a more conventional line of programming, namely using high-level programming languages such as *Java* or *C#*. That being said, we are still missing this ability in low-code development.

Applications built with low-code platforms also need to be unit tested. Therefore, it is necessary to remove dependencies to application components that do not need to be included in the tests. To this end a similar approach to what is done in traditional programming can be applied in a low-code context, using mocks.

This need was confirmed in interviews made to OutSystems developers and testers, who stated that the main issue they face when testing applications is the difficulty of abstracting tests from including unwanted application components. More specifically, the components that developers more frequently feel the necessity to abstract from are web services, databases, and other pieces of logic of the applications like *server actions* (which will be introduced in the next section). This statement is supported by a survey presented by Spadini et al. in [18], which does not differentiate between traditional and low-code applications. Indeed, it identifies the components within an application that are typically mocked for testing purposes, which correspond to those components identified in the interviews we made. Therefore we can conclude that low-code platforms need the ability to mock application components, in order to enhance testing activities.

We address this need by adapting standard mocking mechanisms used in high-level programming languages to a low-code context. Our solution, which is integrated into the OutSystems platform, creates a new testing environment in which *mocks* can be used instead of the real component they are replacing. In particular, there are four main types of dependencies that our solution is able to address, namely, *web services*, *databases*, *service actions* and *server actions* (which will be introduced next).

MODELS '20 Companion, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-8135-2/20/10…$15.00
https://doi.org/10.1145/3417990.3420209

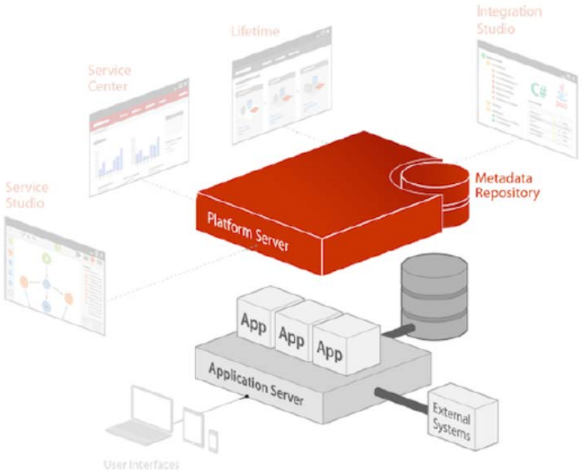


Figure 1: OutSystems platform architecture [8]

The contributions of this paper include the design of a low-code mocking mechanism for various types of dependencies. It also contributes with the implementation and validation of the solution within the OutSystems environment.

The paper is organized as follows. We start by introducing the OutSystems platform. Then, the mocking solution developed is explained. Next, we introduce a testing framework used to test OutSystems applications, that can be used together with our solution to provide mocking abilities to the tests. We then present a preliminary evaluation of the proposed solution, followed by a comparison with related work. Lastly, we briefly discuss perspectives on how to enhance the proposed mocking technique.

2 OUTSYSTEMS OVERVIEW

Low-code platforms offer an innovative software development approach using a *Rapid Application Development* methodology. Applications are built with little to none traditional code. Usually, these platforms offer a visual and *drag-and-drop* way of development [11, 15]. The OutSystems platform [10] is an example of a low-code development platform that also enables the continuous management of built applications, employing a fully *Agile* methodology [17]. The platform can be used to build web and mobile applications, allowing developers to stay away from the complexity inherent to the languages that ultimately implement the solutions.

The OutSystems architecture [8] is divided into five components, illustrated in Figure 1 (*Platform Server*, *Service Studio*, *Service Center*, *LifeTime* and *Integration Studio*). The *Platform Server* [8] refers to the set of components that implement, generate, manage, optimize, and deploy applications. Web applications can be compiled to C# or Java, while mobile applications can deployed to Android and iOS. *Service Studio* is the visual low-code development environment in which OutSystems applications are built in a drag-and-drop fashion. Developers build applications in modules called *eSpaces*. The development environment is divided into tabs. These are the *Interface tab*, the *Logic tab*, the *Data tab*, and the *Processes* tab.

In the Logic tab, *Actions* [7] can be used to create the application logic. Among all *action* types, there are two worth noting: *server actions* and *service actions*. The former are directed graphs that implement the core logic of the application. Inside a server action, there can be calls to other server actions, decision nodes, loop nodes, assignment nodes, among others, similar to methods/functions of traditional languages. Figure 2a shows an example of a server action. Also through this tab, developers can integrate their applications with *web services*. *Service actions* are basically *server actions* used as web services. These can be defined in a *Service* type application, and then imported into applications that use them. This is a recent component of the OutSystems language, and are normally used to implement logic shared between several applications.

Screens, available in the *Interface tab*, can be used to create the User Interface of the applications in a *drag-and-drop* way. In the *Data tab*, developers can define their database model. Data can be fetched using an *Aggregate*, which is a visually defined query within *Service Studio*. Standard *SQL* can also be used to define custom queries and procedures [9]. There is also a tab for *Processes*, where the developer can define the business processes of applications.

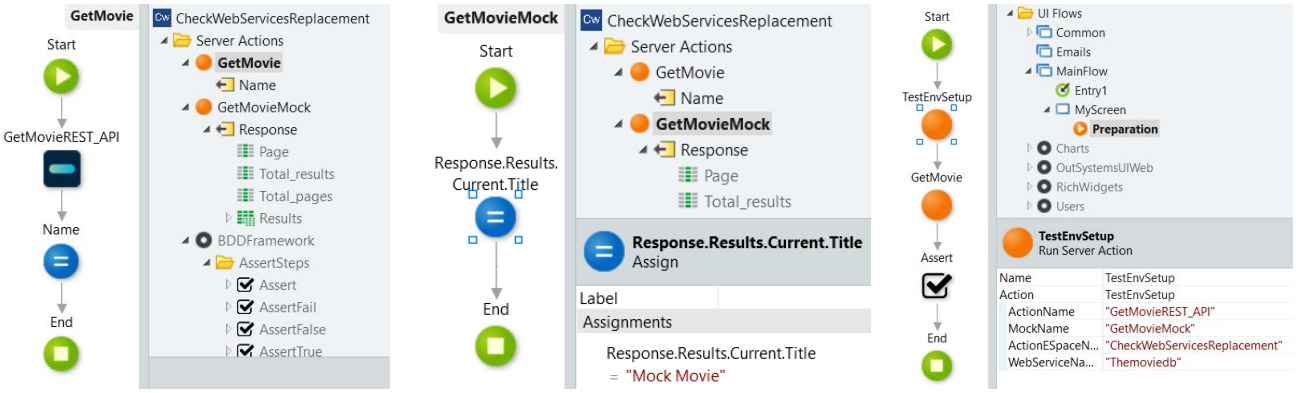
Currently, there is no native support for the definition and execution of unit tests in OutSystems applications. Therefore, end-to-end integration testing is often the lowest level of testing possible. The need to make unit tests in OutSystems applications led to the development of the mocking mechanism, which is explained next.

3 SOLUTION

To help low-code developers in their testing activities, we propose a mocking mechanism to eliminate dependencies to external entities (to the scope of the test). Our mechanism was implemented in the OutSystems platform, allowing developers to test components of their applications in isolation, which consequently promotes continuous integration and delivery in the development process. Test execution can also be more predictable, because developers can abstract their tests from external components, and can determine what mock versions of those components will return.

Our solution introduces a new test environment, in which the *mocking* ability is available. The developer can then choose this environment when testing at a more unitary level, removing some or all dependencies. Here, the components being removed from the test are replaced by their mock version. When executing integration-level tests, he or she can choose the real execution environment, and dependencies to external entities will be kept. For instance, when testing a component of the application that communicates with a web service, the developer can choose to use the mock version of that web service, and as a result, remove the dependency between the application and the service. The mock will then simulate the responses of the web service.

There are four types of dependencies that our solution can mock. These are dependencies to server actions, service actions, web services, and database queries (aggregates). The mocking of these different parts of the applications is made possible through a call replacement mechanism. For instance, the replacement of a call made to a server action, with a call made to that action's mock. Mocks are implemented using *server actions*.



(a) GetMovie action flow **(b) GetMovieMock action flow** **(c) Test action implementation**

Figure 2: Different steps of the methodology of testing with mocks

```
if(testMode && mocks.Contains("GetMovieREST_API")) {
        mocks["GetMovieREST_API"].Invoke();
}
else {
        GetMovieREST_API();
}
```

Figure 3: *GetMovieREST_API* call compiled to C# (simplified)

Call replacement is done at runtime, on the fly. Supporting this dynamic approach required significant changes to the compiler logic that generates the code for applications built in OutSystems' *Service Studio*.

We considered two other approaches. First, we tried a solution that would replace calls before compile time. With this option, the original application is cloned and then manipulated, replacing calls to components with calls to their mocks, resulting in a second application with the replacements done and ready to be tested. This option had the advantage of avoiding changes the compiler logic. However, it ended up being discarded because it did not offer a good *user experience*. The cloning and replacing procedures would have to be repeated for each test case, because different test cases could define different mocks for the same component.

We also considered an approach based on *dependency injection* [13], followed by high-level programming languages, and object-oriented programming languages in particular. We did not follow this approach because the OutSystems language is not object-oriented. Thus, in OutSystems, because dependencies are global to the whole application, they cannot be injected as in object-oriented languages (through the constructor of the class, for example). Therefore, the option that replaces calls at runtime offers the best *user experience*, while not imposing major changes to the OutSystems platform, and consequently was the option we chose.

3.1 An Example of Testing with Mocks

Next we illustrate the testing methodology we propose. There is normally an *action* to test, which in our example is the *GetMovie* action (Figure 2a). Then, we abstract the external component, in this case a webservice call (*GetMovieREST_API*), by defining a *mock* that will replace it (*GetMovieMock* in Figure 2b) which must have the same signature. Lastly, there is the definition of the test depicted in Figure 2c.

The developer will need to set up the test. For that, a special action called *TestEnvSetup*, which is part of our solution, is used. This action sets the environment to test mode. Figure 2c exemplifies how to use this setup action, providing it with necessary information like the name of the action to mock and the name of the mock.

When the test is executing, and there is a call to the *GetMovieREST_API* action, a lookup will be made to find its mock version (*GetMovieMock*), that will be called instead of *GetMovieREST_API*. The flow of execution follows continuously through the mock. Figure 3 contains a simplified version of the compiled code of *GetMovieREST_API* call.

4 BDDFRAMEWORK WITH MOCKING ABILITY

The *BDDFramework* [14] is an open-source testing framework for OutSystems applications. It provides a set of tools to create test scenarios, following the principles of *Behavior Driven Development* [6]. A test scenario includes a set of test steps, where each test step is defined in an *action* that is associated with it. Test scenarios and steps are added to the test in a drag and drop fashion. There is also a *SetupOrTeardownStep* step that can be used to perform setup operations to provide the test with necessary data, or cleanup operations that can be used, for instance, to delete data that is not necessary outside the scope of the test.

The framework promotes *Test Driven Development* since test scenarios can be defined before the functionality is even implemented. Test maintenance is also improved since the framework reports test failures when they may occur. Figure 4 depicts an example of a test scenario created using the *BDDFramework*, within *Service Studio*, testing the functionality of getting the title of a movie from a *REST API*. First, the test is set up in the *SetupTest* action. Then, the *Given* step defines the pre-conditions that need to be met before calling the logic to test and which action is bounded to each pre-condition. In the example, the title of a movie can only be returned

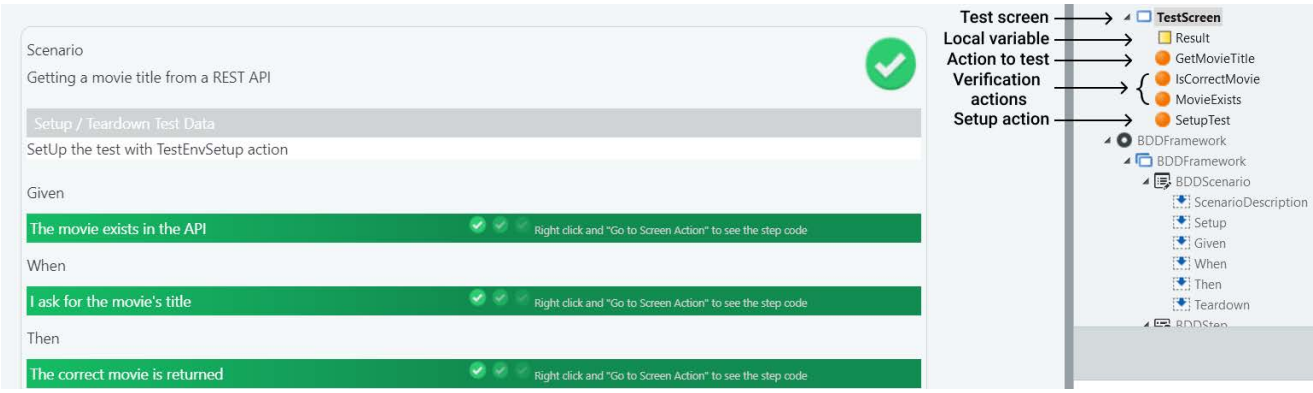


Figure 4: Test scenario created with the BDDFramework

if that movie exists in the *API*. Action *MovieExists* implements this pre-condition. In the *When* step, the logic to test is called, that is, the action that retrieves the movie's title is called (*GetMovieTitle*). Finally, in the *Then* step, the post-conditions are defined, *i.e.*, the verification logic to determine that the test had the intended outcome. In the example, when the movie title is asked for, the correct movie title should be returned. This post-condition is implemented in action *IsCorrectMovie*.

Our solution can be used together with the *BDDFramework*. In the test application where the *BDDScenario* is being defined, the developer can define the mocks for that specific test (using *server actions*). Then, the test can be set up using a *SetupOrTeardownStep*. Inside that step, the *TestEnvSetup* action can be called to associate the action to mock with the mock that will replace it. Figure 5a shows the *TestEnvSetup* being called in the *SetupOrTeardownStep*, inside the *SetupTest* action. After that, the developer can continue implementing the test like it was explained in the previous paragraph, calling the logic to test and defining the necessary logic to verify the test results in the *When* and *Then* steps, respectively. Figures 5b and 5c illustrate this.

5 PRELIMINARY RESULTS

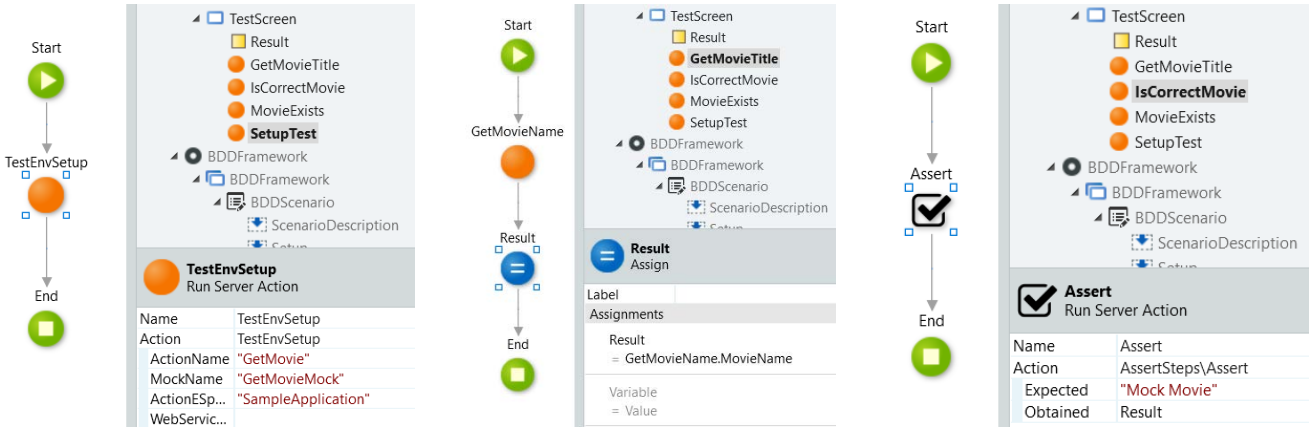
After implementing the solution, we measured the performance of running tests in real execution mode and test mode. We wanted to see the overhead imposed by having the extra step of searching for a mock when a component is called. For that, we built a simple application where a *Server action* is called. The *Server action* has the same complexity as the mock that replaces it. The relative overhead imposed by the use of mocking was approximately 9%, with an absolute value of 0,8 milliseconds, which is a negligible value in more realistic scenarios.

In real scenarios, applications can access databases via aggregate calls, and can also access web services. Sometimes these accesses can take considerable amounts of time, reaching the minute mark, depending on the specific database or web service being accessed. The time a test takes to run is largely based on these types of accesses. With the mocking mechanism available, which can help abstract the tests from these types of components, test runtimes can considerably decrease. Moreover, when testing applications that are CPU intensive, our solution can be used to considerably reduce runtime of the tests, since a mock with much simpler logic can be used to replace a laborous *action*.

Thus, we can safely say that the savings in runtime are significant and that the positive impact that a mocking solution like ours can have in low-code development is clear. Still, a more thorough validation will be conducted in the future, assessing the impact of using our mocking mechanism in the runtime performance of testing real-size applications. A validation concerning user experience is also ongoing, in which OutSystems developers use and evaluate the mocking mechanism while testing their applications. We also expect their feedback regarding how easy it becomes to troubleshoot their tests by abstracting external components, allowing them to focus on debugging the logic of their applications.

6 RELATED WORK

Guerra presented in [2] the ability to remove dependencies to external entities integrated in an OutSystems application. The testing framework prototype enables the developer to use a stub version of that web service, for instance, when testing an action that makes a call to a web service. The removal of connections between the application and external systems allows tests to be made at a level closer to unitary. A *Test Stub*, a new type of *action* introduced in [2], can be used to replace an action in a test. The definition of test stubs is made at the module level, whereas with our solution mocks are defined for each test case. This way, a developer using our solution can easily specify different responses for different test cases. The work shown in [2] only focus on removing dependencies to web services and server actions. Moreover, the prototype implemented in [2] was not incorporated in the OutSystems solution, because, at the time, it was not a priority for customers. Also, the OutSystems product has considerably evolved since then, and the solution implemented by the author could not be used together with the *BDDFramework*, since only actions like *server actions* can be used with it, and not the new *test stub* action type introduced in [2]. Our solution, on the other hand, can not only be used with the *BDDFramework*, but also with any other testing framework for the OutSystems platform. Additionally, in [2] there is no separation between the logic to test, the test, and the stubs, which does not respect the separation of concerns principle [1], while our solution allows developers to implement the mocks in different modules.



(a) Test being set up **(b) Logic to test being called** **(c) Logic to verify test results**

Figure 5: Using the mocking capability together with the BDDFramework

Other low/no-code platforms like *Mendix* [4], *Salesforce* [16], and *Unity* [19] offer mocking capabilities. However, they do it via *third-party* tools, and mocking has to be implemented using traditional programming. As far as adapting our solution to other platforms goes, we believe the generic runtime mechanism that redirects calls to mocks during the tests could be adapted to those platforms, taking the different particularities of each platform into account.

7 CONCLUSION & NEXT STEPS

The mocking solution presented in this paper, when in place, will enhance the testing ability of OutSystems applications. Developers could then deliver updates to customers faster and with higher quality. This is because the *feedback loop*, i.e., the time it takes between introducing a new functionality and the results of testing it, would be much shorter. Apart from this, it is significantly faster to debug a unit test than an integration-level test that includes an external component (e.g. a web service), since it can be hard to know exactly what the response from that component was. Furthermore, the time saved during the execution of unitary tests allows more combinations of inputs to be tested, which increases the quality of the product delivered to customers.

As future work, we will validate the solution with OutSystems developers, which will fall into two developer profiles: experienced and inexperienced. The feedback they will provide will help us in identifying the key aspects to improve in our solution. We also want to assess the improvements in performance of tests done in more realistic and complete applications of large and medium size using our mocking mechanism. We have also assessed the possibility of providing the mocks with *spying* [3] abilities that would help in test monitoring activities. For instance, a developer could benefit from knowing how many times a mock was called during the execution of a test, and with what parameters.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their comments that helped improving the paper. This work was partially supported by FCT/M-CTES grants UIDB/04516/2020 and PTDC/CCI-INF/32081/2017.

REFERENCES

[1] Mehmet Aksit, B. Tekinerdogan, and Lodewijk Bergmans. 2001. The Six concerns for Separation of Concerns. In *Proceedings of ECOOP 2001 Workshop on Advanced Separation of Concerns, Budapest, Hungary, June 18-22, 2001.*
[2] Gustavo Manuel Correia Guerra. 2010. *Testing Support for the OutSystems Agile Platform.* Master's thesis. Instituto Superior Técnico, Universidade Técnica de Lisboa.
[3] Erik Krogen. 2016. *Bond: A Spy-based Testing and Mocking Library.* Technical Report. Eletrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, USA.
[4] Mendix. 2020. *Mendix.* Retrieved August 27, 2020 from https://www.mendix.com/
[5] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3rd ed.). John Wiley & Sons, Inc., Hoboken, NJ, USA.
[6] Dan North. 2006. *Introducing BDD.* Retrieved July 16, 2020 from https://dannorth.net/introducing-bdd/
[7] OutSystems. 2019. *Actions in Web Applications.* Retrieved July 12, 2020 from https://success.outsystems.com/Documentation/11/Developing_an_Application/Implement_Application_Logic/Actions_in_Web_Applications
[8] OutSystems. 2020. OutSystems architecture. Retrieved July 10, 2020 from https://www.outsystems.com/evaluation-guide/outsystems-architecture/
[9] OutSystems. 2020. *OutSystems Evaluation guide.* Retrieved July 12, 2020 from https://www.outsystems.com/evaluation-guide
[10] OutSystems. 2020. *Platform.* Retrieved July 4, 2020 from https://www.outsystems.com/platform/
[11] OutSystems. 2020. *What is Low-Code?* Retrieved July 4, 2020 from https://www.outsystems.com/blog/what-is-low-code.html
[12] Wolfgang Platz. 2019. *Why Software Testing Remains a Bottleneck.* Retrieved July 12, 2020 from https://thenewstack.io/why-software-testing-remains-a-bottleneck/
[13] Dhanji R Prasanna. 2009. *Dependency Injection - Design Patterns Using Spring and Guice.* Vol. 1. Manning Publications, Shelter Island, NY, USA. https://research.library.kutztown.edu/ebooks/2
[14] João Proença. 2019. *How to Automate BDD Testing in OutSystems, Part 1: An Introduction to the BDDFramework.* Retrieved July 2, 2020 from https://www.outsystems.com/blog/posts/intro-bddframework-testing/
[15] John R Rymer and Rob Koplowitz. 2019. *The Forrester Wave™: Low-Code Development Platforms For AD&D Professionals, Q1 2019.* Technical Report. Forrester. https://www.forrester.com/report/The+Forrester+Wave+LowCode+Development+Platforms+For+ADD+Professionals+Q1+2019/-/E-RES144387
[16] Salesforce. 2020. *Salesforce.org.* Retrieved August 27, 2020 from https://www.salesforce.org/
[17] Sheetal Sharma, Darothi Sarkar, and Divya Gupta. 2012. Agile processes and methodologies: A conceptual study. *International Journal on Computer Science and Engineering* 4, 5 (2012), 892.
[18] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To Mock or Not to Mock? An Empirical Study on Mocking Practices. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17).* IEEE Press, 402–412. https://doi.org/10.1109/MSR.2017.61
[19] Unity. 2020. *Unity for all.* Retrieved August 27, 2020 from https://unity.com/