Lenino Manuel Lima Dias

Bachelor in Computer Science

Outsystems Logic Previewer

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Vasco Andrade e Silva, Product Manager, Outsystems
Co-adviser: Bernardo Parente Coutinho Fernandes Toninho, Assistant Professor, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2021

Abstract

Low-Code Platforms have been increasingly adopted by companies to develop their software products since visual programming languages allow a faster development process by removing various user concerns about implementation details. Outsystems enables its users to create software in an agile form in order to improve their productivity, as it is one of the key aspects to fast software delivery.

Outsystems development processes are very fast and allow the creation of state-of-the-art products in a very short time when compared to more traditional approaches. However, the feedback loop received by users at Outsystems during development is very long because the existing methods for checking application behavior such as using the application UI (or a dummy one) to test its behavior, or even using testing tools (such as *BDDFramework*) do not have an immediate feedback. Using tests to check application behavior in Outsystems is also a time-consuming process because of the required *publishes* (compilation and deployment to server) of the tests and the code. This problem in the Outsystems development significantly decreases users productivity.

Therefore, the main goal of this work is to reduce the feedback loop users experience while developing their software products in Outsystems platform by enabling them to *preview* the logic being written. In this work we intend to enhance the compilation process and the testing approaches used to check the applications behavior, by using interpretation or compilation techniques, removing unnecessary dependencies, implementing a testing approach where there is no need to fully publish neither code or tests by introducing a *logic previewer* where users will not need to publish first so they can check their application behavior.

With this work, we expect to significantly improve the users feedback loop by making it immediate (or almost immediate). By achieving this, the users productivity in Outsystems development process will increase considerably.

Keywords: Outsystems, Logic Preview, Software Testing, Feedback Time, Visual Programming Language, Outsystems Development Process

Resumo

Outsystems é uma plataforma Low-Code que permite aos seus utilizadores criar as suas aplicações seguindo o desenvolvimento *agile* de forma a melhorar a produtividade, sendo que é um aspecto muito importante para a rápida produção de software.

Os processos de desenvolvimento em Outsystems são muito rápidos e permitem a criação de aplicações modernas num curto período de tempo. No entanto, o ciclo de feedback recebido pelos utilizadores durante o processo de desenvolvimento em Outsystems (na criação ou alteração de código) é muito longo porque os métodos existentes para testar o comportamento das aplicações tais como testar a aplicação com a sua interface (ou uma fictícia) ou mesmo usar frameworks de testes como o *BDDFramework* não têm um feedback imediato. O uso de testes para verificar a lógica das aplicações em Outsystems é um processo que gasta muito tempo, uma vez que requer que ambos os testes e o código sejam publicados (compilados e carregados ao servidor). Isto faz com que a produtividade dos utilizadores em Outsystems diminua consideravelmente.

Desta forma, o objectivo principal deste trabalho é reduzir o ciclo de feedback que os utilizadores recebem enquanto estão desenvolvendo as suas aplicações na plataforma Outsystems, permitindo-os *pré-visualizar* o resultado da lógica que escrevem. Neste trabalho pretende-se melhorar o processo de compilação dos módulos e também as abordagens de testagem que são usadas para verificar a lógica das aplicações. Para isso pretende-se usar técnicas de compilação ou interpretação , removendo dependências desnecessárias durante o processo, implementar uma abordagem de testagem onde os utilizadores não precisem de publicar nem o código nem os testes a realizar. Isto introduz um novo conceito: *Pré-visualizador de Lógica*, devido ao facto dos utilizadores não precisarem de publicar antes de testar a lógica.

Com este trabalho, pretende-se melhorar significativamente o ciclo de feedback que os utilizadores recebem, tornando imediato (ou quase imediato) o resultado das suas alterações e consequentemente a produtividade.

Palavras-chave: Outsystems, Visualização de Lógica, Testagem de Software, Tempo de Feedback, Linguagens de Programação Visual

Contents

List of Figures

Acronyms

AI Artificial Intelligence

IDE Integrated Development Environment

JIT Just-in-time

REPL Read-Eval-Print-Loop

SBE Specification By Example

TDD Test-Driven Development

UI User Interface

VPL Visual Programming Language

Chapter 1

Introduction

Outsystems is a software company that is Leader in the Magic Quadrant[1] for Low-Code Application Platforms (LCAP) 2019 [31]. The Outsystems Platform is a powerful, feature-packed low-code development platform for large enterprises or developers looking to publish straight application stores [14]. By using visual, model-driven development and AI-powered tools, the Outsystems platform improves the entire application lifecycle; which combine with a cloud-native platform so that users can quickly build, deploy and manage their software [43]. The Visual Programming Languages (VPL) allow users to create software products by focusing on the intended functionality without stressing with implementation details. One of Outsystems' goals is to get people to produce state-of-the-art applications in a very short period of time. The fast development process in Outsystems environment is due to the efficiency provided by the platform owing to the usage of a VPL, the excellent architecture, among other important aspects.

1.1 Motivation and Goals

Outsystems users, when developing their application, put together the components (*widgets*, *screens*, *actions*, *etc.*) and make the necessary modifications that are needed to obtain the expected functionalities of the application. During this development process, which is generally faster than more traditional approaches, users are continuously writing the application logic in order to obtain the desired application behavior. However, there is no way to visualize the effects/behavior of the defined logic without a complete application. During the development process of writing and changing the application logic, users have no feedback about what they are implementing. Nevertheless, users need to know the effects of their code and its behavior, and to accomplish this, they usually

[1] https://www.gartner.com/en/research/magic-quadrant

follow some approaches such as: testing the application using its UI or a dummy UI (for testing purposes); testing the application using testing frameworks; or checking application misbehavior with the debugger of the platform. These approaches are not suitable for interactively visualize application logic since the feedback loop they provide to users (developer at Outsystems) is very long, causing losses of productivity (these approaches have their own limitations that will be covered in more detail in the chapter 2). Currently, in Outsystems, in order to shorten feedback of applications behavior, users employ testing frameworks and tools to check their software. However, this process requires that both the code being developed and the tests are published into the Outsystems server. Publishing modules[2] is a time-consuming process, especially for large modules. Beyond that, compiling the modules with all their dependencies significantly increases the time to finish the process, making the feedback loop very long.

This work will address the problem mentioned above of the logic visualization by allowing users at Outsystems to *preview* the behavior of their application logic during the creation and modification of code, consequently reducing the feedback loop that users experience. With this work, we expect to provide the users at Outsystems a mechanism so they can be able to create/change a piece of logic and immediately have the feedback of its behavior, without waiting for a long period of time. The proposed solution intends to improve the entire process, that starts with compiling the whole module (with its dependencies) and ends with the user testing the application with a testing framework, using approaches that will be broken down into a few stages.

These stages of the proposed work will solve the key problems in the existent process (testing applications to check their behavior) by addressing the need to publish both code and tests when testing the changed code, which is extremely heavy in terms of resources and time. The proposed work will introduce new mechanisms to allow the user at Outsystems to visualize their logic while developing it, without the need to publish first, introducing the concept of a *Logic Previewer*. To accomplish the desired goals, this works intends to use techniques such as compilation or interpretation, mocks and also combine aspects of others tools such as debuggers. Depending on time constraints, we can enhance the visualization with notions of *traceability* to allow users to have a thorough visualization of their logic when executed/tested by including executions details in the presented results. By enabling logic preview in Outsystems development processes we expect to reduces the feedback loop users experience and consequently improving their productivity. When developing software in Service Studio[3], users have access to an immediate preview of the UI being developed (a sketching of the UI) and, with this work we expect to ultimately offer a similar experience, but with applications logic.

[2]Where the user interface and business logic code are developed in Outsystems
[3]Service Studio is the OutSystems low-code and visual development environment (the IDE)

1.2 Document Structure

This dissertation is organized as follows:

- **Chapter** 1 - **Introduction**: this chapter gives an overview of Outsystems enterprise and its platform. It also introduces the problem to address and the goals to achieve with this work.
- **Chapter** 2 - **Background**: describes the Outsystems Platform, the theoretical concepts related with this dissertation such as Testing, and finally the existent approaches related to the work to be developed here.
- **Chapter** 3 - **Proposed Work**: this chapter presents the problem to be addressed and the proposed solution in more details, with the techniques to be used and it ends with the work plan to be followed during this research.

Chapter 2

Background

The Outsystems Platform consists of several tools and cloud solutions, for developers, administrators, and operators allowing rapid application development with advanced capabilities for enterprise mobile and web applications [33].

2.1 Outsystems Platform

The Outsystems Platform allows its users to create both web and mobile applications supported by a visual language, focusing mainly on assembling visual components to obtain the desired products by reducing significantly the users concerns about implementation details. This makes the development experience in Outsystems very fast and scalable. All these approaches combined make the Outsystems Platform much faster than more traditional development, with more quality and requiring less coding efforts from its developers.

The platform empowers its users (developers) with high-productivity, connected, AI-assisted tools for faster of full range of applications [43]. The following sections will explain in more detail the main concepts of the Outsystems Platform architecture, runtime and development experience.

2.1.1 Architecture

The Outsystems Platform architecture can be visualized in the Figure 2.1, that demonstrate the main components that together compose the platform. It can be divided into these main components: *Service Studio*, *Integration Studio* and *Code Generator and Optimizer (Platform Server)* [35].

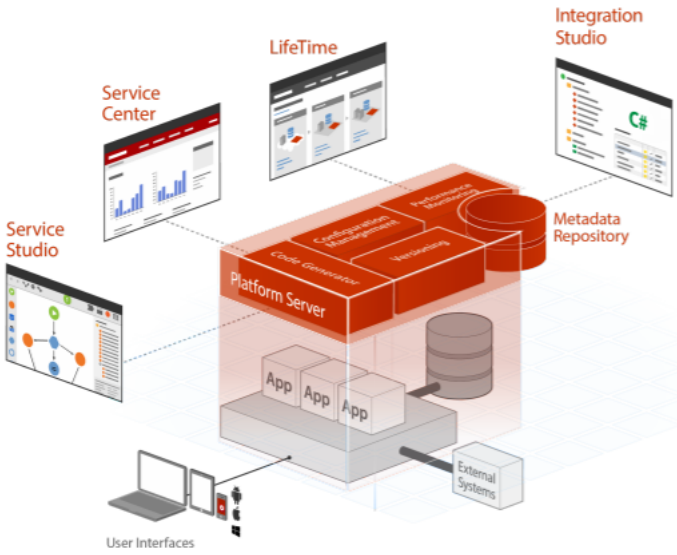

Figure 2.1: Outsystems Platform: the Architecture Overview [35]

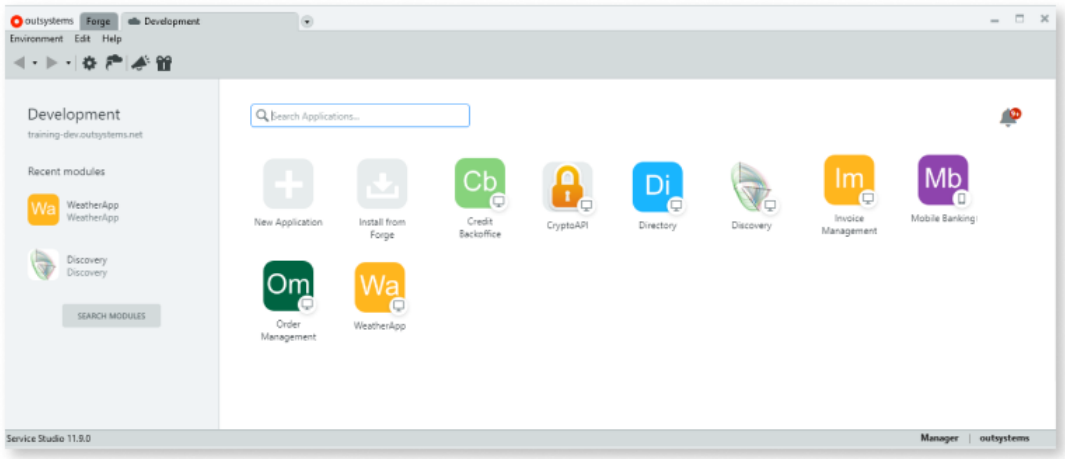

Figure 2.2: Service Studio Workspace [38]

2.1.1.1 Service studio

Service Studio is the environment available to developers to create all parts of the application stack: *the data model*, *application logic*, *UI*, *business process flows*, *integrations*, and *security policies* [32]. This IDE allows the user to drag and drop visual elements to create UIs, business processes, business logic and data models for their application [32]. After opening Service Studio and connecting to the Outsystems environment in the cloud (or an on-premise server in user's data) the user is able to see the list of applications held on the server or additionally create a new one [38]. The Figure 2.2 shows the Service Studio interface.

During the development of an application the IDE will present the users with four

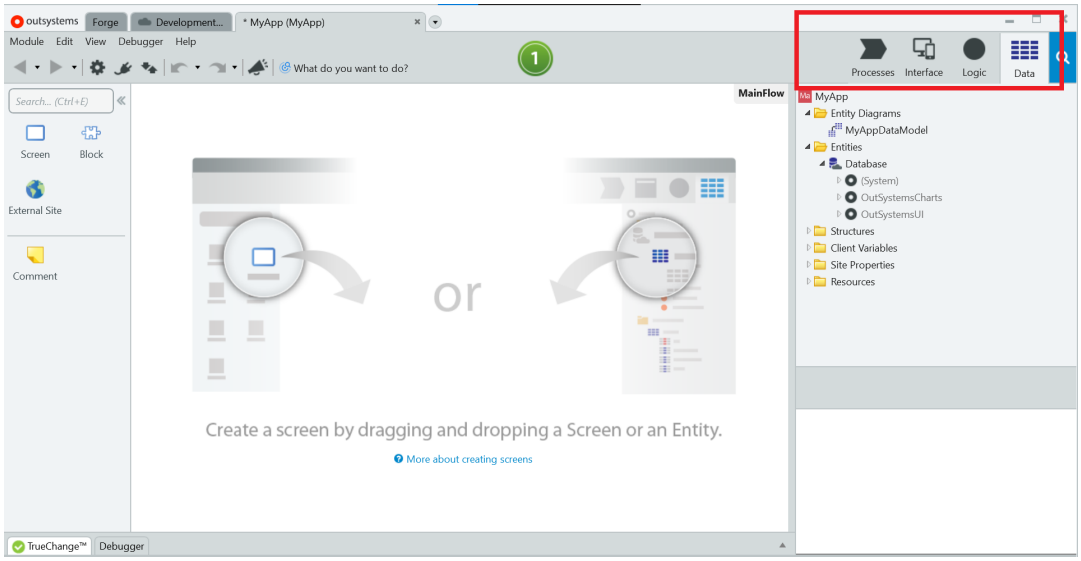

Figure 2.3: Service Studio: Example of the 4 Different Layers

main development areas: the *Interface* tab, the *Data* tab, the *Logic* tab and the *Processes* tab, as shown in the Figure 2.3.

Interface tab: This layer is used to define the UI flows of the applications. It is composed by a group of **widgets** (building blocks of a screen) that can be drag and dropped to the canvas to compose the desired screen interface. Also in this layer it can be applied **styles** to the components as desired. It is also available a **widget tree** to help the developers with the complexity of the widgets numbers [39].

Logic tab: Similar to the UI, this layer has a canvas in the middle of the workspace with visual representation of the application logic. There is also a **toolbox**, identical to the UI, but instead of showing widgets, it shows the elements that can be used inside the logic flows [39]. The Logic layer is divided into **Client Actions** (that runs on the device) and **Server Actions** (that runs on the server) [20]. It also has logic elements that allows the user to integrate with external systems such as *SOAP Web Services*, *RESTful Services* and *SAP*.

Data tab: This is the layer where the user is able to define all the **entities** in the database or in the local storage [39]. There is the possibility for users to create **entity diagrams** to see the visual representation of the data (the data model). In the Data layer it is also held **Structures** (in-memory representation of the data), **Client Variables** (user-specific data in the client side), **Site Properties** (cross application data on the server) and **Resources** (other types of data) [20].

2.1.1.2 Integration Studio

Integration Studio is the Outsystems environment where the users can create components that extend the Outsystems platform and integrate with third-party systems [32]. One of the key-points of the Outsystems platform is that once the components are deployed, they

can be reused by all applications built with Outsystems. Integration Studio allows the creation of **Extensions**, which are sets of actions, structures and entities that increment Outsystems and allows the integration with external systems [26].

2.1.1.3 Platform Server

The Platform Server is the server component and the core of Outsystems platform. It encompasses all the steps required to generate, build, package and deploy applications. These steps are more detailed below [34]:

- *Code Generator*: The code generator service is responsible for receiving the application model and generates all the native application components, ready to be deployed to an application server. This process includes checking for external dependencies, applying optimizations, generating native code for all layers, among other key tasks.

- *Deployment Services*: These services deploy the generated application components to an application server and ensure the application is consistently installed on each front-end server of an organization's server farm (factory that uses Outsystems developing technologies). The deployment service deploys a .NET[1] application on a specific front-end server.

- *Application* Services: These are services for managing the applications at runtime. It can be divided into *Scheduler Service* (The Scheduler Service manages the execution of scheduled tasks) and *Log Service* Applications are automatically instrumented to create error, audit, and performance logs).

2.1.2 Outsystems Language - A visual language

The aim of this section is to explain in more detail how the Outsystems language works and what are the key elements of the language. As mentioned before, Outsystems is a VPL dedicated to answer the challenges of digital transformation, mobile and faster delivery cycles [46].

Outsystems uses a notion of *modular programming* when developing applications, called Outsystems *Modules* [30]. Modules encapsulate everything to execute one particular aspect of functionality. In Outsystems a module is where a UI and business logic code are developed. To have a complete overview of the language it is important to briefly describe the most important elements/components of the language such as *Screens*, *Actions*, *Entities*, *Nodes* and some others that are relevant for the scope of this dissertation.

[1] https://dotnet.microsoft.com/apps/aspnet

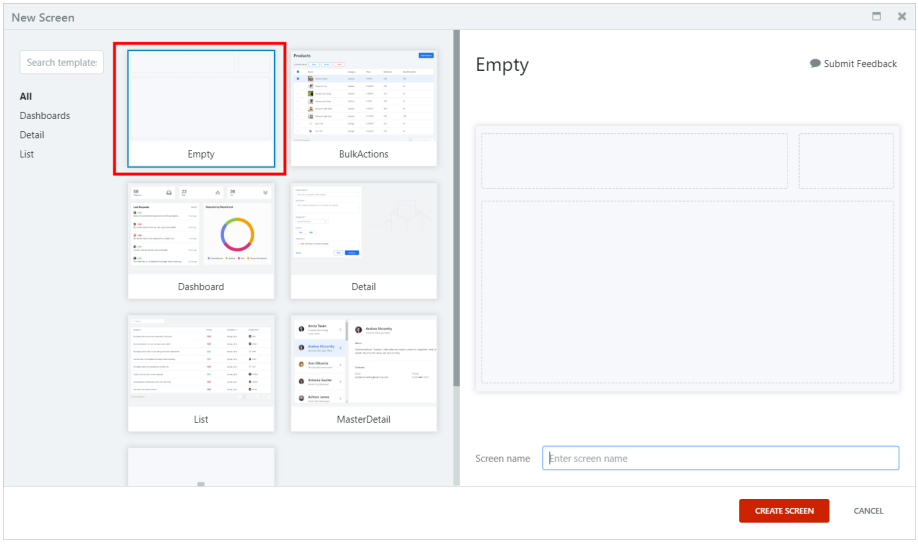

Figure 2.4: Service Studio: Create a New Screen [36]

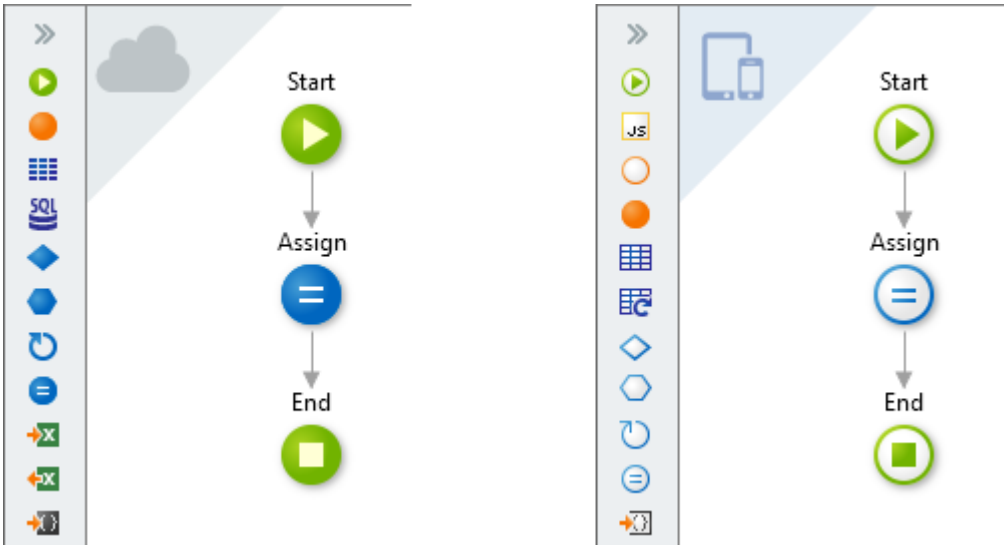

Figure 2.5: Service Studio: Representation of Actions [17]

2.1.2.1 Screens

A screen is an interface element composed by other elements that the user can interact with. Screens can be used to create web pages and mobile screens in the user's applications [36]. During development, when the users want to create a new Screen they can create an *empty screen* or use a *Screen Template* [37]. By creating an *empty screen* the user has to choose the *layout*, *widgets*, *components*, *styles* and *logic*. Unlike the *empty screen*, *screen templates* have predefined values that can increase productivity. An example of how screens can be created in Outsystems is showed in Figure 2.4

2.1.2.2 Actions

In Outsystems, Actions are logic that run on the **server** and logic that run on the **client** device (like tablet or smartphone). The user can create the following Actions: **Data Actions**, **Client Actions**, **Server Actions** [17]:

Data Actions: are actions that run on the server. The user can create Data Actions to fetch complex data from a database, when the retrieval cannot be achieved using a single server aggregate [19] or to fetch data from an external system such as REST API[2].

Client Actions: The user can set a client action as a function and use it directly in Expressions (Action flow) of the client-side logic [17]. Client Actions run logic on the user device. The user is able to create Client Actions in two different scopes: The **Screen** and the **client**. The difference is that in the scope of a screen the user is allowed to run logic when the user interacts with the Screen, and in the client logic the user is allowed to encapsulate logic to reuse in several Screens. In the right side of the Figure 2.5 it can be seen how the client logic is represented in the Service Studio.

Server Action: The users also have the ability to create Server Actions in the application to encapsulate the logic and being able to reuse it in other Actions, such as other Server Actions, Data Actions or Client Actions. These kinds of Actions run logic in the server and are represented in Service Studio on the left side of the Figure 2.5.

2.1.2.3 Data Model - Entities

Entities are elements that allow the user to persist information in the database and to implement the database model [25]. Entities are similar to tables or views in a relational database[3]. Entities are composed by:

- **Primary key**: In Outsystems, a primary key is called *Entity Identifier*. It is created automatically as an attribute called Id and added to the Entity.
- **Sequential Attributes**: are normally useful for Entity Identifier attributes. It is an easy way to ensure that each record has a unique primary key.
- **Indexes**: Similar to relational databases, Outsystems provides indexes for faster access to data in the entity.

Apart from that, in Outsystems, the user can employ **Aggregates** to fetch data using optimized queries. Aggregates can load data from the server of the local database, and they support combining several Entities and advanced filtering [19]. Aggregates can be divided into *Client-side aggregates* - run in the client logic; and *Server-side aggregates* - the ones that are in the logic flows.

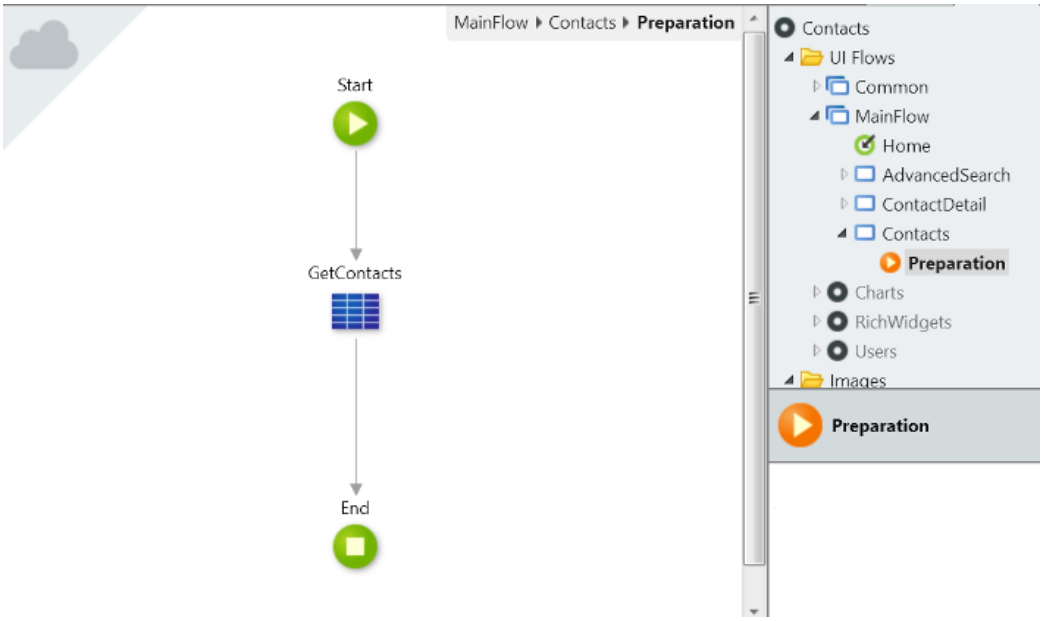

Figure 2.6: Service Studio: Nodes in an Action Flow [18]

2.1.2.4 Nodes

Figure 2.6 shows an Action Flow built in Service Studio, where different types of nodes such as *Start*, *Aggregate* and *End* can be seen. The nodes are combined in a graph representation of functions, procedures, methods, etc. We now detail some of the Outsystems Platform nodes that are relevant for this work:

- **Aggregates**: as already mentioned, aggregates are used to fetch data using an optimized query. They can load data from the server of the local database, and they support several Entities and advanced filtering [19].
- **Assign**: used to assign values to variables in Outsystems [21].
- **If**: the user can use the If Tool to execute a branch of the action flow if the condition is evaluated as *True*, and another branch if the condition is evaluated to *False* [29].
- **For Each**: repeats the execution of an action path for each entry in the Record List [27].
- **Switch**: the Switch Tool splits the action flow into two or more paths, where the first action path whose connector evaluated as true will run [41].
- **End**: used when designing the process flow of the user process, the user must end the flow paths [24].
- **Start**: this node indicates where a flow starts executing [40].

[2] https://restfulapi.net/
[3] https://aws.amazon.com/relational-database/

These are just some of the components (Nodes) that exist in the Outsystems Platform's environment. There are several others that are used by the the developers (users) to achieve their goals when developing applications with Outsystems Platform, such as *Refresh Data*, *Attach File*, *Download*, *Destination*, etc. A complete description can be found in the Outsystems Platform Documentation [45].

2.2 Software Testing: Overview

Testing plays an important role on the software life cycle and it is required from the early stages of software development such as requirements specification. Testing refers to many different approaches that intend to validate a piece of software. Testing is a challenging activity involving demanding tasks such as deriving adequate suite of tests, according to a feasible and cost/effective selection technique; the ability to run tests (the environment); deciding if the tests outcome are acceptable or not; judging whether testing is enough or not; among others. These tasks provide significant challenges to developers and testers, where skills and expertise remain of high importance [3].

2.2.1 Software Testing Techniques

Testing refers to a full range of techniques, that are split into two categories: *Static Techniques* and *Dynamic Techniques* [3]:

- **Static Techniques**: the main difference between these techniques and the dynamic ones depend whether the code is executed or not. Static techniques are based solely on the examination of the software code, documentation or information about requirements specifications and design. This characteristic allows the static techniques to be employed at any stage during the development, despite being highly desired at the early stages. Traditional static techniques include:
  - *Software inspection*: the analysis of the software documents produced, considering a compiled checklist of common and historical defects.
  - *Software reviews*: in this process, all the aspects related to the software product are presented to managers, users, customers or other different stakeholder for approval and comment.
  - *Code reading*: the analysis of the code (on a screen) to discover typing errors not related to style or syntax.
  - *Algorithm analysis and tracing*: the analysis of employed algorithms to study their worst-case, average-case and other probabilistic analyses evaluations.

  All these techniques are error-prone, time-consuming and done manually. To minimize these issues, there are some static analysis techniques proposed by researchers

relying on the use of formal methods. The goal is also to automate as much as possible the verification of the properties of the requirements and design.

- **Dynamic Techniques**: these techniques obtain information of the software by observing some executions. *Testing* in a literal sense is a dynamic technique, based on the execution of the code on specific input values. The amount of inputs to be tested are infinite, meaning that the testers/developers must wisely choose interesting ones that can coverage the software at most and run at a reasonable time. In other words, the programs are tested to observe some samples of the program's behavior. Therefore, tests strategies must be adopted to find a trade-off between number of inputs chosen and effort dedicated to testing purposes.

2.2.2 Test Classification

Many people classify tests into different categories based on their purposes and scope. Martin Fowler [12] classifies test into the following categories:

- **Unit Tests**: Some confusion often occurs with the real meaning of unit testing. There are several definitions of unit testing. Despite the variations, there are some common elements such as the notion that unit test are low-level (focusing on small parts of the systems). Unit tests are nowadays written by programmers themselves using their regular tools (sometimes the difference being the framework) and they are expected to be much faster than other kinds of tests. Besides the common elements, there are also differences among the concepts, for example, what people consider to be a *unit* (Figure 2.7). Object-oriented designs usually treat a class as a unit, while functional or procedural approaches consider a single function as a unit. It is actually a matter of situation, depending on what makes sense for a given system and its testing. The unit tests can be *Sociable Tests*, meaning they interact with other units to fulfill their behavior, or *Solitary Tests*, meaning that the tested unit is isolated. The key quality of the unit tests is their **speed**, allowing the programmers to join them into *compile suite*: a suite of unit tests that programmers run whenever they think of compiling, and *commit suite*: a suite of unit tests that programmers run before committing new versions of software. To have a good coverage of their code, programmers must build complete test suites, but they should keep in mind that the test suites should run fast enough that they are not discouraged from running them frequently enough [11].

- **Integration Tests**: determine if independently developed software units work correctly when they are connected to each other. Unit testing should be the first one to be performed to test a module on its own. Once that is complete, we can now do integration testing to test the connections among the various modules into the entire system or even sub-systems. The point of this kind of testing is essentially to

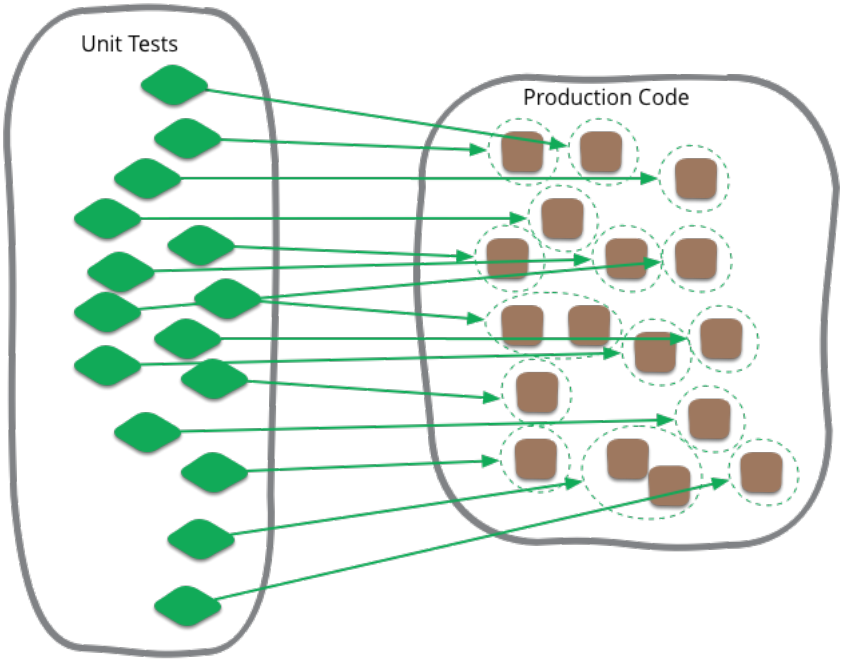

Figure 2.7: Unit Testing Sketching [11]

test if independent modules (developed separately) work together correctly. In integration testing, it is usual to use *Test Doubles*[4] to test interaction behavior without activate a third full component instance. If the third component is another service, then this technique brings many advantages as the external service requires its own build tools, environments, and network connections (Figure 2.8). *Contract Tests*[5] can be used to check if doubles are truly faithful [7].

- **Broad Stack Tests**: are the tests that exercise the most of the parts of a large application. They are also know as **end-to-end tests** or **full-stack tests** and they contrast with *Component Tests*, which only exercise part of the system. Broad-stack tests often manipulate the entire systems, going through the UI to the lower levels of the system. On this kind of tests it is often usually a good idea to use test doubles as calling remote systems are unnecessarily slow and brittle. Although the broad-stack tests exercise the whole system, we should use fewer of these as they are hard to maintain and much slower than component tests [4].

Listed above are some tests classifications that are relevant for the context of this dissertation, but there are more such *Story Tests*, *Contract Tests*, *Components Tests*, among others described by Martin Fowler [12].

[4] https://martinfowler.com/bliki/TestDouble.html
[5] https://martinfowler.com/bliki/ContractTest.html

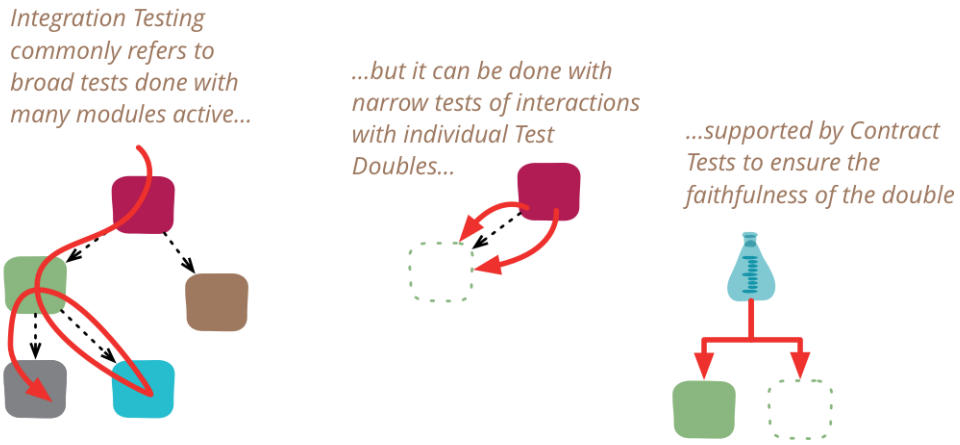

Figure 2.8: Integration Testing Sketching [7]

2.2.3 Test Design Techniques

There are two main techniques usually used to find errors, and the main difference between them stands on the working (source) code [16]:

- **Black-box testing**: is a testing technique employed without any knowledge of the source code application. It uses the main aspects or functionalities of the system for examination, not having any relevance with the internal logical structure of the software. The system under test will be treated as a "black box", but the tester must know the software architecture and its behavior (no source code details).

- **White-box testing**: it investigates the internal logic and the structure of the source code. This technique requires the testers to have complete knowledge of the source code, allowing them to find out a lot of implementation errors. One of the most significant advantages of this technique is to provide a maximum coverage of the system.

For large code segments, Black-box testing is much more efficient than White-box testing, however, Black-box testing provides a limited coverage of the system's behaviors and it requires clear specifications to be employed.

2.2.4 Testing in Outsystems Platform

Given the abstraction provided by visual languages like Outsystems, in association with continuous integrity validation that is built into the Outsystems platform, users will notice that the number of bugs introduced into code is much lower compared to other technologies, requiring fewer testing and fix cycles [42]. This accelerates the development and application delivery.

However, testing is still a necessary step during the application development. The Outsystems approach is to keep the Outsystems environment open so it is compatible with tools typically used such as *Ghost Inspector*, *Katalon*, *JMeter*, among others [22]. This way, testing is integrated in the continuous delivery cycle so there are no losses in productivity [28]. Another important aspect to highlight is Outsystems *Impact analysis and self-healing*. Outsystems tracks global dependencies and pinpoints the impacts of a change on all layers of an application, assuring nothing breaks when the application goes live, even if major changes are made on the application (data model, APIs and architecture). These capabilities automatically correct problems or inform developers of any corrections they must handle. At a broader level, Outsystems does impact analysis even for multi-applications, for example, preventing deployment from the test environment if it is missing a dependency in production [28]. The following tools are available to test applications in Outsystems [22]:

- **Unit Testing Framework**: Unit Testing Framework allows the user to develop and run unit tests for Outsystems Platform projects. This tool can be applied in a number of ways, depending on the scale and the complexity of the applications and its architecture. Unit Testing can be particularly effective for calculation engines and for business service components. Having a good set of unit tests for the system can help greatly when the user come to change or refactor a system [44].

- **Behavior Driven Development Framework**: the primary purpose of Behavior Driven Development framework (BDD framework) is to support Behavior-Driven Development, where all technical (e.g., developers) and non-technical (e.g., business analyst) participants in a software project collaborate to define a common understanding of how the software should behave.

  The BDD framework is a component that facilitates test automation where tests are specified using *Gherkin*[6] syntax. Gherkin is a human-readable language for structuring and describing application's expected behaviors. This scenario can be used to build BDD test automation for user's applications [47]. The main focus is testing the logic of user's modules, by exercising the critical actions that support the application's use cases. It also provides a set of tools for easily creating BDD/TDD (Test Driven Development) styles tests for Outsystems applications. This component creates test *scenarios* and *steps* that are conformant to the principles of BDD, Enables TDD, enhance test maintenance among other key-points [23]. Figure 2.10 shows an example of a Gherkin scenario:

  - ***Scenario***: describes the specific scenario that illustrates a business rule.
  - ***Given***: describes the initial context of the scenario. All the required preconditions that needs to hold before conducting the action or event to be tested.

[6] https://cucumber.io/docs/gherkin/

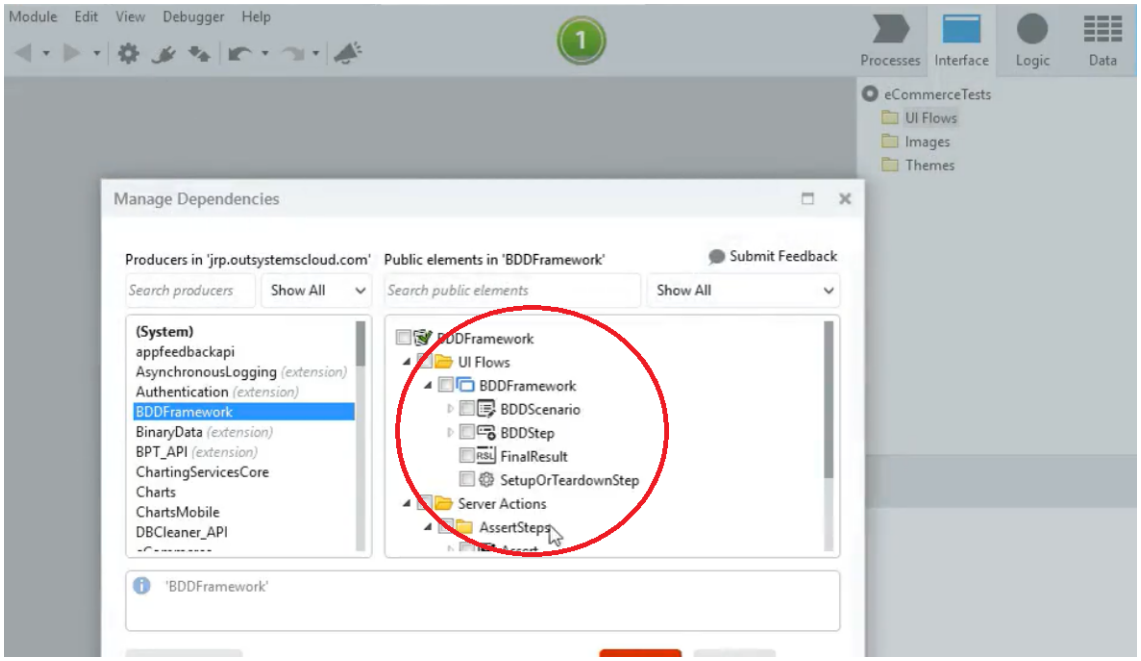

Figure 2.9: Service Studio: Web Blocks in *BDDFramework* [47]

- ***When***: describes the specific action or event. In many scenarios there should only be one such step. If more is needed, the user should consider to break up the scenario into two or more.
- ***Then***: describes the expected outcomes of conducting the action or event in the system. These steps commonly contain various assertions that verify everything we want to check as a result of this test.

This syntax requires the scenario to be clear to anyone who reads it, whether they are technical or non-technical participants. After the creation of the scenario, the user can start creating an automated BDD test using *BDDFramework*. The BDD testing framework includes four web blocks (Figure 2.9) that can be used to built tests [47]:

- ***BDDScenario***: each scenario is represented by a BDDScenario web block.
- ***BDDStep***: each group of steps is represented by a BDDStep web block.
- ***FinalResult***: returns stats about all scenarios run on the web screen. It should always be included at the end.
- ***SetupOrTeardownStep***: is a special kind of step that can be included in scenarios to perform setup or cleanup operations of data that is outside of the scope of the scenario from a functional or business perspective.

Finally, this framework can be used for Traditional Web and Service Applications, the server component of Reactive Web or Mobile Applications, and REST and SOAP APIs [47].

```
Scenario: Adding a product to the cart
Given:
That I have a cart
And there is a product called “Prosecco Armani DOC”
When:
I add the product to the cart
Then:
The operation should be successful
And the cart should have been correctly updated
```

Figure 2.10: Gherkin Scenario Example [47]

All industry testing practices can be used for testing Outsystems projects/applications, although some approaches produce better results than others. That is why Outsystems has a set of guidelines available for its users to follow:

- *Component or Unit Testing*: allows customers to independently validate small parts of an application. This is particularly effective for business service components.
- *Integration and API Testing*: focus on validating that parts of an application work together. This approach is particularly effective for complex systems.
- *Functional Testing*: is the practice of validating all components included in an application against its functional requirements, most frequently captured inside user stories[7].

Although a very useful tool for Outsystems platform environment, BDDFramework has its own limitations, for example, it does not support creation of test using Reactive Web or Mobile module, it does not support either tests being run in parallel over the same Outsystems environment, among other limitations [23]. These limitations are barriers to use BDDFramework as logic visualization tool, because the feedback that the developer needs must be acceptable (as little as possible) and with this framework it is very long. With BDDFramework we have to write the logic, then switch to BDDFramework environment, write some tests and run, and finally to see the result of our logic. That is why it is not a suitable solution for the context of logic visualization.

2.3 Logic Visualization

During the development of any application, the developers have the need to have feedback over the work that has been done. Developers aim for this feedback to be as immediate as possible, so they can visualize the effect of what they have been building and also to check if it is exactly as intended. When the subject to visualize is, for example *graphic components* (UI components, *Widgets*), The Outsystems Platform has available a preview

[7] https://www.atlassian.com/agile/project-management/user-stories

of the layout so the developers can check if the layout is exactly as expected. On the other hand, the subject to visualize could be the *logic*, which is exactly what this section will address. In order to visualize logic, most of the programmers use some approaches that are strongly related to *Software Testing*. Software Testing allows the programmers and the users to be confident about the software developed. Not only to be confident about the software being developed, but also to spot unexpected behavior from the software or to find out bugs, the developers use the Quality Assurance aspect of the tests. The following subsections covers some testing methods that are used by the developers in order to obtain feedback of the logic while developing an application [12].

2.3.1 Exploratory Testing

Exploratory testing explores the characteristics of the software, raising discoveries that will be classified depending on the behavior that can be considered either reasonable or a failure. Exploratory testing is a style that emphasizes a rapid cycle of learning, test design and execution rather than trying to verify if the software conforms to a pre-written test script. Exploratory testing is the opposite of *scripted testing*, since as in scripted testing, test designers create a script of tests that are executed everytime the developing code is manipulated. These scripts can be executed by different users (not necessarily the person who wrote them) and if any test shows different behavior from what is expected, it will be considered a *failure*. Before the influence of *Extreme Programming*[8], the scripted test was executed following the script by testers and checking the result. Extreme Programming allowed the automation of the previous method, delivering a faster execution and elimination of the human error involved in evaluating expected behavior. However, even the most exhaustive automated test suite has its own limitations. Scripted testing can only verify what is in the script, referring only the conditions that are known about. It is a very good technique to spot bugs that try to go through it, but there is no guaranty that it covers all it should.

Exploratory testing emerges as a technique that seeks to test the boundaries of the scripted test suite, finding new behaviors not covered by the script. The failures found by this technique can often be added to the scripts. Although Exploratory testing is a much more informal process than the scripted testing, it still requires a very disciplined way to be done well. A very good way to do it is in time-boxed sessions, where the sessions focus on a particular aspect of the software. Exploratory testing involves trying things, learning more about what the software does and using the learning to generate questions and hypotheses, so it can be generated new tests in the moment to gather new information. This a technique that requires skilled and curious testers, who are comfortable with learning about the software and generating new test designs during the session. It also requires for the testers to be observant, and to look up to any behavior that might seem odd. It also is an activity to be done regularly during the software development [6].

[8] https://martinfowler.com/bliki/ExtremeProgramming.html

There are some particular ways of Exploratory Testing that are used by developers in order to visualize logic while developing software:

- **Debugger**: is a tool that offers a closer look into the execution as it allows the programmer to work through the code line-by-line to find out what is going wrong or unexpected, where and why. With the debugger the programmer is able to interact with the flow of the execution, changing the code flow whilst running, stopping whenever needed and controlling all the execution. Some debuggers require that the code to be debugged must be compiled with the debugging information inserted. This information is provided by *debugging symbols* included by the compiler in the binaries and they describe where functions and variables are stored in memory. These symbols normally make the executable run a bit slower, but still runs as a normal program.

  Debuggers have an important feature which is the possibility to set *breakpoints* that allow the developer to stop the program execution on demand: the program runs normally until the execution is at the same point as the breakpoint address. After the breakpoint is reached, the execution drops into the debugger to look at the variables or even to continue the execution. The breakpoints can be set to beginning of functions, at specific addresses or specific line number. After the execution stops as consequence from a breakpoint, the debugger can execute the next program line *stepping over* any functions calls in the line as a single instruction. On the opposite side, the debugger can *step into* a function call and execute all its code, line-by-line. There is also available in the debuggers the possibility to set *watchpoints*, which are a particular type of breakpoint that stop the execution whenever a variable changes (even if the current line does not refer to the variable explicitly). Differently from breakpoints, watchpoints look at the memory address and notify the programmer if something is written to it [1].

  Errors that are made during the implementation of algorithms are reasonably easy to track with debuggers, and that is one of the keypoints for the use of debugger to visualize logic. Programmers use debugger to spot bugs and wrong behavior on their software which also allows them to visualize part of the software logic. Although debuggers are normally used to find out bugs in algorithms it is unusual to use them to certify that some logic is correct. It is not used to test correctness and its scope can be limited. Also, debugging is a process that can be very exhaustive because it requires human involvement and a large amount of time and resources [15].

- **Manual UI Testing: Real Application**: when the application is already finished or it has one of its first versions implemented, including the UI, there is something that is usually done by its developers: use the brand new application functionalities in order to look at the behavior and check if it is exactly as expected. This seems to

be a very simple task, but it can actually be a good technique to verify if the product meets its goals. If done very carefully and by experienced testers it can be very useful to spot errors. Experienced testers hold a lot of knowledge acquired from past projects that can be used to test critical aspects of the applications. This technique is a kind of high level testing method that allows the developers to visualize the logic behind their application.

However, this is a method that requires the software to be finished or having one implemented version (including the UI) in order for the developer to use the application to test the behavior. This aspect discourages the developers to adopt this method as the main technique to visualize the application logic as it requires the application to be implemented. The feedback to observe the logic is too big, not allowing this technique to succeed as one of the main approach.

- **Manual UI Testing: Dummy UI**: this method is very similar to the previous one. The keypoint of this approach is also to test the application functionalities to check if they are in conformance with the expected. Nevertheless it has a significant difference: this method does not require the application to be fully implemented, neither to have a version of the application with the UI implemented. The developer can create a *dummy* UI just for the purpose of testing the functionalities. This notably reduces the feedback time in comparison with the real application testing.

  These properties bring good advantages in terms of time and productivity as it allows the developers to have an early feedback of the logic, but it is not the best solution because the feedback still takes some time. The developer needs to spend time creating a dummy component to visualize the logic/behavior of the application and sometimes even the dummy component will require some aspects that are not related to the logic that the developer wants to visualize.

2.3.2 Logging and Monitoring

Logging is a concept that refers to the printing of messages that normally is an aid to debugging. Logging is the automatic recording of information messages or events with the intent of monitoring the status of a program and to diagnose problems. Some systems use this approach because the failure can affect the correctness of the whole system. It is also used to evaluate products about their reliability [1].

Logging brings a concept of "*Observability*", that is, for example, the tracking of things like memory, CPU utilization, network and disk I/O, thread counts. It can also track things related to business or domain as session duration and payment failure rate, for example. Besides, product-oriented metrics are more valuable to observe because they closely reflect that the systems is performing in conformance to its business goals [5]. Logging is a very powerful way of gathering data about a system. They are no longer just text files to look up in case something goes wrong. As mentioned, it is not restricted to

technical data, it can also log valuable data. There are some tools available like *Splunk*[9] that can build indexes based on the kind of the logs (*ERROR*, *WARN* and *INFO*) and offer optimised search, aggregation and visualisations. When the developers look at the log files, they should be able to filter the results to see just errors, for example, in order to ensure that nothing has gone wrong [51].

During the development of a software product, developers take advantage of this approach to visualize software logic and to identify the errors that have been made. This technique, if well used, allows the access to precious and structured information in case of error or even information about the business logic. Despite very useful, Observability requires hand-rolled instrumentation logic and the instrumentation code can be very noisy, easily leading to a distracting mess [5]. These characteristics make this method an unreasonable strategy to visualize logic in Outsystems Platform.

2.3.3 Specification By Example

Martin Fowler [8] characterizes SBE as an approach used in *Agile software development*[10] for specifying requirements and tests that are supposed to be general and cover all cases. Examples are very important, but they highlight only few points, being the developer/tester the one who should infer generalizations. Rigorous specifications refers the ones that are found to be passed or failed and the main idea is to use pre and post conditions. These methods can be used in some cases, but they are not an ideal choice because, depending on the scenario, the pre-post conditions can be easy to write or very tricky. Examples are much easier to come up with, particularly for non-technical people and that is a good inspiration for this technique. These tests have an important property which is the involvement of double-checking that is a vital principle, and much used by humans. Combining SBE with production code increases their ability to find errors. It is important to highlight that, although test are very important for any software product, they are incomplete. Tests should be backed up with other mechanisms. This clarifies that SBE is not enough and we need to do more to ensure that everything is working as is it supposed to [8].

Specification By Example plays an important role on application logic visualization as the developers can use SBE to spot misbehavior during the software development or even to gain confidence about the correctness. Nevertheless, this approach requires **automation** in order to be used as a solid candidate to play the main role. Testing is not practical to be done manually since tests are vulnerable to inaccurate results and manual test can be extremely slow, even when it is used in the context of logic visualization. On other hand, manual tests are difficult to maintain by developers and require a significant amount of time and human resources, making automation a crucial need.

[9] https://www.splunk.com/
[10] https://en.wikipedia.org/wiki/Agile_software_development

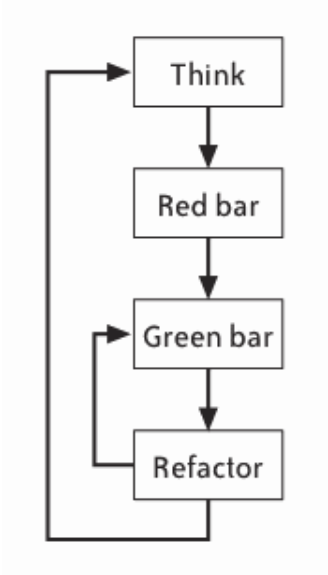

Figure 2.11: The TDD Cycle [49]

As users, every one of us interacts with an ever-increasing amount of software everyday, and if developers wants to keep pace, they should look into ways of delivering software faster without sacrificing its quality. Building, testing and deploying an ever-increasing amount of software manually becomes an impossible task. It emerges the need to automate everything: from tests building, deployment and infrastructure. The traditional way of deploying the application into a testing environment and check everything manually is no longer an option as it is time-consuming and repetitive [10]. All that has been said also applies to the context of logic visualization. The developers need to obtain feedback of the logic being built almost immediately, otherwise it can compromises their productivity.

Some frameworks previous mentioned such *BDDFramework* and *Unit Testing Framework* that are used in Outsystems Platform combine aspects of SBE and testing automation, but they still not accomplish the goal of making the logic feedback cycle as fast as desired in Outsystems Platform.

2.3.4 Test-Driven Development

Test-Driven Development (TDD) is a technique for building software where tests guide the software development, developed by Kent Beck as part of Extreme Programming in 1990 [9]. In TDD, tests are the core concept because they lead software development. With this approach where development is driven by tests, once a test is working, we are one step closer to have everything working than before, and this is an important aspect of TDD. The general cycle (Figure 2.11) of TDD is:

- *Write a test (Red Bar)*: in this step the developer needs to imagine how the piece of code being developed will appear in the code, even with an imaginary interface on

his head. We should include all elements necessary to calculate the right answers into the story he is imagining.

- *Make it run (Green Bar)*: the goal here is to make the test work. If a simple, clean solution is obvious, we just type it. If the simple and clean solution is obvious but it will take sometime, then we make a note and come back to it later because the objective here is to make test passes, even if it is just for a moment.

- *Make it right (Refactor)*: at this point the system is working as expected, and it is time to do things right by eliminating duplication introduced previously in order to get a quick green (test correct). At this point we have to refactor our code.

The goal of TDD is clean code that works. However, this is not easy to achieve even for the best and experienced programmers, and that is why we should split tasks into smaller ones (divide to conquer) [2]. Nevertheless, TDD's benefits sometimes may not be achieved and one the most common reasons is neglecting the third step. If the code is not refactored, it can end up being a messy aggregation of code fragments [9].

There was a time where programmers hand-checked their code in case of some compilations errors looking for misplaced characters, but currently, most of the IDEs check the syntax as we type. The feedback loop to catch these errors is so fast that if something does not compile we know exactly where to look at. TDD applies the same principle to programmer intent by verifying if the code does what is expected and in case something goes wrong, there are not many lines to check, making mistakes much easier to find and reducing the feedback time to visualize logic. This technique is very important to the concept of logic visualization as it allows the developers to be confident about their logic written, notifying them in case of errors and facilitating the search of mistakes by reducing the number of lines to be checked. TDD also takes advantages of Unit tests (and unit testing tools) during the process because, as already mentioned, unit tests are the lowest-level tests and they are very fast to run [49].

2.3.5 Read-Eval-Print-Loop (REPL)

REPL basic idea is to process a sequence of commands one by one, and display the results to the user. The main phases of the loop are explained as follows (in an approximation from the LISP[11] interpreter) [50]:

- **READ**: process the syntax of the expression: *internalize* it as semantic operation of the state. This phase must be an operation that terminates quickly and the syntax errors need to be encoded into the result.

- **EVAL**: evaluate the internalized expression in the current state: *run* it and update the top level state accordingly.

[11] https://en.wikipedia.org/wiki/Lisp_(programming_language)

- **PRINT**: output the result of the evaluation: *externalize* values, usually in the same notation as the input. However, the original intention of the REPL model is to externalize the result of the evaluation in a human-readable form.
- **LOOP**: continue the cycle above until the user terminates it.

If developers have a REPL environment available during software development where they can use the environment to write code (with their logic) and the environment is able to evaluate the expression given a set of inputs and present the result of the logic (applied to the given inputs) to the developer, we have then a way of visualizing logic (during the development). If the feedback loop is small enough, then we achieved our goal of reducing the logic visualization feedback to the lowest possible.

This approach brings another discussion to the context of logic visualization: how to *evaluate* the logic so that the feedback time is as short as possible? This discussion can be matter of **Compilation and Interpretation**.

Compilers and Interpreters are essential programs, whose inputs (sometimes even outputs) are programs. In general, program's actual behavior are dependent from their inputs values and the abstract behavior are defined in relation to the inputs values, formally by types and conditions or informally by programmers. Interpreters can be seen as virtual machines and compilers as programs translators. A compiler transforms the source code language (higher abstraction level) into the target language (lower abstraction level) that can be executed by a machine. Interpreters take as input the abstract representation of the the input, and evaluate it with relation to the input data. The output of the interpreter is the output produced by the program being executed. Bellow we can see the comparison between compilers and interpreters accordingly to the following aspects [48]:

- **Efficiency**: usually, executing a code generated by a compiler is faster and safer than the interpretation of the same code. The fetching and execution cycle is hardware based, hence faster. Many times the interpreted languages are dynamically typed, meaning that more time is spent at runtime checking the operands while the static ones do not need any checking at runtime.
- **Safety**: interpreted languages are sometimes considered to be unsafe, because errors are only detected at runtime. Nevertheless, the combination of the compiler checks and absence of running checks can be a gateway to build unsafe languages.
- **Flexibility**: by taking advantages, for instance, of the dynamic loading of modules, interpreters usually provide a more flexible development process, allowing for the immediate update of new code in a running installation.

Therefore, the success of using a REPL environment to visualize logic depends on how the *evaluate* phase is implemented, which may be with interpretation, compilation or another approach suitable to the context.

Chapter 3

Proposed Work

This chapter presents the work to be developed during this dissertation, by explaining the details of the problem to address, the methods thought to address it and also to explain the goals to be achieved with this work. It also contains the validation of the work and the planning on how tasks will be slip to achieve the desired goals.

3.1 Logic Previewer

As software development increases day by day, the goal of the companies that build software products is to increase productivity as much as possible in order to respond to high demand. Outsystems is improving everyday its method to satisfy its clients with excellent products and short delivery time, greatly due to improvement of the software development environment. As already mentioned, VPLs accelerate the development process by reducing the concerns of the users with implementation details, consequently improving their productivity. Despite the software development process in Outsystems Platform being fast, there is an important problem that needs to be solved: *The feedback loop of logic visualization*. How can we improve the cycle of feedback received by developers when writing their logic in Outsystems Platform? This is a question that we will try to answer during the implementation of this work.

3.1.1 The Problem

Each amount of time that can be reduced during software development is a significant asset for productivity, even small reductions may present significant value. Therefore, all processes (during development process and not only) should be carefully and continuously analyzed in order to obtain some improvements. Developers at Outsystems, when writing their code (such as Outsystems action logic) do not have an immediate nor

short-term feedback of the logic's behavior. This is the key problem to be address in this work: During development, users (developers) do not know immediately, or even in a short time what are the effects of the logic being written, which can significantly impact the user's productivity.

But how can logic be visualized at Outsystems? Developers at Outsystems have some approaches to visualize the behavior of the logic they are writing, which can be shown bellow:

- **By testing**: developers at Outsystems (and not only) use testing properties to check the behavior of applications and to be able to spot bugs introduced during development. Section 2.3 presented different strategies to visualize logic and developers use many of them at Outsystems. Starting by **Manual UI testing using real application**, which is an approach used to check the application behavior by using the finished (or a released version) of the product UI. Although with this strategy the developer is able to spot some misbehavior and have a clear notion of the functionalities usage, it is not a suitable approach to visualize logic during development, as it requires the software product to be implemented or at least have a released version. Similarly, we have the **Manual UI testing with a dummy application** whose concept is almost the same as the real app testing but it does not require a finished version of the UI, hence reducing some feedback time. However, this approach also requires the implementation of a dummy UI for testing proposes, delaying the feedback time to the developer. In addition, both approaches mentioned are within Exploratory Testing and, as mentioned, it is a technique that requires high levels of experience and expertise from application developers.

  Still on testing, developers at Outsystems also use testing frameworks to test their software and to check the application's behavior. **BDDFramework** and **Unit Testing Framework** (presented in 2.2.4) are the most used in Outsystems platform for testing applications. We saw that both are good tools within the testing context, but when it comes to visualizing the logic they are nowhere near a reasonable solution because of their previously mentioned limitations (in 2.2.4) such as the incompatibility of using BDDFramework with the Outsystems environment at the same time, not allowing the developer to visualize the effects of the logic being developed.

- **With the debugger**: the debugger is a very useful tool in Outsystems platform as it allows to have closer look into the applications execution, which is used to spot misbehavior and bugs in the software. Nonetheless, debugging requires the software to be published, then a browser or a mobile device to check the execution flow. Also, as mentioned in the previous chapter, debugging is a process that requires large amount of time and resources and its not suitable for checking software correctness.

In summary, all the approaches used in Outsystems for visualizing logic described above have their particular limitation and all of them have the one main problem in

the context of logic visualization: **The feedback cycle to visualize logic during development is extremely high**.

3.1.2 Goals

The main goal of this work is to **enable the preview of logic** in Outsystems development processes. By doing this, we aim to reduce the **feedback loop** that developers experience during the development process (creating and changing code) in Outsystems environment.

We have presented how logic is currently visualized in Outsystems development processes and it was shown how long these processes are. The problem presented above is a real barrier for productivity and it can be noticed more easily in large applications. This work introduces a new concept in Outsystems development processes: **Logic Previewer**. We want to allow the developers to create/change their code and be able to preview its behavior immediately, thus improving the feedback cycle that they have about the code logic being developed and consequently their productivity. The solution being proposed, aims:

1. To enable the preview of the logic (the behavior) being written by developers in their application during Outsystems development.

2. To reduce the feedback loop that developers have of their logic being implemented, which is too long.

3.1.3 Proposed Approach

This section presents a description of the approach to be developed in this work in order to solve the problems highlighted above. This approach aims to improve the experience of Outsystems users when developing software by allowing them to visualize their application behavior and reducing the feedback loop they have when creating and changing code. The process of creating software, compiling it, then publishing into Outsystems server, and finally testing to check its behavior (by using the application or with a testing framework) is very long and hampers user productivity.

Currently in Outsystems, when users want to test an application behavior, they need to follow some approaches described previously in this document such as testing the application directly with its UI, testing with a dummy UI, among others. In case of using testing frameworks (which is one of the fastest approaches among the existing ones), one of their limitations is that they require the code (being developed by user) and the tests to be published (compiled and deployed).

Publishing modules in Outsystems is a costly operation, specially when we have large-scale applications with large modules. This process takes significant time, and when developing code, users are constantly changing the logic to obtain the expected behavior from applications. Every time users make a change they have publish the whole module,

having to wait for this process to finish, which is a very time-consuming process. As the module gets bigger, its dependencies also grow, making the feedback loop experienced by users that wish to view the effects of changes on their application even longer. The proposed work aims to change the status quo of logic visualization, by putting together the following stages:

1. Interpret or **partially** compile the modified code (actions, modules) **without unnecessary dependencies**.

2. Test the logic in the development environment **without publishing neither the code or the tests**.

3. show the results of the logic to the user (developer) **immediately**.

The first stage of the proposed work (**stage 1**) will start by tackling the question about compiling a module in a way that allows for separate executions. Here we have some hypotheses that will be considered to address this problem using **interpretation** techniques. By using Interpretation in this stage we can have some limitations with respect to external dependencies of the actions or module to be interpreted (dependencies from internal actions/modules in Service Studio environment can be interpreted along with the modified code). Another hypothesis is to use **partial** compilation in this stage by removing extra dependencies from the code to be compiled in order to obtain a faster compilation. However, in **stage 1**, one of the main requirements of the proposed work is to remove the code dependencies that do not have impact on the created/changed code. It is important to highlight that Outsystems compiles its modules, meaning that using interpretation is a process that will be developed from scratch.

In **stage 2**, we will test the created/changed logic (code being developed) in order to check if the behavior is as expected. As mentioned before, in Outsystems, testing is employed by first publishing the module (with the created code) and then test the code, which is a time-consuming process, specially during development process (where changes are frequent). This work intends to test the application logic without publishing neither the code or the tests, introducing then the concept of *Logic Previewer*. This is an important step to reduce the feedback loop users experience as publishing is a extremely costly process. We can also use *Mocks* to substitute external services (including databases, file systems, etc.).

We already have started working on this stage by analysing a data set with information retrieved from Outsystems customer applications (applications developed using Outsystems environment that include modules, flows, actions, nodes, etc.) in order to study existing inputs/outputs, actions side-effects, input/output types, predominance of external dependencies, among other relevant aspects to the context. We used Python language to analyse the data set (that has more then 5GB of data) and produced some counts in order to have detailed information of the elements mentioned before. A *.csv* file

was generated with some relevant counting and uploaded to a database where next we intend to carry out some deeper studies (queries, patterns, etc.).

Finally, in **stage 3** we will show to users the results of the tests where they receive the immediate feedback of their application behavior. By concluding these 3 stages of the proposed work we expect to have a feedback loop that is shorter (immediate feedback) and enable preview of the application logic during development. Depending on the available time to implement this proposed solution, the logic visualization (presented by this solution) could be enhanced by adding to the work a notion of **traceability** by showing to users the execution details (e.g. variable values) of their software components. This would be useful for users to have richer feedback on the behavior of the software.

3.2 Evaluation

We will evaluate our work by attempting to measure the benefits it introduces to the Outsystems environment. Therefore, to obtain concrete results we will organize experiments where real users build software in the current Outsystems development processes and also with the proposed solution. From these experiments we will gather some important metrics such as development time, bugs found in the final code, among other important aspects, and finally make an accurate analysis to compare the results.

Another key point is that the benefits of our work strongly depend on the kinds of actions (code) that users are able to effectively preview, and how prevalent are actions that cannot be previewed by our work. To this end, we plan to make a statistical evaluation of the preview coverage of our work, using a large data set of real-world Outsystems code.

3.3 Work Plan

In order to successfully complete this work, we propose to divide the work into the following tasks:

1. Analyze and quantify action types within the universe of OutSystems customers so that we can prioritize and target a wider range of actions in the logic previewer.

2. Study the Outsystems compiler internal pipeline, with an emphasis on dependency tracking.

3. Recover and perform an assessment on Test Action work. Test Action is a capability that was introduced as proof of concept in the product [13] that allows to just-in-time (JIT) compile and execute a single OutSystems action within a module.

4. Determine feasibility of partial compilation vs interpretation approach to logic preview.

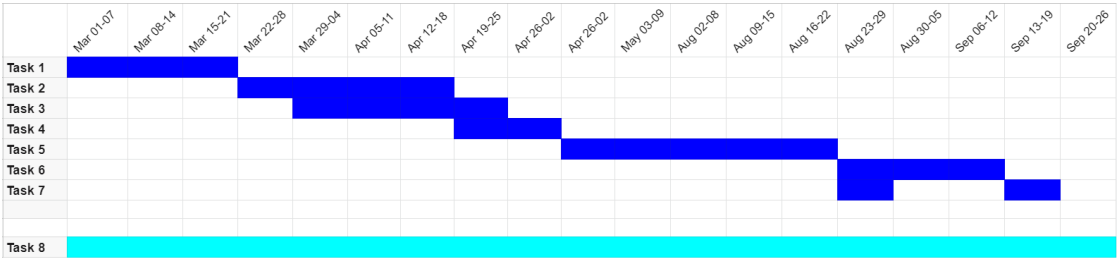

Figure 3.1: Task Schedule

5. Implement a proof of concept interpreter/JIT compiler focused on executing server actions with inputs provided from test cases.

6. Prototype an experience how this solution could be integrated in Service Studio

7. Validate the approach and benchmark it against Test Action and the existing compile and publish development cycle times.

8. Write the dissertation.

After listing above the main tasks of the work, we present an estimated duration of each one of them in the Figure 3.1.

Bibliography

[1] P. Adragna. "Software debugging techniques." In: *Inverted CERN School of Computing, iCSC 2005 and iCSC 2006 - Proceedings* (2008), pp. 71–86.

[2] K. Beck. "Test-Driven Development By Example." In: c (2002).

[3] A. Bertolino and E. Marchetti. "A Brief Essay on Software Testing." In: *Area* (2003), pp. 1–14.

[4] M. Fowler. *BroadStackTest*. https://martinfowler.com/bliki/BroadStackTest.html. (Accessed on 02/18/2021).

[5] M. Fowler. *Domain-Oriented Observability*. https://martinfowler.com/articles/domain-oriented-observability.html. (Accessed on 02/12/2021).

[6] M. Fowler. *Exploratory Testing*. https://martinfowler.com/bliki/ExploratoryTesting.html. (Accessed on 02/11/2021).

[7] M. Fowler. *Integration Test*. https://martinfowler.com/bliki/IntegrationTest.html. (Accessed on 02/17/2021).

[8] M. Fowler. *Specification By Example*. https://martinfowler.com/bliki/SpecificationByExample.html. (Accessed on 02/15/2021).

[9] M. Fowler. *Test Driven Development*. https://martinfowler.com/bliki/TestDrivenDevelopment.html. (Accessed on 02/19/2021).

[10] M. Fowler. *The Practical Test Pyramid*. https://martinfowler.com/articles/practical-test-pyramid.html?ref=hackernoon.com. (Accessed on 02/16/2021).

[11] M. Fowler. *UnitTest*. https://martinfowler.com/bliki/UnitTest.html. (Accessed on 02/17/2021).

[12] M. Fowler. *Testing Guide*. https://martinfowler.com/testing/. (Accessed on 02/11/2021). Dec. 2019.

[13] G. Guerra. "Testing support for the OutSystems Agile Platform." In: (2010).

[14] R. Marvin. *OutSystems Review | PCMag*. https://www.pcmag.com/reviews/outsystems. (Accessed on 02/22/2021).

[15] Minigranth. *Software Debugging | Software Testing Tutorial | Minigranth*. https://minigranth.in/software-testing-tutorial/software-debugging. (Accessed on 02/12/2021).

[16] K. Mohd. Ehmer and K. Farmeena. "A Comparative Study of White Box , Black Box and Grey Box Testing Techniques." In: *International Journal of Advanced Computer Science and Applications* 3.6 (2012), pp. 12–15. ISSN: 1098-6596. arXiv: arXiv:1011.1669v3.

[17] OutSystems. *Actions in Reactive Web and Mobile Apps*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Implement_Application_Logic/Actions_in_Reactive_Web_and_Mobile_Apps. (Accessed on 02/05/2021).

[18] OutSystems. *Actions in Web Applications*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Implement_Application_Logic/Actions_in_Web_Applications. (Accessed on 02/08/2021).

[19] OutSystems. *Aggregate*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Data/Handling_Data/Queries/Aggregate. (Accessed on 02/05/2021).

[20] OutSystems. *Application Layers < Service Studio Overview - Training*. https://www.outsystems.com/training/lesson/2186/application-layers. (Accessed on 02/04/2021).

[21] OutSystems. *Assign*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/Assign. (Accessed on 02/08/2021).

[22] OutSystems. *Automated Testing Tools*. https://success.outsystems.com/Documentation/Best_Practices/OutSystems_Testing_Guidelines/Automated_Testing_Tools?_gl=1*tch51t*_ga*MTAxOTI4MjgOMC4xNjA2NTg2NTgy*_ga_ZD4DTMHWR2*MTYxMjg4NDMwMy4zMi4xLjE2MTI4ODg5MzMuMjY.. (Accessed on 02/09/2021).

[23] OutSystems. *BDDFramework - Overview*. https://www.outsystems.com/forge/component-overview/1201/bddframework. (Accessed on 02/09/2021).

[24] OutSystems. *End*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/End. (Accessed on 02/08/2021).

[25] OutSystems. *Entities*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Data/Data_Modeling/Entities. (Accessed on 02/08/2021).

[26] OutSystems. *Extensions*. https://success.outsystems.com/Documentation/11/Extensibility_and_Integration/Extend_Logic_with_Your_Own_Code/Extensions. (Accessed on 02/04/2021).

[27] OutSystems. *For Each*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/For_Each. (Accessed on 02/08/2021).

[28] OutSystems. *How does OutSystems support testing and quality assurance? | Evaluation Guide*. https://www.outsystems.com/evaluation-guide/how-does-outsystems-support-testing-and-quality-assurance/?origin=d. (Accessed on 02/09/2021).

[29] OutSystems. *If*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/If. (Accessed on 02/08/2021).

[30] OutSystems. *Modular Programming < Intro to OutSystems Development - Training*. https://www.outsystems.com/training/lesson/2159/modular-programming. (Accessed on 02/04/2021).

[31] OutSystems. *OutSystems Again Named a Leader in Gartner's 2018 Magic Quadrant for Enterprise High-Productivity Application Platform as a Service*. https://www.outsystems.com/News/high-productivity-apaas-gartner-leader/. (Accessed on 02/03/2021).

[32] OutSystems. *OutSystems development and management tools | Evaluation Guide*. https://www.outsystems.com/evaluation-guide/development-and-management-tools/. (Accessed on 02/03/2021).

[33] OutSystems. *OutSystems Evaluation Guide*. https://www.outsystems.com/evaluation-guide/. (Accessed on 02/03/2021).

[34] OutSystems. *OutSystems Platform Services | Evaluation Guide*. (Accessed on 02/05/2021). URL: https://www.outsystems.com/evaluation-guide/platform-services/#1.

[35] OutSystems. *Platform Runtime | Evaluation Guide*. https://www.outsystems.com/evaluation-guide/platform-runtime/. (Accessed on 02/03/2021).

[36] OutSystems. *Screen*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Design_UI/Screen. (Accessed on 02/05/2021).

[37] OutSystems. *Screen Templates*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Design_UI/Screen_Templates. (Accessed on 02/05/2021).

[38] OutSystems. *Service Studio Overview*. https://success.outsystems.com/Documentation/11/Getting_started/Service_Studio_Overview. (Accessed on 02/03/2021).

[39] OutSystems. *Service Studio Walkthrough < Service Studio Overview - Training*. https://www.outsystems.com/training/lesson/2185/service-studio-walkthrough. (Accessed on 02/04/2021).

[40] OutSystems. *Start*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/Start. (Accessed on 02/08/2021).

[41] OutSystems. *Switch*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools/Switch. (Accessed on 02/08/2021).

[42] OutSystems. *Testing OutSystems Applications | Evaluation Guide*. https://www.outsystems.com/evaluation-guide/testing-outsystems-applications/. (Accessed on 02/09/2021).

[43] OutSystems. *The Modern Application Development Platform*. https://www.outsystems.com/platform/. (Accessed on 02/03/2021).

[44] OutSystems. *Unit Testing Framework - Overview*. https://www.outsystems.com/forge/component-overview/387/Unit+Testing+Framework/. (Accessed on 02/09/2021).

[45] OutSystems. *Web Logic Tools*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Traditional_Web/Web_Logic_Tools. (Accessed on 02/08/2021).

[46] OutSystems. *Why OutSystems? | Evaluation Guide*. https://www.outsystems.com/evaluation-guide/why-outsystems/. (Accessed on 02/04/2021).

[47] J. Proenca. *Your Complete Guide To BDD Testing In OutSystems*. https://www.outsystems.com/blog/posts/bdd-testing/. (Accessed on 02/09/2021).

[48] J. C. Seco. "Interpretation and Compilation of Programming Languages Part 1 - Overview." In: (2014), pp. 1–11.

[49] J. Shore. *The Art of Agile Development: Test-Driven Development*. http://www.jamesshore.com/v2/books/aoad1/test_driven_development. (Accessed on 02/19/2021).

[50] M. Wenzel. "READ-EVAL-PRINT in Parallel and Asynchronous Proof-checking." In: *Electronic Proceedings in Theoretical Computer Science* 118 (2013), pp. 57–71. ISSN: 2075-2180. DOI: 10.4204/eptcs.118.4.

[51] R. Wilsenach. *QA in Production*. https://martinfowler.com/articles/qa-in-production.html. (Accessed on 02/12/2021).