



**Nuno Miguel André Pulido**

Bachelor in Computer Science

## **Applying Behavior Driven Development Practices and Tools to Low-Code Technology**

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

Master of Science in  
**Computer Science and Informatics Engineering**

Adviser: Miguel Carlos Pacheco Afonso Goulão,  
Assistant Professor,  
NOVA University of Lisbon

Co-adviser: João Rosa Lã Pais Proença, Quality Owner,  
OutSystems

Examination Committee

Chairperson: Prof. Pedro Medeiros, FCT-Nova  
Members: Prof. João Pascoal Faria, FEUP  
Prof. Miguel Goulão, FCT-Nova



FACULDADE DE  
CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE NOVA DE LISBOA

September, 2019



## **Applying Behavior Driven Development Practices and Tools to Low-Code Technology**

Copyright © Nuno Miguel André Pulido, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.

A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.



*To my family.*



## ACKNOWLEDGEMENTS

I would like to start by thanking my advisors, Professor Miguel Goulão from FCT and João Proença from OutSystems. Thank you Professor for being available to respond to all my email spam, for regularly visiting me at OutSystems to follow my work, and for all the help writing this dissertation. Thank you very much for your patience and all the advises in the reviewing of the report ... I promise I will be careful and try to avoid very long sentences and the excessive use of the passive voice. Thank you João for always being by my side (literally) throughout this research, for all the technical support and for the helpful advice you have been giving, in a topic you were more sensitive than anyone else. You were undoubtedly the right person to guide this dissertation and I am very grateful for the opportunity you gave me.

I would also like to give a very special thanks to all the FCT teachers I have been through during this course. Without their lessons it would not be possible to do this dissertation. To the people at OutSystems, and especially to the Life-Cycle team members who have always been available to help me in my work and for welcoming me in the team, as one of their own. The time spent at the company was very pleasant and I am genuinely surprised by the positive and helpful spirit that exists at OutSystems. Speaking of OutSystems, of course I could not forget to thank my mates from the “All-together” team. We entered as strangers and left as a group of friends for life! Thanks for all the meal-time deep conversations, billiard sessions, pranks, night outs, concerts, team lunches, football matches, video calls....

Finally I want give a big thank you my parents for giving me this opportunity to study away from home and for everything they did for me. Without them none of this would be possible. A big one also to my friends, both from the Algarve and from the FCT, and of these last I would like to highlight Pedro and Sergio who always accompanied me throughout this last five years, without forgetting all the others of course. A big thank you to Rita and finally a big thank you to Bia aka *Quidditch Captain*. They are amazing.





*“If you are working on something that you really care about,  
you do not have to be pushed. The vision pulls you.”*

*– Steve Jobs*



## ABSTRACT

---

One of the main reasons software projects fail is the lack of communication between stakeholders. Low Code Platforms have been recently growing in the software development market. These allow creating state-of-the-art applications with facilitated design and integration, in a rapid development fashion and as such, avoiding communication errors among stakeholders is indispensable to avoid regressions. Behavior-Driven Development (BDD) is a practice that focuses on developing value-based software, promoting communication by bringing business people into development.

The *BDDFramework* is an open-source testing automation framework within the OutSystems environment. It allows describing test scenarios using the Gherkin syntax but it is not focused on enabling the BDD process. Our main challenge is: - *How can we apply the BDD process in Low Code and support it from a technological point of view, considering the particularities of Low Code environments and having as case study the OutSystems platform? Is the BDDFramework prepared for this?*

We interviewed some people in the domain to understand their development and testing challenges and their experience with the *BDDFramework*. With the information gathered and after studying other existing BDD process supporting tools for other languages, we built a prototype that uses the existing *BDDFramework* and automates it, allowing scenarios to be described in text files, which helps the introduction of business people in the process. The prototype generates all the test structure automatically, reusing equal steps while detecting parameters in the Gherkin descriptions.

We performed some real user testing to validate our prototype and we found that our solution was faster, easier, with better usability and we obtained more correct tests than with the previous approach - the *BDDFramework* alone.

Testing in Low Code is still an area with a lot to explore and errors have a huge impact when development is very accelerated, so as communication errors tend to decrease we can start building software even faster and this is what BDD proposes to solve. With this prototype we have been able to demonstrate that it is possible to build a framework that will allow us to enable this process by taking advantage of the OutSystems language particularities to potentiate the BDD practice technologically, while setting a test standard for the OutSystems language.

---

**Keywords:** Low Code Development, Behavior-Driven Development, OutSystems, Gherkin Scenarios, Automation Frameworks

---

## RESUMO

---

Uma das principais razões para o insucesso dos projectos de software é a falta de comunicação entre as partes envolvidas. As plataformas *Low Code* têm vindo a crescer no mercado do desenvolvimento de software nos últimos anos. Estas permitem criar aplicações recorrendo a uma linguagem visual que facilita o design e a integração num ritmo de desenvolvimento acelerado, o que faz com que seja fundamental potenciar a boa comunicação entre as partes interessadas, de forma a evitar regressões. O *Behavior-Driven Development* (BDD) é uma prática que tem como objectivo o desenvolvimento de software com valor, promovendo a comunicação e aproximando as partes envolvidas da fase de desenvolvimento.

A OutSystems tem uma ferramenta para automação de testes, na qual os cenários são descritos na sintaxe Gherkin, a *BDDFramework*. No entanto, e apesar do nome, esta não foi criada com o propósito de auxiliar na prática do BDD, mas apenas e só para efeitos de automação de testes. O nosso principal objectivo com esta dissertação é perceber como podemos potenciar o processo de BDD em OutSystems, de um ponto de vista tecnológico e perceber se a *BDDFramework* está preparada para isto.

Entrevistámos várias pessoas no domínio OutSystems para perceber como funcionava o processo de desenvolvimento e teste, de forma a entendermos quais eram os desafios e a experiência que tinham com a *BDDFramework*. Com a informação recolhida e juntamente com a análise que fizemos a outras ferramentas semelhantes, conhecidas para outras linguagens de programação, construímos um protótipo que utiliza a *BDDFramework* mas que agiliza todo o processo. Permite a descrição de cenários em ficheiros de texto (o que facilita a participação do negócio) e automatiza a criação de ecrãs de teste, permite o reaproveitamento de passos e ainda a detecção automática de parâmetros.

Realizámos testes com utilizadores para validar o nosso protótipo e este apresentou melhor performance em termos de velocidade, facilidade, usabilidade e correcção do que a *BDDFramework*.

A área de testes no domínio *Low Code* é ainda uma área com muito por explorar e os erros podem ser especialmente penalizadores dada a velocidade de desenvolvimento. Com este protótipo conseguimos demonstrar que é possível construir uma framework de automação de testes de BDD neste domínio, tirando proveito das características da linguagem OutSystems e estabelecendo um *standard* de teste.



# CONTENTS

|  |              |
|--|--------------|
| <b>List of Figures</b>   | <b>xix</b>   |
| <b>List of Tables</b>  | <b>xxiii</b> |
| <b>Listings</b>  | <b>xxv</b>   |
| <b>Acronyms</b>  | <b>xxvii</b> |
| <b>1 Introduction</b>  | <b>1</b>     |
| 1.1 Context and Description . . . . .                                      | 1            |
| 1.2 Motivation . . . . .   | 2            |
| 1.3 Objectives and Expected Results . . . . .                              | 3            |
| 1.4 Structure . . . . .  | 4            |
| <b>2 Background</b>  | <b>5</b>     |
| 2.1 Software Development Processes . . . . .                               | 5            |
| 2.2 Agile Development . . . . .  | 5            |
| 2.3 Test-Driven Development . . . . .                                      | 6            |
| 2.4 Acceptance Test-Driven Development . . . . .                           | 7            |
| 2.5 Behavior-Driven Development . . . . .                                  | 7            |
| 2.5.1 Advantages . . . . .   | 9            |
| 2.5.2 Disadvantages . . . . .  | 9            |
| 2.6 User Stories . . . . .   | 10           |
| 2.7 Gherkin Scenarios . . . . .  | 10           |
| 2.8 BDD vs ADTT vs TDD . . . . .   | 11           |
| 2.9 Software Testing . . . . .   | 13           |
| 2.10 Test design techniques . . . . .                                      | 13           |
| 2.11 Test Automation . . . . .   | 14           |
| 2.12 Regression Testing . . . . .  | 15           |
| 2.13 Exploratory Testing . . . . .   | 15           |
| 2.14 Continuous Integration, Continuous Delivery and Continuous Deployment | 16           |
| 2.15 Low Code Model-Driven Development . . . . .                           | 16           |
| 2.16 OutSystems Architecture . . . . .                                     | 17           |

|          |   |           |
|----------|---|-----------|
| 2.16.1   | Service Studio . . . . .                            | 17        |
| 2.16.2   | High-Level Architecture of Applications . . . . .   | 19        |
| 2.16.3   | Visual Language . . . . .                           | 21        |
| 2.16.4   | Testing Methods . . . . .                           | 22        |
| 2.17     | UI Testing . . . . .                                | 24        |
| 2.18     | BDDFramework . . . . .                              | 26        |
| 2.18.1   | Recommended practices . . . . .                     | 30        |
| 2.18.2   | Data-Driven Testing . . . . .                       | 31        |
| <b>3</b> | <b>Related Work</b>                                 | <b>33</b> |
| 3.1      | Low Code and BDD . . . . .                          | 33        |
| 3.2      | BDD principles and supporting tools . . . . .       | 34        |
| 3.2.1    | Discussion on the study by Wang and Solís . . . . . | 35        |
| 3.3      | Software evolution challenges . . . . .             | 37        |
| 3.4      | Automation frameworks . . . . .                     | 38        |
| 3.4.1    | How BDD frameworks work . . . . .                   | 38        |
| 3.4.2    | Well-known examples . . . . .                       | 40        |
| <b>4</b> | <b>Case study and initial considerations</b>        | <b>41</b> |
| 4.1      | Interviews . . . . .                                | 41        |
| 4.1.1    | Interview Questions . . . . .                       | 42        |
| 4.1.2    | Interview Results . . . . .                         | 42        |
| 4.2      | Analysis of BDD Automation Frameworks . . . . .     | 46        |
| 4.2.1    | Cucumber . . . . .                                  | 46        |
| 4.2.2    | SpecFlow . . . . .                                  | 52        |
| 4.2.3    | Framework Evaluation Proposal . . . . .             | 54        |
| 4.3      | Prototype Features Identification . . . . .         | 57        |
| 4.4      | Prototype Alternatives . . . . .                    | 58        |
| 4.5      | Decision making and strategy adopted . . . . .      | 60        |
| <b>5</b> | <b>Prototype Implementation</b>                     | <b>63</b> |
| 5.1      | Prototype Description . . . . .                     | 63        |
| 5.2      | Implementation Analysis . . . . .                   | 71        |
| 5.2.1    | Architecture Overview . . . . .                     | 71        |
| 5.2.2    | Development process . . . . .                       | 73        |
| 5.2.3    | Implementation Analysis . . . . .                   | 73        |
| <b>6</b> | <b>Validation and Results</b>                       | <b>85</b> |
| 6.1      | Planning . . . . .                                  | 86        |
| 6.1.1    | Goals . . . . .                                     | 86        |
| 6.1.2    | Participants . . . . .                              | 88        |
| 6.1.3    | Materials . . . . .                                 | 89        |



---

|            |  |            |
|------------|--|------------|
| 6.1.4      | Tasks  | 90         |
| 6.1.5      | Hypotheses, Parameters and Variables                 | 92         |
| 6.1.6      | Quasi-experiment Design                              | 95         |
| 6.1.7      | Quasi-experiment Procedure                           | 96         |
| 6.2        | Results and Analysis                                 | 99         |
| 6.3        | Discussion   | 106        |
| 6.4        | Comparison with other well-know BDD automation tools | 111        |
| <b>7</b>   | <b>Conclusions</b>                                   | <b>115</b> |
| 7.1        | Overview of the developed work                       | 115        |
| 7.2        | Contributions  | 117        |
| 7.3        | Future Work  | 117        |
|            | <b>Bibliography</b>                                  | <b>119</b> |
| <b>A</b>   | <b>Appendix 1 Interview Scripts</b>                  | <b>125</b> |
| <b>B</b>   | <b>Appendix 2 Experiment Introductory Scripts</b>    | <b>127</b> |
| <b>I</b>   | <b>Annex 1 NASA Task Load Index questionnaire</b>    | <b>133</b> |
| <b>II</b>  | <b>Annex 2 System Usability Scale questionnaire</b>  | <b>135</b> |
| <b>III</b> | <b>Annex 3 Task descriptions</b>                     | <b>137</b> |



## LIST OF FIGURES

|      |   |    |
|------|---|----|
| 2.1  | Outside-in development used in BDD[75]. . . . .   | 12 |
| 2.2  | OutSystems Platform Architecture [54] . . . . .   | 18 |
| 2.3  | <i>Service Studio</i> . . . . .   | 18 |
| 2.4  | The Interface tab . . . . .   | 19 |
| 2.5  | The Logic tab . . . . .   | 20 |
| 2.6  | The Data tab . . . . .  | 20 |
| 2.7  | BDD scenario specification to test the Equilateral triangle within <i>Service Studio</i> using the <i>BDDFramework</i> . . . . .  | 28 |
| 2.8  | All <i>BDDSteps</i> that constitute the (single) scenario were successfully performed, culminating in the positive outcome of the scenario. . . . .   | 28 |
| 2.9  | A <i>BDDStep</i> failed, resulting in a failing scenario. the failure report is displayed below the failing step. . . . .   | 29 |
| 2.10 | The expected result was "Equilateral"and the obtained is the value of variable kind previously assigned in the logic implementing the when clause . . . . .   | 29 |
| 3.1  | The BDD Characteristics support from seven BDD toolkits . . . . .   | 35 |
| 4.1  | The Cucumber Process, as described in <i>The Cucumber For Java Book</i> [63] . . . .  | 47 |
| 4.2  | The Java FizzBuzz method under test . . . . .   | 47 |
| 4.3  | Feature File with the plain text description of 2 Gherkin Scenarios, one for the <i>Fizz</i> case and the other for the <i>Buzz</i> case, as expected results . . . . .   | 48 |
| 4.4  | Test class generated from the feature file with the step definitions already implemented . . . . .  | 49 |
| 4.5  | In this two scenarios the values (“missing name error” and “missing email error”) in the <i>Then</i> clause are introduced manually. These scenarios could be compressed into a unique scenario using a scenario outline with the examples table containing the values to replace the variable. . . . . | 49 |
| 4.6  | Using the scenario outline we compressed the 2 scenarios into one unique scenario outline that will execute as many times as there are lines in the examples table. In this particular case 2 times, one for each value of the variable “Error”   | 50 |

|      |  |    |
|------|--|----|
| 4.7  | The immediate table below the scenario specification represents a data table. This data is all used within an execution of the scenario and represents the information retrieved when we search for some word which is in the scenario outline table below . . . . .                     | 50 |
| 4.8  | The first step of the three scenarios displayed is equal. . . . .  | 51 |
| 4.9  | In this case it is possible to group the equal steps in a background step that will be executed for all scenarios considered. . . . .  | 51 |
| 4.10 | Example of a more complete visual report, obtained with the execution information generated by Cucumber, integrated with Jenkins. . . . .  | 52 |
| 4.11 | Another example of a more complete visual report, obtained with the execution information generated by Cucumber, integrated with Jenkins . . . . .   | 52 |
| 4.12 | The C# FizzBuzz method under test . . . . .  | 53 |
| 4.13 | Feature File with the plain text description of 2 Gherkin Scenarios, one for Fizz and the other for Buzz, as expected results . . . . .  | 53 |
| 4.14 | Test class generated from the feature file with the step definitions already implemented . . . . .   | 54 |
| 4.15 | The most used frameworks compared with the <i>BDDFramework</i> in the new proposal of evaluation model. Attributes marked with a check mark are present in the frameworks. Attributes marked with a X mark are not present in the frameworks. . . . .                                    | 56 |
| 4.16 | Automated generation of the test logic and filling of the <i>BDDFramework</i> scenarios and steps process, from the external feature files by the prototype component. . . . .   | 59 |
| 4.17 | <i>eSpace</i> organization inside Service Studio. . . . .  | 59 |
| 4.18 | Schema that represents Hypothesis 2. . . . .   | 60 |
| 5.1  | Example of a feature file containing 2 features and 3 scenarios. The first feature contains two test scenarios and the second only one. The component parses the file and the highlighted words identify special keywords where new Features, Scenarios and Gherkin steps begin. . . . . | 64 |
| 5.2  | The following test module is obtained when we execute the BDD command using the previously presented feature file example. . . . .   | 65 |
| 5.3  | Each sentence in the Gherkin scenario is connected with a screen action containing its implementation. . . . .   | 65 |
| 5.4  | The Screen Action associated with a Gherkin step calls a Server Action (centralized) with the same name containing its implementation. There is <i>one Screen action per Gherkin Step</i> . . . . .  | 66 |

|      |  |    |
|------|--|----|
| 5.5  | The Server Actions hold the centralized implementations of the BDD steps. There is only one Server Action <i>per different Gherkin step</i> , avoiding action redundancy. Every Gherkin step “I have a valid card” will call the same Server Action, since their implementation is the same (equal steps). As we can see, there are 11 Gherkin steps in the example feature file (figure 5.1) but only 6 Server Actions in the generated eSpace, since some of the steps are the same. In the second and third scenarios, only one of the steps is new. . . . .  | 67 |
| 5.6  | The textual parameter is replaced by a variable in the sentence. It is ignored when we compare sentences to check for equality. Its value is passed as an input parameter for the server action that contains the corresponding step implementation. . . . .   | 67 |
| 5.7  | The Server Action (not yet implemented) receives the parameter as an input and it automatically infers its type, in this case a text containing the error. . .   | 68 |
| 5.8  | The <i>NotImplementedException</i> is defined for the generated Server Actions, when those are created. . . . .  | 69 |
| 5.9  | The <i>BDDFinalResult</i> block shows how many scenarios have failed during the execution. . . . .   | 70 |
| 5.10 | The result of publishing and displaying in the browser the scenario results from the <i>Withdraw cash from bank account</i> screen. The (single) scenario was not implemented and as expected the unique scenario fails right on the first step, where an exception is raised and the other steps are skipped. . . . .   | 70 |
| 5.11 | The Class Diagram representing the 3 classes which compose the Prototype.  | 72 |
| 5.12 | The <b>Generate BDD Scenarios</b> command execution flow. . . . .  | 74 |
| 6.1  | Scheme of the division of participants into two groups, experimental and control. . . . .  | 89 |
| 6.2  | Interface of the <i>Rectangle Area</i> OutSystems application, created to test Task 1.   | 91 |
| 6.3  | Scenario description to test the Equilateral Triangle demonstrated in the demo phase. . . . .  | 92 |
| 6.4  | Interface of the <i>Triangle Kind</i> OutSystems application, created to test Task 2.  | 92 |
| 6.5  | Hypotheses and sub-hypotheses formulated for the desired high level goals.   | 93 |
| 6.6  | Visual representation of the quasi-experiment procedures and estimated times for each phase of the quasi-experiment. The introductory script (1) and the presentation (2) were different for each approach, while tasks and questionnaires were the same. Phases 1 and 4 were done on paper, while phases 2 and 3 were done using a laptop computer. The estimated duration of the tasks was 60 minutes, although in most cases it was less, since participants were able to perform the tasks in time. Following phase 4, there was usually an informal conversation in a more relaxed context (but still important) and outside the quasi-experiment environment, where participants gave their informal feedback and opinions about the frameworks and testing processes presented. | 97 |

|       |  |     |
|-------|--|-----|
| 6.7   | Gherkin scenario given to the user as a test example for the application <i>Type Of Triangle</i> . . . . .   | 98  |
| 6.8   | Speed for the BDDFramework (blue) and Prototype (red) in task 1. . . . .   | 100 |
| 6.9   | Speed for the BDDFramework (blue) and Prototype (red) in task 2. . . . .   | 101 |
| 6.10  | Gaph comparing the mean score for each of the workload metric in both approaches. . . . .  | 102 |
| 6.11  | Boxplot for the Nasa TLX mean classifications for both tools. . . . .  | 102 |
| 6.12  | Graph representing the mean SUS responses for each item, for both frameworks. . . . .  | 105 |
| 6.13  | Boxplot for the SUS mean score for both tools. . . . .   | 106 |
| 6.14  | Mean speed for task 1, for participants with <i>BDDFramework</i> experience (orange) and for participants without <i>BDDFramework</i> experience (blue). . . . .                   | 107 |
| 6.15  | Mean speed for task 2, for participants with <i>BDDFramework</i> experience (orange) and for participants without <i>BDDFramework</i> experience (blue). . . . .                   | 108 |
| 6.16  | The most used BDD frameworks compared with the Prototype and the <i>BDDFramework</i> in the new proposal of evaluation model. Attributes marked with a X mark are missing. . . . . | 112 |
| A.1   | Script for the interviewees who only had contact with the BDDFramework . . . . .   | 125 |
| A.2   | Script for the interviewees who had contact with both the BDD process and the BDDFramework . . . . .   | 126 |
| B.1   | Introductory guide for Approach 1 ( <i>BDDFramework</i> ), page 1. . . . .   | 128 |
| B.2   | Introductory guide for Approach 1 ( <i>BDDFramework</i> ), page 2. . . . .   | 129 |
| B.3   | Introductory guide for Approach 2 ( <i>Prototype</i> ), page 1. . . . .  | 130 |
| B.4   | Introductory guide for Approach 2 ( <i>Prototype</i> ), page 2. . . . .  | 131 |
| I.1   | NASA-TLX questionnaire. . . . .  | 133 |
| II.1  | System Usability Scale (SUS) questionnaire. . . . .  | 135 |
| III.1 | First task. . . . .  | 137 |
| III.2 | Second task. . . . .   | 138 |

## LIST OF TABLES

|      |  |     |
|------|--|-----|
| 6.1  | Overview of the <i>independent</i> variables. . . . .  | 93  |
| 6.2  | Overview of the <i>dependent</i> variables. . . . .  | 93  |
| 6.3  | <i>Speed</i> descriptive statistics for the creation and the reuse tasks. . . . .                    | 100 |
| 6.4  | NASA TLX score interpretation. . . . .   | 101 |
| 6.5  | NASA TLX mean scores for both tools. . . . .   | 101 |
| 6.6  | NASA TLX descriptive statistics. . . . .   | 101 |
| 6.7  | Overview of the correctness results in the <i>BDDFramework</i> and in the <i>Prototype</i> . . . . . | 103 |
| 6.8  | Meaning of SUS score. . . . .  | 104 |
| 6.9  | Mean SUS answer for each question, for the <i>BDDFramework</i> testers. . . . .                      | 104 |
| 6.10 | Mean SUS answer for each question, for the <i>Prototype</i> testers. . . . .                         | 105 |
| 6.11 | SUS descriptive statistics. . . . .  | 106 |
| 6.12 | Welch t-test for task 1, concerning the speed variable. . . . .                                      | 108 |
| 6.13 | Welch t-test for task 2, concerning the speed variable. . . . .                                      | 109 |
| 6.14 | Welch t-test for the NASA-TLX, concerning the ease variable. . . . .                                 | 110 |
| 6.15 | Welch t-test for SUS mean score. . . . .   | 111 |





## LISTINGS

|     |   |    |
|-----|---|----|
| 2.1 | Gherkin Scenario Example . . . . .      | 10 |
| 2.2 | Equilateral Triangle Scenario . . . . . | 27 |
| 3.1 | Scenario Example . . . . .              | 38 |
| 3.2 | <i>Given</i> clause . . . . .           | 39 |
| 3.3 | <i>And</i> clause . . . . .             | 39 |
| 3.4 | <i>When</i> clause . . . . .            | 39 |
| 3.5 | <i>Then</i> clause . . . . .            | 40 |
| 3.6 | Step definition in SpecFlow . . . . .   | 40 |



## ACRONYMS

|       |                                     |
|-------|-------------------------------------|
| API   | Application Programming Interface.  |
| ATDD  | Acceptance Test-Driven Development. |
| BDD   | Behaviour-Driven Development.       |
| BDT   | Behavior-Driven Traceability.       |
| CD    | Continuous delivery.                |
| CI    | Continuous Integration.             |
| DDT   | Data-Driven Testing.                |
| DSL   | Domain Specific Language.           |
| IDE   | Integrated Development Environment. |
| LCMDD | Low Code Model-Driven Development.  |
| SUS   | System Usability Scale.             |
| TDD   | Test-Driven Development.            |
| UI    | User Interface.                     |



## INTRODUCTION

### 1.1 Context and Description

Over the past few years, Low Code platforms have gained increasing popularity in the software development market. These platforms allow users to create state-of-the-art web (and mobile) applications using a visual programming language while having little concern for the complex technologies that implement those applications and having facilitated design and integration, in a rapid development fashion. OutSystems is a good example of success among the Low Code platforms, according to recent reports[65].

According to some studies, about half of the software projects fail to deliver in some way. The Standish Group's annual *CHAOS Report* of 2015 found that 52% of the software projects were delivered late, ran over budget, or simply failed to deliver the requested features. According to the same source 19% of the software projects were cancelled entirely[73]. This corresponds to millions of euros lost in developing software that does not solve the business problems they were intended to solve.

In an attempt of addressing these problems, Behavior-Driven Development (BDD) has gained increasing popularity even though this a relatively recent practice. BDD was presented by Dan North in 2003 as an evolution of other existing Agile practices, like Test-Driven Development among others[45]. It is not a replacement for those methodologies but instead incorporates and enhances ideas from them and can be integrated and incorporated with other practices. It focuses on developing value-based software, promoting communication and understanding among all stakeholders (technical and business), by bringing the business into the development process as a way to ensure software success. In this Agile technique, the development process is thought in terms of behavior, not in terms of code implementation. Behaviors constitute a form of documentation, in a language that can be perceived by all, and they are the basis for the testing process. Later

on, the behavior descriptions are compared with the final product. These are defined by all stakeholders. On one hand, the business is more within the development process and can better perceive the technical limitations of the developers who in turn can better understand features demanded by the business and their value. This communication promises to reduce the number of errors that arise from poor communication which is one of the main causes of failed projects [16]. This and the fact that BDD promotes understanding, documentation, and enables test automation, leads us to believe that this methodology can be seen as a software development accelerator for OutSystems given its characteristics and considering the characteristics of Low Code languages. We considered this assumption during this dissertation.

In the OutSystems context, there is an open-source framework designed for test automation purposes, the *BDDFramework*, which allows the specification of executable Gherkin scenarios. However, despite its name, it was not built to support the BDD process, but only as a test automation tool where scenarios are described with the Gherkin syntax. One of the main goals of this dissertation will be to analyze the *BDDFramework* strengths and weaknesses and assess whether or not it can be improved to potentiate the practice of BDD in OutSystems.

## 1.2 Motivation

In Low Code, where development is fast, errors can have a significant impact on software development and lead to major regressions. Therefore it is important to write good software since the beginning, avoiding errors at all cost. The *BDDFramework* helps us building the software right and this is promoted by the OutSystems infrastructure, which raises the level of abstraction and facilitates integration. However, the *BDDFramework* does not help us building the right software (i.e. the software user wants). Behavior-Driven Development can be a valuable help in relation to this. BDD is about promoting communication between stakeholders through conversations around the expected behaviors of the system. From these conversations result features that are further broken down into scenarios that constitute a form of acceptance criteria of the system (a form of living-documentation and the basis of automation). We want to enable this from a technological point of view, with a tool that can assist in this process.

Communication is key in software projects. One of the main causes pointed to the failure of software projects is the misinterpretation that developers make of the requirements, which is a result of poor communication between stakeholders. Developers often misunderstand the requirements that come from the business who in turn do not know the capabilities of the developers, nor the technical challenges associated with the requirements they are asking for. This the problem that BDD tries to overcome. Simply with conversations between all the stakeholders, in a language spoken by everyone.

Besides that, due to the fact that Low Code raises the level of abstraction, we experienced that conventional unit-testing is not so prevalent as in other high code languages

[12]. We need something of a higher level of abstraction to perform and standardize tests, like the BDD behavioral scenarios, which are great for test automation and documentation as it is already done with success in other languages and with the aid of powerful support tools.

One of our biggest motivations and one of the reasons why we consider to be so important the test automation that is allowed by the BDD process is that it releases the software testers from the tedious task of repeating the same assignment over and over again, freeing them to perform other types of tasks and testing that can not be automated, opening space for creativity, being these some of the factors concluded in said to improve software testers motivation at work [64].

The world of software development is still coming to terms with the Low Code reality and further research about the best practices to develop software according to its characteristics and speed of development is required. The literature addressing to this topic is also scarce and hence the importance of this investigation. We need to fulfill this gap because other methodologies, although applicable may not be well optimized to Low Code and sometimes require adaptation.

We will study the BDD process in this particular environment that is the Low Code, and try to find the best way to do it, enhancing the practice from a technological point of view.

### 1.3 Objectives and Expected Results

By the end of this dissertation, we expected to have clearly defined requirements for a Low Code framework that supports BDD as a development process in OutSystems and that can also be used for test automation purposes. Those should work for OutSystems, without the need to leave this domain and taking advantage of the platform characteristics to technologically empower it. The requirements specified, based on the principles of BDD and Low Code, should culminate in the implementation of a prototype. For that, it will be necessary to make a properly justified choice: to use and extend the *BDDFramework* to support the BDD process in OutSystems like other tools that exist for other languages, to extend one of those existing test automation tools that use the Gherkin syntax and integrate it with the OutSystems language or to create a new tool completely from scratch. During the accomplishment of this work, it is expected the constant contribution of people in the field to obtain feedback and for understanding and analyze their development process, to better understand the needs of developers and the OutSystems platform dynamics. This is a very important aspect and this feedback will be obtained on a day to day basis whether in the form of interviews or informal conversation.

In addition to the description of the whole process, at the end of the development phase there will also be a phase for the realization of tests with real users. They will use the developed prototype with the objective of comparing the test approaches carried out in the past with the approach using the prototype. All the results, as well as the detailed

description of the whole process and the choices made, will be described in detail in this report.

We can summarize the objective of this dissertation as:

Realizing the characteristics of Behavior-Driven Development and Low Code technologies, we want to develop a test automation framework in the OutSystems domain that enables the BDD process technologically, having as a starting point the existing *BDDFramework*.

It can unfold into 2 main research questions:

- **RQ1:** What are the main strengths and weaknesses of the *BDDFramework* and how could it be improved to support the BDD process?
- **RQ2:** How can we build a BDD testing automation framework for the OutSystems language?

## 1.4 Structure

The remainder of this dissertation report is organized as follows:

- Chapter 2 - **Background**: this section addresses the main research concepts, being the main topics the OutSystems Platform, BDD, Automated testing and Automation frameworks;
- Chapter 3 - **Related Work**: in this section we will present some related work on BDD and Low Code, that relate to the context of this dissertation;
- Chapter 4 - **Case study and initial considerations**: this section describes the interviews conducted, to better understand what are the problems on the field and describes the options for facing the problem that were left on the table and the motivation for choosing one of them above the others.
- Chapter 5 - **Prototype Implementation**: this section describes the implementation of the prototype, considering all the decisions taken, the procedures and algorithms used.
- Chapter 6 - **Validation and Results**: presents the results obtained for this work, both by the execution of the tool as well as from usability tests performed. Contains a detailed description of the whole experimental process.
- Chapter 7 - **Conclusions**: finally a quick overview of the work produced during the dissertation with some future considerations regarding possible improvements and features.



## BACKGROUND

### 2.1 Software Development Processes

A software development process is a structured set of activities required to develop a software product. There are multiple types of software processes, but all involve the following phases:

- **Specification:** defining what the system should do;
- **Design:** defining the organization of the system;
- **Implementation:** implementing the system;
- **Verification and Validation:** checking the correctness of the system and that it does what the customer wants;
- **Evolution:** changing the system in response to changing customer needs.

Nowadays, agile methodologies are among the most common approaches used in practice to conceive software products [74]. In this kind of approach, the planning phase is incremental and functionalities are developed in iterative development cycles. It is easier to change the process to reflect upcoming customer requirement changes in opposition to traditional Plan-driven approaches, in which all the process activities are planned, and progress is measured against a plan [71]. The customers are more involved in the process and there is less documentation involved than in plan-driven approaches.

### 2.2 Agile Development

Agile methods represent iterative and incremental development processes. Their main goal is to help teams in **evolving environments**, maintaining focus on **fast software**

**releases** with business value and **dealing with constantly changing requirements**. **Customers are involved in the process** and participate in an active way for quick feedback and reporting requirements. Typically, agile methods work through development in **sprints**<sup>1</sup>. The focus consists in reducing the overall risks associated with long-term planning and changing requirements, building software that does not serve the customer [34]. Scrum, eXtreme Programming (XP), Pair Programming are some examples of agile methodologies. These methods assist teams in responding to the unpredictability of constructing software and help follow the agile manifesto which is a set of principles that is based on **continuous improvement, flexibility, input of the team and delivery** of results with high quality. Individuals and interactions are prioritized over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation and finally responding to change over following a plan [36]. The manifesto argued that we should pay more attention to some aspects, but of course not neglecting others that were previously considered more important.

### 2.3 Test-Driven Development

In agile development, one of the most well-known practices to develop the software code is Test-Driven Development (TDD). In this incremental software development process, **first the developer writes the test for a given new piece of functionality** (a unit). This test initially fails, as the functionality is not yet implemented. **Then, the developer writes the code that implements the failing functionality**, just enough code to make the test pass. **Finally, the developer refactors** the new code [7]. In Test-Driven Development (TDD) automated unit tests are written before the code itself is made. Running these tests gives developers a fast confirmation of whether the code behaves as it should or not [30].

In some contexts TDD can be difficult to apply in practice and it does not provide a standardized structure and guidelines on how testing should be developed, but instead a wide range of recommended practices. This can make testing difficult to understand for technical workers who are not participating in the process (and even more for business people). These business stakeholders, such as customers and business analysts, can not easily contribute to assessing whether the product meets the demanded requirements which might lead to a frequent misunderstanding about how the software should behave, leading to delays as it can waste a lot of time in the next sprints correcting things from the previous ones that must be corrected before advancing [43]. TDD is often associated with Unit Testing, so the level of abstraction to which TDD refers is usually very low.

**Sprints** are periods of time during which the defined work and tasks must be completed by the development teams. In an initial phase the tasks to be implemented during the sprint are decided and analysed by the teams and planning is done for the time available (Refinement and Planning phases). In the end of the it, the results are analysed and

---

<sup>1</sup>Incremental, iterative work sequences with limited duration in which the tasks to be developed are previously defined and planned by the development teams

the new features are integrated and released after refactoring [62, 66].

## 2.4 Acceptance Test-Driven Development

An acceptance test is a **description of the expected behavior of a software product**, usually expressed as a **scenario** for automation and documentation purposes. **It should be possible to execute such specifications with automation frameworks**. Acceptance testing is a way of **functional specification** and **formal expression** of business requirements in Agile[1, 2].

Acceptance Test-Driven Development (ATDD) is a **technique used to bring customers into the test design process** before coding has even begun. Customers, testers, and developers define the automated acceptance criteria in a **collaborative fashion**. ATDD is related to TDD in the sense that it highlights writing acceptance tests before developers begin coding. However, the main difference is the emphasis on the collaborative participation of developers, testers and business people, commonly known as the *Three Amigos*[32].

ATDD is a way to ensure that all stakeholders understand what needs to be done and implemented. Tests are specified in **business domain terms** and each of them tests features with measurable business value (software that matters)[19].

So, ATDD is a process in which high-level acceptance tests, designed by all the stakeholders (including the customer), are automated to initially fail and then developed just to create enough production code to make them pass (following a TDD fashion). This constitutes a “contract” between customers and developers as a feature is considered adequate if it passes the acceptance tests. Despite all this, ATDD requires a lot of discipline and communication to make it worth and this communication should go from the Product Owners to the developers and the other way around, in both directions. However, in the end, we get an easy to read **living documentation** reflecting how the system behaves.

## 2.5 Behavior-Driven Development

There are many reasons for software projects to be unsuccessful: delays, poorly calculated costs, non-compliant end-products, among others. One of the most common problems in software projects arises from the **lack of communication between the development teams and the business people**[16]. Due to this poor communication, often developers do not quite understand what needs to be done, and the business people misunderstand what are the capacities of the developers and the implementation difficulties of the software they ask for[69].

Behavior-Driven Development (BDD), also known as *Specification-by-example* was created by Dan North in 2003 as an evolution of Test-Driven Development to deal with these communication problems mentioned above and to help developers know **where to start, what to test, how to name their tests and why tests failed**[46].

According to North:

*“BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.”*

BDD is about having conversations to help teams avoid misinterpreting requirements, while promoting a shared knowledge between team members as early as possible in a user story lifecycle[33]. It is about describing an application by its behavior, from the perspective of its stakeholders, in this case, the *Three Amigos* representing the developer, the tester and the business. The main difference between BDD and other Agile approaches is the importance it gives to business value, by including the business people in the conversations about the development process in order to build software that matters, which is the software the customer wants and avoiding misunderstandings with the development team – writing proper code from the beginning[45].

North sees BDD as a **centered community** and not as a bounded one, as BDD presents a set of principles and values but with undefined borders. In bounded communities it is much easier to define whether we are doing a given practice, based on a set of principles, but in a centered one is not that easy[12, 61]. **BDD has evolved out of established agile practices** (like TDD and ATDD) and is designed to guide and enable agile software delivery to teams new to this approach.

When adopting BDD it is important to focus on solving the problems of delivering the software customers want and not only in testing automation techniques. Sometimes BDD is seen, incorrectly, as a way of generating automated auto-descriptive tests through BDD frameworks like Cucumber[18] or SpecFlow[72]. Although this automation is an important part of the BDD process, the main focus should always be, first on having the conversations between the *Three Amigos* and only then in automation.

One of the most important aspects of BDD is the definition of a **Ubiquitous Language** that allows communication between the different stakeholders in domain terms perceived by all. This is crucial since the success of the practice relies on good communication without misinterpretations, to accelerate the software process and make it less error-prone[15].

The Ubiquitous Language definition constitutes a Domain Specific Language (DSL) which is a computer language that allows to provide a solution for a particular class of problems. Among other things, it makes easier to express domain terms. In BDD this is usually done in **Plain English Text User Stories** and **Gherkin Scenarios**[69].

BDD has a major goal of determining the behavior that is right for the business before code gets written. However, the resulting Gherkin scenarios that are produced are convenient for **test automation** and **documentation**. This has led to the popularization of frameworks like Cucumber, JBehave or SpecFlow. These Gherkin frameworks have become also popular **outside the context of BDD**, for the purpose of test automation alone. This is due to the fact that some teams see benefits in using Gherkin only for automation:

the self-descriptive nature of test specifications (documentation), the common understanding through an ubiquitous language, the reuse of step implementations and having a standard to structure tests. The nature of the Gherkin syntax by itself is very appealing, just in the sense that it provides a standard for everyone to follow when doing certain levels of test-automation, while assuring that the **tests provide a clear documentation of themselves** (living documentation).

### 2.5.1 Advantages

According to Smart in his book *BDD In Action* (commended by North), the main advantages of the BDD process are[69]:

- **Reduced waste and costs:** since there is an increased effort in finding business valued features and devalue those which do not represent business value, there is a waste reduction and consequent cost savings, producing software the customer wants since the beginning;
- **Changes are easier to accommodate:** since living documentation is generated through executable specifications in a common language to all stakeholders, the code becomes easier to understand, perceive and maintain. The kind of documentation makes it easier to understand what each feature represents, the meaning of the tests and why they fail;
- **Faster releases:** with test automation it is no longer necessary to spend much time running tests manually so more time can be invested in exploratory testing and other kinds of testing which require more skill and attention from the developer. Releases may come out faster once the testing process is simplified;

### 2.5.2 Disadvantages

According to Smart, the fact that it is a relatively recent practice that has been gaining increasing popularity only in recent years makes the literature sparse on this subject. Also, the fact that it is a second-generation method makes people often confuse it other technologies like TDD and ATDD that gave rise to it. But, above all, the fact that BDD is still widely **seen only as a form of test automation** and not for the importance it gives to business value, leads to misuse and consequent failure of the method in some situations[69]. Among the many reasons, the ones that stand out the most are:

- BDD **requires high business engagement** in order to be efficient and it can be difficult to implement in large companies or companies that practice“off-shore testing”, because teams work in separate spaces, making it more difficult to communicate;
- BDD is **highly focused on functional requirements** not offering many solutions for non-functional requirements;

- **The use of BDD as a bounded set** (“Do this and that and you will succeed”)[61]. BDD is a practice that derives from other agile practices and does not follow a set of rules or principles and instead follows some baselines, but the process can and should be tailored to each context and reality;
- Users who wish to use BDD, usually search for available tools and frameworks that support the process which are often an illusion that a complete and reasoned BDD process will automatically be followed;
- **Scalability can be difficult to achieve** in large organizations where communication between different teams is not easy because of the physical or even geographical separation of its members.

## 2.6 User Stories

User stories are feature descriptions told from the perspective of the customer. They typically follow the structure:

**As a** < Role >, **I want** < Goal > **So that** < Motivation >.

User stories are **written for discussion**. After a feature is identified, along with its users (with roles) we can describe scenarios from them that implement and constitute concrete examples of those features. Scenarios are expressed in a natural like language, such as Gherkin. User Stories are the basis of the discussion that takes place at the beginning of each development phase (Sprint) and can be written by any stakeholder (from a user perspective) [4].

## 2.7 Gherkin Scenarios

Gherkin is a plain-textual language with the **Given, When, Then** structure which represents the **initial state**, the **action** and the expected **result** of the action, respectively. Gherkin is designed to be easy to learn by non-technical stakeholders and it is used to express software in behavior terms to make features easier to understand.

The **Given** clause represents the initial context of the scenario - the various states that we should verify before we perform the action.

The **When** clause represents a specific action that must happen to trigger the behavior represented by the Then clause.

The **Then** clause describes the expected outcomes of conducting the action/event in the system. All of these should be written in a simple and clear way for better understanding of the behavioral scenarios [24].

Example:

Listing 2.1: Gherkin Scenario Example

```
1 Scenario - Wrong credit card number
```

```

2 GIVEN The user inserts a valid credit card in the ATM machine
3 AND The user inserts a wrong number
4 WHEN The user confirms the number by pressing the green button
5 THEN An error message is displayed

```

In addition to these main keywords, some descriptions using Gherkin and especially some software also allow the use of other keywords:

- **And:** is used to add conditions to our steps. Refers to the previous indicated keyword (Given, When or Then);
- **But:** Like the *And* keyword, it also refers to the previous step, but this keyword is used to add negative type comments. It is good to use when our step describes conditions which are not expected, for example when we are expecting some text or when an element should not be present on the page.

The close relation between specification and acceptance testing allows BDD scenarios to become the living documentation of the system. In BDD, examples of behavior become test code and ultimately documentation, with scenarios becoming acceptance tests and eventually regressions tests.

Once the conversations between stakeholders are happening, they can be captured using the Gherkin syntax. Then, we can use the captured examples along with automation tools to develop automated tests, in what is known as an **outside-in approach**. The main automation tools that support BDD (such as Cucumber and SpecFlow) work like this:

1. The framework reads a specification file with the scenario descriptions;
2. It translates the formal parts of the scenario's ubiquitous language (the Gherkin keywords - given, when, then) breaking each scenario into meaningful individual clauses (usually called *steps*);
3. Each clause is then transformed into some method for testing. The generated methods (step definitions) should then be implemented by the developers;
4. The framework allows executing the test, reporting the results at the end, usually with information about the scenarios that passed and those that failed.

## 2.8 BDD vs ADTT vs TDD

BDD is a methodology that originated from TDD and ATDD, among other practices. TDD is focused towards building correct software but not on building the software that the user wants. On the other hand, ATDD focuses on helping developers build the right software by promoting the collaborative construction of high-level acceptance tests (integration and acceptance testing), unlike TDD that is used at a lower level of abstraction

(unit level mostly). Both practices complement each other, acting at different levels of abstraction. It is sometimes said that unit tests ensure you build the thing right, whereas acceptance tests ensure you build the right thing. These are also the informal definitions of **verification** and **validation**, respectively.

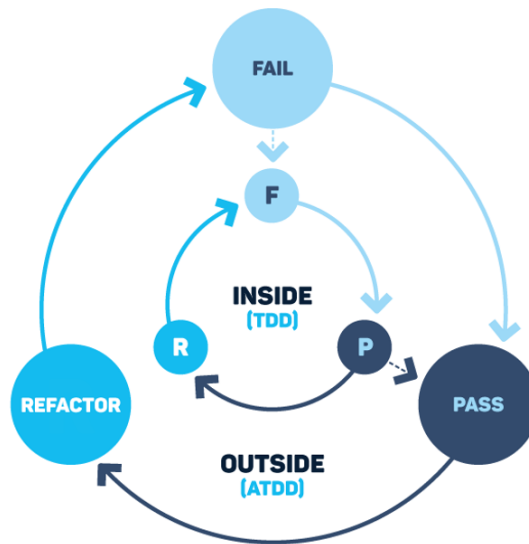


Figure 2.1: Outside-in development used in BDD[75].

BDD combines these two practices, in an approach called **outside-in**. The developer connects a Gherkin scenario (built from conversations between stakeholders) to failing automation code (not yet implemented). This phase of the process is similar to what happens in ATDD. The main difference with the ATDD process comes when we go inside the circle and apply the TDD process, creating unit tests for each software unit needed to make the outer circle pass. The value of the outside in development approach is that the developer is guided towards delivering the right software as their focus is to make the example pass, thus comprising understand the verification and validation phases. ATDD and TDD are methodologies used within BDD, however, these are more developer-sided rather than client-sided[75].

BDD leverages the ATDD approach around conversations, collaboration, and automation to ensure that a team delivers what the business wants [19]. The conversation should always be over automation. Usually in Behavior-Driven Development we define behavioral scenarios expressed in plain text, in a language perceived by all stakeholders. To express the scenarios developers combine plain English (or other native language supported) with the ubiquitous language structured with Gherkin Syntax with Domain-Driven Design (DDD).

BDD is focused on the customer and ATDD is more focused on development, although it usually also has users participating in the acceptance criteria definition.



## 2.9 Software Testing

Testing can be performed at different abstraction levels. A software system goes through several stages of testing before it is available for deployment [8]:

1. **Unit level:** individual program pieces (units) are tested in isolation;
2. **Integration level:** integration is the activity of aggregating software units to create a larger and consolidated component. Integration testing aims at testing the larger components to guarantee that the pieces that were tested in isolation can now work together as functional piece;
3. **System level:** this level includes a wide variety of tests for the system. Verifies that the system works as a whole and that the software is well-built. It is the last test of software before it is passed to the customer;
4. **Acceptance level:** customers perform their own series of tests, based on their expectations for the system. The objective of acceptance testing is to measure the quality of the product, checking if it validates the intended behavior the customer wanted.

The first three levels of testing are performed by several different technical stakeholders in the development organization, whereas acceptance testing can be performed by customers or other non-technical stakeholders.

## 2.10 Test design techniques

The two main concepts relating testing and the availability/accessibility of the code under test are **White-box** and **Black-box** Testing [35]

- **White-box testing:** we have access to the source code of the system and the test case approach is based on software implementation. The goal of selecting such test cases is to cause the execution of specific isolated parts of the software. The expected results are evaluated under a set of code coverage criteria. Usually applied at unit, integration and system levels.
- **Black-box testing:** the internal details of a program are not known (code is not available), and it is thus treated as a black box. The selection of test cases is based on the requirement or design specification of the software under test (**functional testing**). Functional testing relies on the external behavior of the software. Usually applied at integration, system and acceptance levels.

Black-box testing is very common in Low Code Platforms. White box testing is often difficult or even impossible to apply to these since they do not often offer the mechanisms or flexibility to do so.

## 2.11 Test Automation

It is difficult and sometimes not practical to test software manually since tests are vulnerable to inaccurate results and manual tests are also **slow** and **difficult to maintain** by developers. In some cases a manual approach might not be the most effective in finding certain bugs in the software. Test automation has the objective of making the software as error-free as possible in order to be delivered in the market. Another goal of test automation is to **reduce the work of developers**, since in very large projects it is impractical to do the entire testing process manually. Automation may include:

- Automatic **generation of test cases**;
- **Selection of inputs and computation of outputs** and evaluation;
- **Automatic execution of tests** and regression testing.

It allows that **releases are delivered much faster** and that **less staff is assigned to manual testing** [20]. Often test automation is combined with manual testing for tests that cannot be automated and this is critical when we are in continuous delivery scenarios.

In test automation, we have four basic components: **testers**, **test automation frameworks**, **test cases**, and the **system** under test. Quality engineers interact with the test automation tools and develop test cases which are then executed using the chosen test automation tool. The tests exercise the system and the framework provides test reports for users to interpret. Even though the introduction of test automation often increases cost for creating tests, the cost of re-running them decreases[28].

In BDD, not all scenarios need automation: some may be difficult to automate cost-effectively and should be left for manual testing or on the other hand might be just experimental scenarios. Others, may not have much interest to business and might be better off implemented as unit or integration tests. But usually, given the nature and structure of the Gherkin syntax, BDD scenarios are a **great source for test automation**, so it is necessary to emphasize the importance of writing scenarios carefully, so they can bring value to the table.

The main positive aspects of test automation using BDD are:

1. **Provides the ability to perform tests that are very difficult to execute manually thus increasing test coverage**: test automation frameworks can look inside an application and see memory contents, data tables, file contents, and internal program states to determine if the product is behaving as expected, easily executing thousands of different complex test cases during every test run and providing coverage levels that are impossible to achieve with manual testing. These tools already have integrated mechanisms that can simulate many virtual users interacting with the network, software and web applications, something that was also extremely difficult to do with manual testing.

2. **Testers can understand more easily what the automated tests are testing:** they helped designing them, through the collaboration writing the acceptance scenarios. It also **frees testers to perform other types of testing** like exploratory testing or more complex experimental tests.
3. **Faster releases:** New releases can be delivered faster and new versions are less likely to introduce regressions as testers can focus their time in other types of testing if the automated testing process is good. This is very important to continuous integration and delivery.
4. **Scenarios are living documentation** - It is easier to monitor results and keep track of the tests. Most of the tools that perform automated testing support monitoring and management of test suites offer test reporting mechanisms and in BDD the tests are self explanatory of themselves given their nature.
5. **Improves consistency of test results:** Even the most experienced testers will make mistakes during monotonous manual testing. Automated tests perform the same steps precisely every time they are executed and never forget to record detailed results.

## 2.12 Regression Testing

Whenever developers change their software, either by adding new features or modifying existing ones, there is always the risk of introducing errors in their programs. Even a small tweak can have unexpected consequences in the operation of an application and introduce bugs in features that have not been modified. Regression Testing is a Software Testing method in which **test cases are re-executed when a change is introduced**, in order to **check whether the previous functionalities are working fine** after we introduce new changes to software. It is done to make sure that the new code does not have side effects on the existing functionalities. Repeating a suite of tests each time an update is made is usually a time-consuming task in medium to large size projects, so a test automation tool is typically required [40, 60]. Every release usually adds more regression testing to the next release. This means that a software development organization has to keep increasing its testing capability every release and adding more people is not a scalable solution, so regression testing mechanisms and supporting tools are needed.

## 2.13 Exploratory Testing

In Agile approaches, because of the frequent releases, test automation becomes very important as developers need to get quick feedback on the status of the application. The automated executions work as regression tests to ensure that with each release the software has not regressed. Exploratory Testing focuses on areas that the existing test

automation frameworks might not cover. It is usually performed by testers, who are more experienced in the testing phase, unlike the regression tests that are normally automated by developers. Also due to the short development periods, testing inherently becomes risk based, and exploratory testing can focus on high risk areas to find potential problems [3, 37].

## 2.14 Continuous Integration, Continuous Delivery and Continuous Deployment

**Continuous integration** involves automatically building and testing a project whenever a code change is made. Continuous integration (CI) alerts developers to regressions and build problems as early as possible. CI relies on a well-designed set of automated tests in order to be efficient[39].

**Continuous delivery** is an extension of continuous integration. Whenever a developer inserts new code into the source, a server compiles the new candidate version to be released. If this released candidate passes a series of automated regression tests (unit tests, automated acceptance tests, quality tests), it can go into production as soon the business stakeholders want. [26].

**Continuous deployment** is similar to continuous delivery, but without the manual approval stage. Any candidate release that passes the automated quality checks will automatically be deployed into production[49, 69].

## 2.15 Low Code Model-Driven Development

Low Code platforms are software development tools that allow users to develop and deploy software in a fast and efficient way, abstracting many code concepts and making possible for the developer to accelerate the development process with already pre-designed and pre-integrated models in a Low Code Model Driven Development (LCMDD) fashion. The OutSystems Platform is a good example of a Low Code Platform with its own unique language [49].

Outside the context of the low code, Unit Testing is the basis of Software Testing especially in approaches that use TDD. In Low Code Languages the code is not sometimes available to be directly or conveniently tested unit by unit, separately. We are talking about visual models with higher levels of abstraction, where the details of implementation are not in sight of the user that news approaches or specific software development processes[12]. Development in these languages is often similar to what is practiced in other approaches, with agile practices predominating, in accelerated development cycles in order to put applications into production as soon as possible, exposing them to their real users, in order to be iterated and continuously improved. New development

approaches might emerge that can explore the LCMDD characteristics and the speed of development.

## 2.16 OutSystems Architecture

The OutSystems platform allows to create state-of-the-art web and mobile applications, through a visual programming language, while having little concern for the technologies that implement those applications. This Low Code service supports development at a higher abstraction level, simplifying the daily life of IT professionals with a strong focus on performance, scalability and availability.

The Platform is composed by two main components: the *Development Environment* (composed by Service Studio and Integration Studio) that interact with the other main component through web services, the *Platform Server*[47]:

- **Service Studio:** Service Studio is a computer environment where we can build web and mobile OutSystems applications using visual models in a drag-and-drop fashion;
- **Integration Studio:** In the Integration Studio, developers can create components to integrate existing third-party systems, micro-services and databases, or to extend the platform with their own sources of code.
- **The Platform Server:** contains all the components needed to generate, optimize, compile, and deploy native C# or Java web applications. For mobile applications, it also builds, packages and deploys native applications for Android and iOS.

We can see the OutSystems Platform architecture in Figure 2.2.

### 2.16.1 Service Studio

*Service Studio* is the Integrated Development Environment (IDE) used to develop web and mobile applications with the OutSystems language. With its visual domain specific language, the users can define business processes, the interface of the application they want to implement, the logic behind it, and the data layer of the application. These three main areas are presented as tabs in the *Service Studio's* Interface.

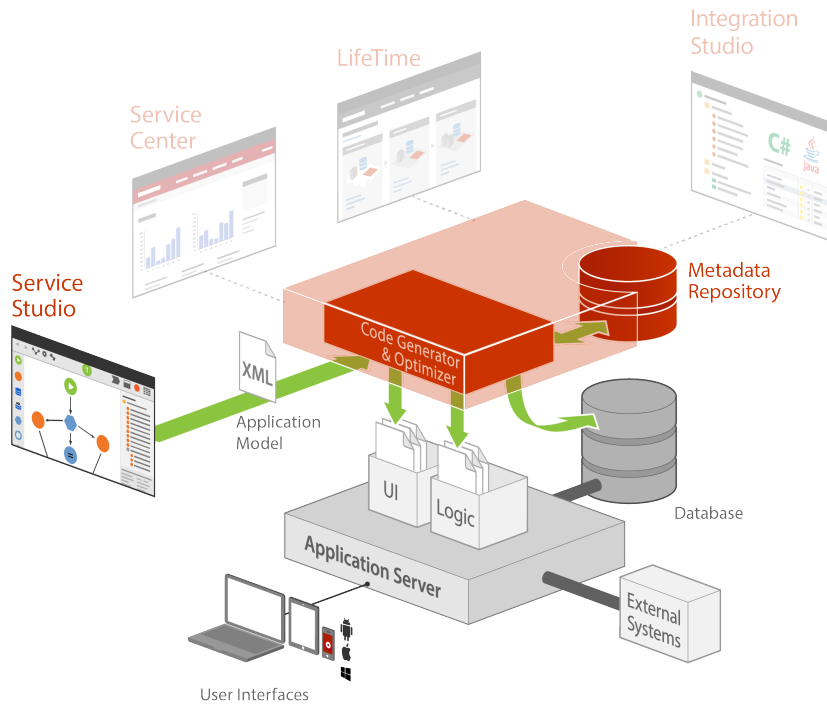


Figure 2.2: OutSystems Platform Architecture [54]

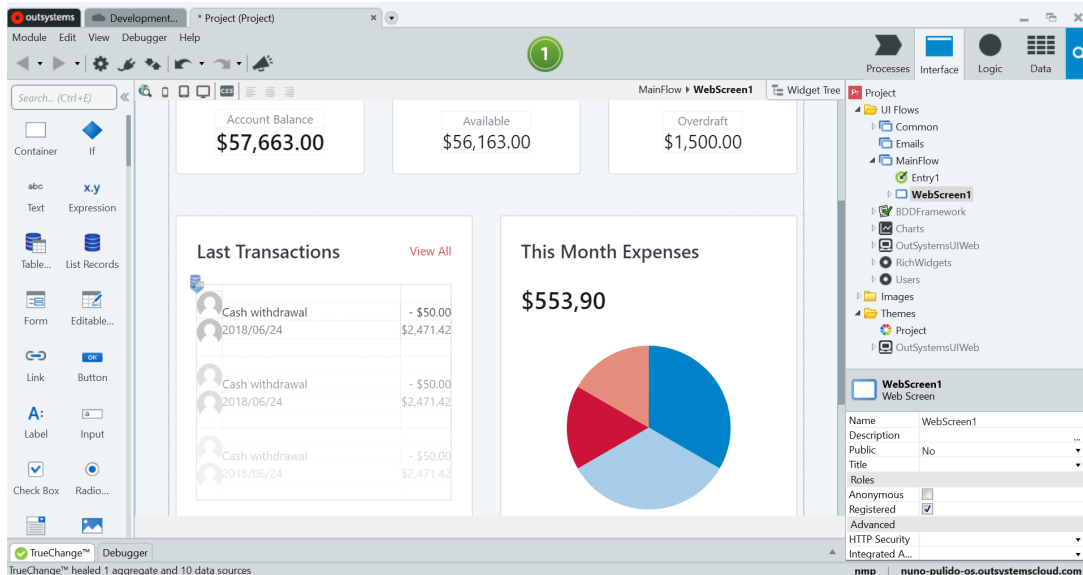


Figure 2.3: Service Studio

The **Interface** tab is used to define the UI of applications. In order to do this, developers can rely on the platform **widgets** and they can also define his own **reusable interface blocks**. Besides using the visual representation of the interface, developers can also have an overview of the interface tree, and use it to navigate to a certain component, where they can see and modify its properties. In this tab we can also manage the UI flows of the

application to organize our screens into groups and have an overview of interfaces and interactions. All screens in the same flow share common settings.

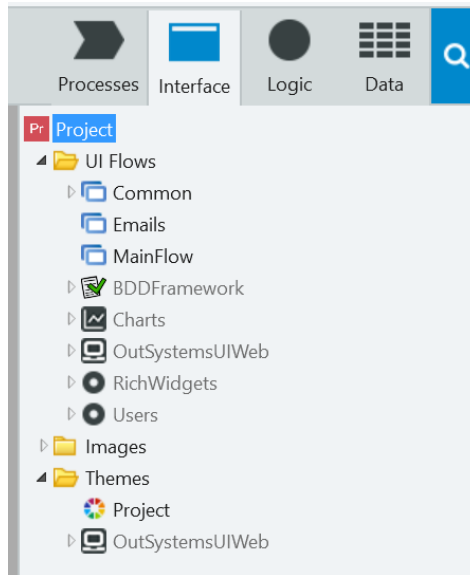


Figure 2.4: The Interface tab

The **Logic tab** allows the definition of Actions. These actions are divided into **Client Actions** (executable on mobile devices) and **Server Actions** (which run on the Server). The developer has a series of operators he can use to define his actions, which can be used to execute a broad range of tasks. Server Actions are centralized actions that contain the visual logic that **can be used anywhere** in applications and are not only associated with a screen like the logic in the interface tab. This tab can contain wrappers that add additional logic to the creation of new server records, as well as actions to handle the synchronization, or any other actions to be executed either on the client device or on the server.

The **Data tab** allows the definition of the data model. It allows the creation of server entities and the local entities stored in mobile devices. These entities can be static or dynamic and include a set of predefined logic operations that can be used to create, add, update and remove records of them, among other things.

The platform also enables the modeling of UIs, Business Processes, Business Logic, Databases, Integration Components, SOAP and REST Web Services, Security Rules, and Scheduling activities, among other features.

## 2.16.2 High-Level Architecture of Applications

In OutSystems a **Module** is either an **eSpace** or an **Extension**.

An application consists of multiple modules, and a **Solution** consists of multiple applications.

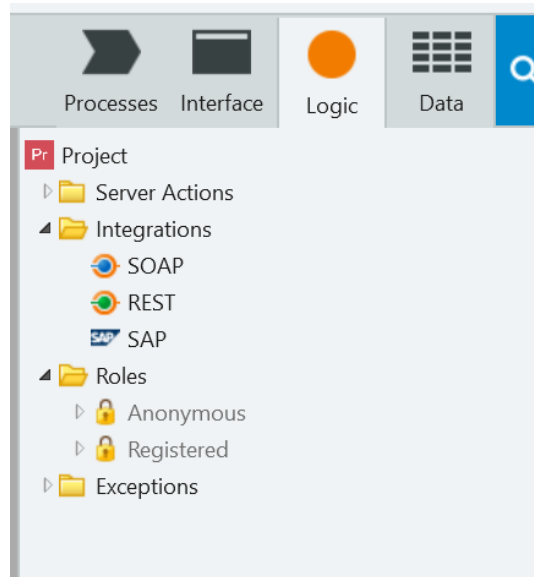


Figure 2.5: The Logic tab

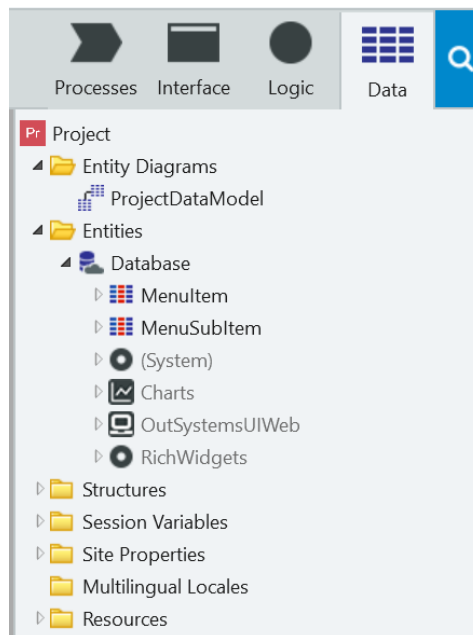


Figure 2.6: The Data tab



An **Extension** is some code written in .NET, JAVA, among other languages. These extensions can be used to extend the functionality of applications.

An **eSpace** is a module where we develop an application, in which we can create screens, logic, manage data, expose web services, among other things.

An application is a **collection of modules** and a **Solution** is basically everything we have inside the environment with all the eSpaces we have [49].

### 2.16.3 Visual Language

OutSystems' visual language for web development allows users to develop the UI of applications using web screens and web blocks, managed on the Interface tab. Actions are also an important part of applications. They contain the functional logic and can be associated with screens and screen elements (preparation actions and screen actions) and they are managed in the interface tab. **Server actions** are another very important type of actions. . These can be reused inside and outside applications and are on the server side. Server actions are not associated with screens but instead are associated with the application itself, and can be exported. They are in the logic tab, like the folder containing the System exceptions, which can be of various types. Customized user exceptions can also be added manually. In the context of this investigation we will focus on the 2 tabs mentioned above (Interface and Logic) and we will now explore a little more of what they have to offer that can be used in the context of this dissertation:

**Web Screens:** User interface pages that end-users interact with. They can contain all kinds of elements including **blocks, screen actions, parameters and variables**.

**Web Blocks:** Reusable screen parts that can implement its own logic. These can be used within screens and within other blocks and contain their own variables (local, input and output) and their main advantage is that they are **reusable**.

**Screen Actions:** Actions that include logic regarding user interactions with screens, such as clicking on a button or a link. They **run in the client side** (UI, browser).

**Server Actions:** In OutSystems we can create Server Actions to encapsulate the logic that implements the business rules of our applications and use them in other actions, such as Preparation actions, Screen Actions, other Server Actions or even other applications.

**Input/Output Parameters:** Input/Output parameters are associated with actions (inputs can be also associated with screens) and for those actions to be executed they must receive values for the inputs that are either computed or directly assigned from the user and return their values in output parameters.

**Local Variables:** Local variable to a screen, action or block.

To implement the logic of actions, we have at our disposal a vast set of **web logic tools** among which we highlight the following:

**Assign Statement:** The Assign Statement is used to assign values to variables. The

Properties Panel shows assignments in variable-value pairs, forming the assignment statements. The value in the statement can be other variable, literal or a computed value.

**Raise Exception:** throws an existing systems exception or we can create a new one. If we create a new exception we can customize the message displayed to the end-user. This is an element that ends the action flow, so it is not possible to define new actions after it, unlike the *Exception Handler*.

**Run Server Action:** Executes an action that runs logic on the server side (Server Action). Dragging the tool on the action flow will open the Select Action dialogue, for selecting an existing action or creating a new action. The action will be listed in Logic tab, under the Server Actions.

**If Statement:** The If node executes a branch of the action flow if the condition is evaluated as True and another branch if the condition is evaluated as False. The condition can be edited in the Properties Pane.

**Start Node:** Indicates where a flow starts executing.

**End Node:** When designing the process flow we must terminate the flow paths with the End node activity which we can drag onto the canvas from the Process Flow Toolbox. The process execution terminates when all of the flow paths in the main process flow (the one that begins with the Start process activity) reach their End process activity.

These are the components that we found most important to highlight and that will be most used in the context of this research, but all others can also be found in the OutSystems web documentation<sup>2</sup>.

#### 2.16.4 Testing Methods

Due to the nature of the visual language on par with the continuous integrity validation built in OutSystems, users do not need to worry so much about some technical integration challenges, given the abstractions that OutSystems provides, which makes applications less error prone and freeing developers to worry about other types of errors, like functional and behavioral errors (check if the application behaves as desired).

In OutSystems, the platform is kept partially open so it is compatible with the tools developers and teams like to use for testing. In fact, this is only partially true since this only applies to higher level tests. For these cases, the platform is flexible enough to allow the use of standard tools to test the UI or to perform API testing for example. Testing is integrated in the continuous delivery cycle so there are no losses in productivity. However, for lower level testing (Unit Testing, Component Testing) this is not applicable. There are some tools available that allow the execution of tests for applications created in OutSystems[48]:

The *Unit Testing Framework* is an old framework used for implementing, executing

---

<sup>2</sup>[https://success.outsystems.com/Documentation/11/Reference/OutSystems\\_Language/Logic/Implementing\\_Logic/Web\\_Logic\\_Tools/](https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Logic/Implementing_Logic/Web_Logic_Tools/)

and managing OutSystems unit tests. Developers find this approach effective for calculation engines and business components[53]. The framework was created to address the problem of managing and accessing test code.

The *BDDFramework* is an open-source component. It enables the creation and description of BDD test scenarios inside screen templates (blocks), with support for multiple tests in the same page and report statistics (like the number of successful/failed tests). It provides an Assertions library (*AssertSteps*), among other features. It was created to allow the description of tests in a textual language, promoting automation and test comprehension even for people who do not know the tests. We will explore this tool in detail in the next section.

For functional and regression testing in web applications, Selenium[67] can also be adopted, and any strategy currently used to test traditional web applications applies as well. Additionally, we have *Test Automator*[52], a browser (Selenium-based) and unit (Web Service-based) regression testing tool that helps guarantee the quality of the solution development, by automating the execution of tests over the application.

Quality Assurance within the Engineering department in OutSystems holds his base on top of two systems: *Dashboard* and *CINTIA*. They are both essential in order to guarantee the quality of the software produced at OutSystems:

- **Dashboard:** Dashboard is a legacy web application built in OutSystems, which centralizes all the logic related to build processes, test orchestrations and infrastructure/stack management. It provides a unique view over the state of every living branch, as it supports running the complete set of tests that run against a particular version of the product. It is still a fundamental tool in the validation cycle, as its logic has been developed and maintained over several years.
- **CINTIA:** Continuous Integration and Intelligent Alert system (CINTIA) is a in-house developed Continuous Integration system (built also in OutSystems), more modern and agile than the Dashboard. It allows for developers to have a quicker feedback on their changes on the most relevant branches, by continuously building the assets and testing them using a subset of the existing test base, providing a user-friendly UI with alerts and information on problems that occur on the build/test pipeline.

To run tests, some open-source frameworks are used, including *JUnit*, *NUnit*, the *BDDFramework* and legacy tests that were built on top of Dashboard, among others. A methodology of single branch development is employed. After each commit to the branch, the following steps are executed through a pipeline process: (1) build → (2) CINTIA → (3) Dashboard, after which a conclusion is reached: either the commit succeeded (green) or failed (red). For the commit to be successful, it must pass all three steps without a failed build or test. If the build process failed, responsible are asked to solve the problem as quickly as possible, to allow the activities to progress normally without causing many

problems. When there are failing tests, the pipeline is not blocked and developers can still perform changes and commit them. Cyclically, a *Code freeze* checkpoint is reached, meaning that there is a new branch created from the last successful commit in the main development branch which is meant to be released to the outside. After *code freeze*, typically there is a phase of *stabilization* (running the entirety of all tests present in the test base) and a phase of *dogfooding* (get the company to use its own product. It is an in-house product release, installing its new versions on internal systems to be exercised by Out-Systems employees in their daily work). This phase precedes the public release. When a candidate version has gone through all these rings and no problems were found, it is considered ready to be released.

## 2.17 UI Testing

User Interface (UI) testing refers to ensuring that the User Interface functions properly, that an application follows the specifications and that bugs are identified, all through the interface. Other than that, we check that the design elements are good. This involves checking the screens with the controls like menus, bars, colours, fonts, icons, content, buttons, among others. To perform UI testing, we usually use various test cases (set of conditions that will help the tester determine if a system is working as it is supposed to) and there are 2 ways of conducting this: manually (with a human software tester) or automatically (with the use of a software program)[31].

Selenium is an open-source UI automation testing tool for web applications across different web browsers and programming languages. It is one of the most recognized tools for UI testing and we will use it as a case study in this section because it is widely used in practice. Selenium is capable of interacting with the HTML elements of a web application to simulate user activity[13]. All major languages (Java, C#, Ruby, JavaScript, Python,...), browsers (Chrome, Firefox, Safari, Internet Explorer, Opera, ...) and Operating Systems are supported, with easy reuse across platforms, parallel execution and with a huge community, documentation and many releases over the years.

However, not all types of testing can/should be performed using the UI layer of applications. Unit testing or underlying application logic testing should not be done using the UI layer (like sub-cutaneous testing<sup>3</sup>)[21]. In addition, users often complain that UI automation testing is a time consuming, boring and expensive activity (that also does not support non web-based applications)[55]. Among the main problems pointed to UI layer testing we highlight the following::

1. **The UI of the applications is continuously changing:** As technologies evolve, application interfaces are constantly evolving so testing from them may not be easy

---

<sup>3</sup>Martin Fowler uses the term *subcutaneous test* to refer to a test that operates just under the UI of an application. This is particularly valuable when doing functional testing of an application: when we want to test end-to-end behavior, but it is difficult to test through the UI itself.

because of the constant changes, which sometimes makes us want to avoid such tests;

2. **Increasing Complexity of Testing Web Elements:** The features we implement in our web applications can include various web elements which are hard to maintain. Those elements can be embedded frames and other software products as well, and sometimes large websites can contain complex flowcharts, diagrams, maps, among other interface models and patterns;
3. **It is a slow process:** The problem with running automated tests through the UI layer is that we have to wait for the browser to launch. Secondly, most modern web applications have third-party tracking tags which could slow down the page load;
4. **Handling multiple errors:** Error handling has been an issue with UI automation testing because whenever there are complex UI test scenarios with tight deadlines, most time is utilized in creating UI test scripts. Additionally, testers normally choose manual testing over automation for UI testing. Having said that, error handling becomes extremely difficult when you manually revoke the error messages and automate them;
5. **UI testing makes the code review process harder:** Developers involved in an application development often have different coding styles. Creating UI automated tests takes more time than other types of testing and creates heavier coded tests to perform the browser calls. Without maintaining the coding standards, it will be extremely challenging to review, modify or maintain this code.

Another important point to note is that in large applications it may sometimes not be recommended to use a BDD test automation tool like Cucumber (which is the most known) together with UI testing using Selenium, when there are feature dependencies.

At the time of development, when each feature is developed one by one in each iteration, the feature files would be focused on one feature itself. At some point, when we have multiple features, we need to start thinking about testing these, not only in isolation but also creative scenarios where we combine different ones (integration).

UI testing for functional verification is not recommendable but it is for API level testing with integration tests. It should be reserved for checking the user flows through the application, or for end-to-end testing, making sure relevant expected modules are present on the pages as the user navigates from one page to the others.

Another important aspect is that feature files for testing screen transitions that will combine several features makes no sense and goes against the BDD process, as we are testing the system and not a single feature. Each feature file in Cucumber typically represents a single feature.

Briefly, Cucumber is more relevant at the feature level and Selenium for end-to-end integration testing. It is also not recommended for the same application to use these 2

technologies separately for each type of testing as this could lead to duplicate testing. This would go against the benefits of automation and UI testing is never a good idea to duplicate since they are slow and brittle[23].

## 2.18 BDDFramework

The BDDFramework is an open-source test automation framework for the OutSystems Language. It allows users to **specify BDD scenarios using the Gherkin syntax within the development platform** (Service Studio) to test OutSystems' applications. This component is available for download in the forge<sup>4</sup>. Its biggest focus and the reason it was created back in 2016 was to allow OutSystems developers to **write tests in a more structured and self-descriptive way using the Gherkin syntax and enabling automation**, rather than in support of the BDD practice itself in an OutSystems context (despite its name). The idea behind the tool was to find a **standard way of describing tests in OutSystems** and its structured and self-explanatory nature seemed ideal for this, as these constitute the basis for test automation and otherwise a form of documentation, respectively. Its introduction made it easy for developers to understand tests they had not written and to understand why they were failing more easily.

This component provides the user a **set of templates for describing and filling in Gherkin scenarios** within Service Studio. Inside each scenario, a description is requested and there is space to place the *BDDsteps* that will fill the Gherkin structure in each scenario, within their respective placeholders (Given, When and Then). Each *BDDstep* is composed of a textual description and is implemented by a *Screen Action* that must contain the code necessary for its implementation. There are also *setup* and *teardown* steps available, which correspond to logic that can be executed before or after the scenario implementation, respectively, for example to create or delete data required for running the test. In the logic tab a library with various types of *AssertSteps* (value, positive, negative or default) is also made available to the user. It is also possible to put in each test screen the final result block which reports the results of the execution. When the execution is done statistics of how many of the total scenarios were executed successfully (i.e, all its steps had the intended result) are available, upon publishing. It is also possible to see which steps failed for each scenario, and why they are failing[57].

One of the features of the framework is the fact that it allows running tests through a REST API endpoint. This is particularly useful if we want to have the tests being triggered by some sort of orchestration process[49, 56]. An important note is that in the *BDDFramework* the tests are executed sequentially and inside each scenario the steps are also performed sequentially in the order they appear arranged (Setup, Given, When, Then, Teardown).

---

<sup>4</sup><https://www.outsystems.com/forge/component-overview/1201/bddframework>

To better demonstrate how this tool really works in practice for test automation in OutSystems, we will demonstrate it a practical example:

We have an OutSystems' application that returns the kind of a triangle given the length of the 3 edges. About the application:

- A triangle is called **Equilateral** if all sides have the same length;
- A triangle is called **Isosceles** if 2 sides have the same length, and the other is different;
- A triangle is called **Scalene** if all sides have different lengths;
- If we enter lengths that do not form a valid triangle then the application returns that it is **Not a triangle**. To form a triangle **all sides must be smaller than the sum of the other 2** ( $side1 < side2 + side3$  ;  $side2 < side1 + side3$  ;  $side3 < side1 + side2$ ) and all sides must have length  $> 0$ .

We will build a BDD scenario to test the Equilateral triangle:

Listing 2.2: Equilateral Triangle Scenario

```
1 Scenario - When all edges are valid and have the same size (3) then the result
2 must be Equilateral
3 GIVEN The first edge has length "3"
4 AND The second edge has length "3"
5 AND The third edge has length "3"
6 WHEN The user clicks in the button to calculate the result
7 THEN The result should be "Equilateral"
```

In figure 2.7 we can now see the scenario described in the development platform using the *BDDFramework*:

And in figure 2.8 the results of the test execution obtained when we publish the test project after all steps have been implemented in the corresponding actions.

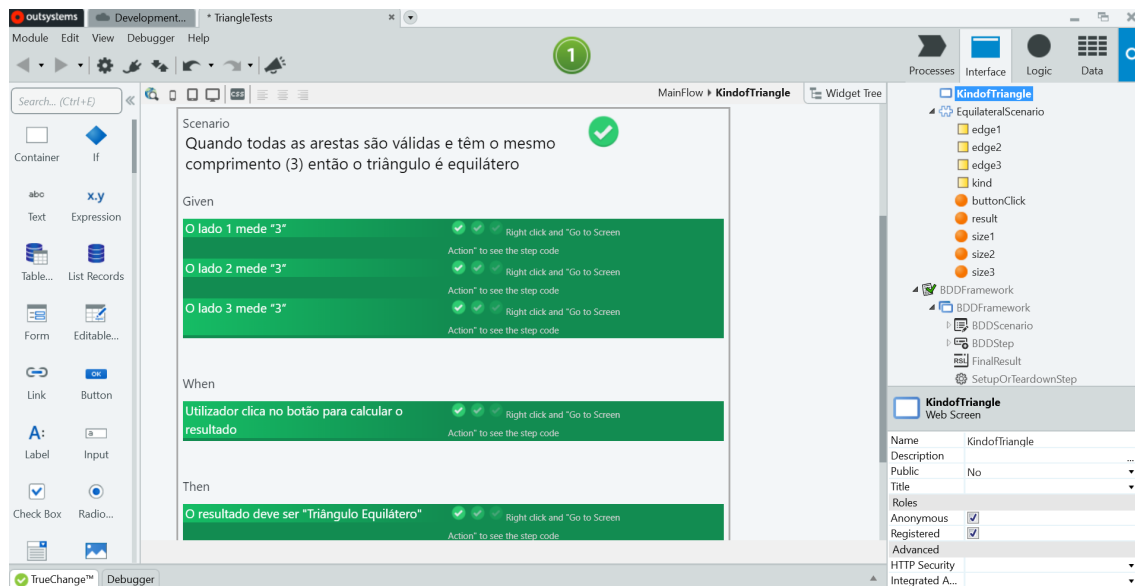


Figure 2.7: BDD scenario specification to test the Equilateral triangle within *Service Studio* using the *BDDFramework*

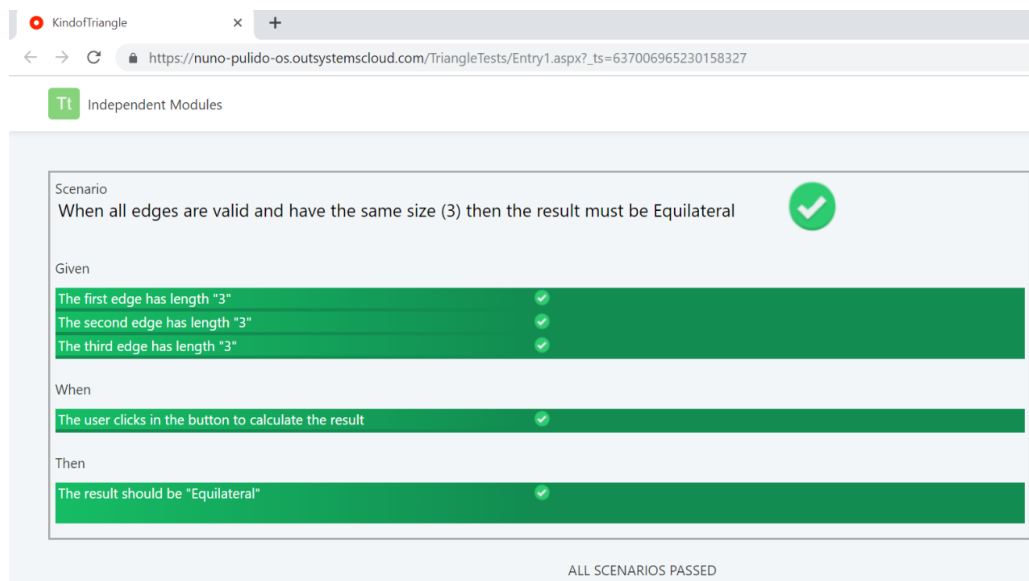


Figure 2.8: All *BDDSteps* that constitute the (single) scenario were successfully performed, culminating in the positive outcome of the scenario.

If any *BDDStep* was not verified as correct (failed) and had an non error-free execution then the scenario would fail, like in figure 2.9.



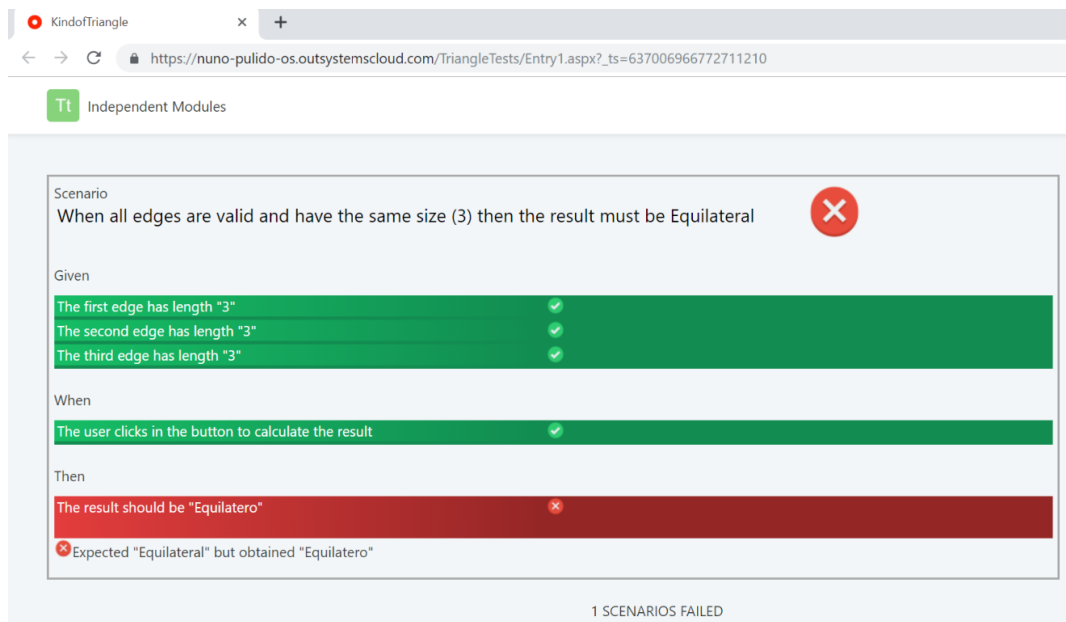


Figure 2.9: A *BDDStep* failed, resulting in a failing scenario. the failure report is displayed below the failing step.

In this case the failure was due to the fact that the test was expecting "Equilateral" triangle as output but instead obtained "Equilatero" from the previous steps, causing that step to fail as it is possible to see in the step implementation (figure 2.10).

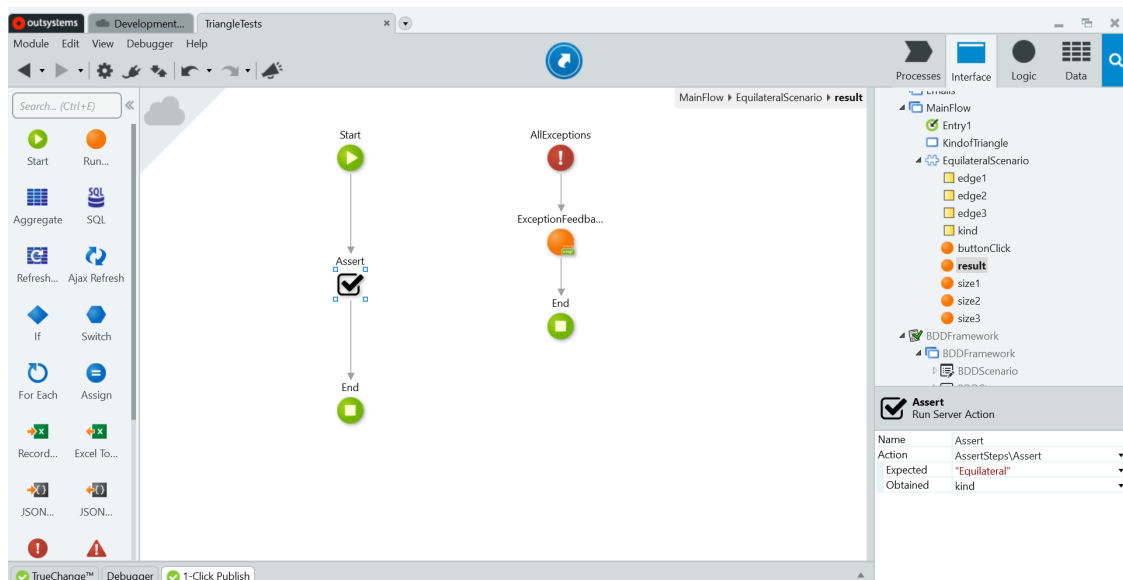


Figure 2.10: The expected result was "Equilateral" and the obtained is the value of variable *kind* previously assigned in the logic implementing the when clause

In the *BDDFramework* the developer has the following blocks available:

**BDDScenario:** corresponds to the template that represents a scenario individually, in which it is possible to place its description, the *BDDSteps* inside each of the clauses (Given, When, Then) and the setup and teardown steps.

**BDDSteps:** represent each of the steps that compose the structure of Gherkin. These are inserted inside one of the scenario placeholders: Given, When or Then. *BDDSteps* are associated with an input text widget for the user to enter the step description and are linked to a screen action that contains the implementation logic of the step.

**Setup or Teardown steps:** These are the steps that contain the setup or teardown logic of the scenarios (hook steps that do not belong to the scenario structure but can be used to create data or perform actions necessary to perform the scenario itself, at the beginning or at the end of it, respectively)

**Final results block:** statistics block that tells us how many scenarios failed and in which steps.

By convention and to standardize the writing and facilitate the comprehension of the tests, some practices were adopted regarding the structure of BDD tests by OutSystems developers. Some of these "recommended practices" come from conventions from the BDD process itself and others from the nature of the OutSystems language.

### 2.18.1 Recommended practices

For the description of scenarios and test implementation, a testing project should be created. It must import the dependencies of the *BDDFramework* (to have access to its features, otherwise it will not be possible to test in this way) and the module with the application to be tested (which should be immediately developed with this kind of testing in mind, **exporting the necessary logic in server actions** to run in the test project, making it easier for the developer to test). By convention, if we are testing the *KindOfTriangle* application, the test project should be named *KindOfTriangleTests*.

Typically each *BDDScenario* is placed within a (reusable) *Web Block*. All *Web Blocks* that have scenarios that test the same functionality are placed within the same *Web Screen*. For example, in the application shown earlier we would have a *Web Screen* (for the triangle type functionality) and inside we could have at least 4 web blocks with test scenarios (for the equilateral, isosceles, scalene and the not a triangle cases). **Screen names should be suggestive of the features and scenarios being implemented**, and the description of the *BDDSteps* and the scenario itself should be a **short, concise and meaningful**, as it is in the nature of BDD that developers can look at the description of a test they did not know and understand where it is failing and what is the functionality (system behavior) being tested. Tests should be self-describing as all actions that implement them. In addition to these cautions, developers at OutSystems also have some naming conventions featuring some "tricks" like a small code or counter to make it easy to identify tests when they are run in test suites with many scenarios so they can quickly link them to the corresponding testing code within Service Studio. Scenarios and the corresponding web blocks should

have identifiers and these identifiers should be acronyms for the methods being tested. This is very useful, for example, to identify which tests have failed in Continuous Integration. Regarding actions and their organization: shared behavior between tests should be abstracted into Server Actions and then used across tests and those should then be grouped into folders based on a topic (security, authentication, ...) and/or purpose (eg. given, when, then)[49].

### 2.18.2 Data-Driven Testing

Data-Driven Testing (DDT) is a test design technique where test scripts read and interpret test data from some kind of **data source**. The iterative repetition of the same sequence of test steps is performed in order to drive the input values of those steps and/or the expected values while verification steps are performed. Sometimes, there are some data sets that in which we have to run the same tests on. It is a time-consuming and inefficient activity to create a different test for each entry of the data set values and DDT helps us in this case, allowing us to reuse the test implementation code to be fed by a data set [41].

This type of testing can be very useful when we want to test a large set of values for a single test without defining different scenarios and OutSystems already has some coverage for this using the *BDDFramework*, applying some techniques that are facilitated by language structure[59].

In OutSystems this is done by creating input parameters for the data source variables that we will use in the test and using expressions in the description of the Gherkin steps instead of text instantiating the test values, as it was previously done[49].

Finally, we can now easily setup a set of data-driven tests for the **Scenario Outlines** by using *Static Entities* (set of named values, enums or literal values stored in the database with global scope) and *Table Records* (widget that displays the records of an Entity in a tabular layout) [51] [50]. First, we define a static entity containing all of the examples we want to test and create a *Web screen* with *Table Records* being fed with all of the data from that *Static Entity* and showing the Scenario Outline *web block* for each row[49, 59]. This process is not supported by any BDDFramework specific mechanisms but is easily reachable with the help of the OutSystems structure and characteristics



## RELATED WORK

In this section we present and summarize the work of some authors related to the topics addressed in this dissertation.

### 3.1 Low Code and BDD

The literature relating Low Code technologies to Behavior Driven Development is very scarce. However, some research work has been done, as the one conducted by Stephan Braams that aims to analyze the practice of Behavior-Driven Development in Low Code Model-Driven Development (LCMDD) with the case study of a company that uses the Low Code Platform Mendix[12]. Braams argues that nowadays testing is still a big bottleneck for the software development success, stating that organizations spend about 50% of their resources at this stage of the development cycle, but the this stage is still seen as unpopular and boring, and still very susceptible to failures and often the problem is building software that is not what the customer wanted. Braams considers Unit Testing as the basis of the testing processes. However, in Low Code, units of code are not often accessible because the code is not directly or conveniently “handy” to be tested and some research has to be made to address this in the Low Code domain. In addition to the setbacks mentioned in this research, the author goes further and argues that many times people do not define proper priority between features, not measuring accurately what is value for the client, having as consequence the creation of many non-important features. Braams justifies that BDD can be a good complement to LCMDD, since by nature Low Code languages already cover the lower level tests (such as unit and component testing) as well as tests for non-functional requirements (performance, security, among others) given the structure and characteristics of these languages. It turns out that the types of tests that are most needed (not so covered) in this LCMDD domain are precisely the types

of tests that BDD focuses on: functional testing, scenario/example (acceptance) testing, exploratory testing, among other types of testing.

One thing that is very important in BDD is the connection between the Gherkin descriptions and their implementation. It is important that they maintain a close relationship both for reporting purposes and for testing or debugging (traceability). This is not always achieved as evidenced by the study of Silva, which aims to investigate the use of ontologies for specifying the automated test using an approach based on Behavior-Driven Development (BDD)[68]. According to the author failure to trace tests to requirements is one of the main causes for the failure of software projects. He noticed that traceability between requirements and tests was rarely maintained in practice and the cause was the incapacity to update traces when requirements change. One of the reasons was the difficulty to conduct the process manually. The solution according to Silva's research is to use the ontological model which describes only behaviors that report steps performing common actions directly in the User Interface through Interaction Elements. In OutSystems this is already achieved by the *BDDFramework*, since the description of steps is linked to their implementation but we are within the platform. This case may be interesting for cases where requirements are separated from implementation, or for project management and issue tracking tools.

## 3.2 BDD principles and supporting tools

When we start to focus more on the technological part of BDD, especially when we talk about the concept of test automation, there is much more research done in the area often perhaps because BDD is frequently associated with test automation. It is true that the nature of Gherkin and the scenarios that are obtained in the BDD process provide an excellent basis for test automation but the truth is that BDD is much more than this. This is one of the main conclusions of the study performed by Wang and Solís[70]. The authors studied the literature to find the BDD characteristics and then seek support for them in the most commonly used in practice BDD testing automation tools, like Cucumber and SpecFlow. One of the main conclusions reached by Wang and Solís was that these tools only supported some stages of the BDD process, especially the development phase, lacking support for the analysis phase (feature definition) but especially for the planning phase (identification of business values).

The framework comparison results can be found in figure 3.1:

The methodology used by the authors to find the BDD characteristics represented was the following: After reviewing several studies and defining an initial set of the BDD characteristics, they analysed one framework at a time using the same set, recording how the framework supported them. If they found a characteristic that was not in the list, they went back to the literature to understand if it could be considered a BDD characteristic or not.

| Support of the BDD Characteristics              |                     | xBehave Family |         | xSpec Family |       | StoryQ | Cucumber | SpecFlow |
|---|---------------------|----------------|---------|--------------|-------|--------|----------|----------|
|   |                     | JBehave        | NBehave | RSpec        | MSpec |        |          |          |
| Ubiquitous language definition                  |                     | x              | x       | x            | x     | x      | x        | x        |
| Iterative decomposition process                 |                     | x              | x       | x            | x     | x      | x        | x        |
| Editing plain text based on                     | User story template | √              | √       | x            | x     | x      | √        | √        |
|   | Scenario template   | √              | √       | x            | x     | x      | √        | √        |
| Automated acceptance testing with mapping rules |                     | √              | √       | x            | x     | x      | √        | √        |
| Readable behaviour oriented specification code  |                     | √              | √       | √            | √     | √      | x        | √        |
| Behaviour driven at different phases            | Planning            | x              | x       | x            | x     | x      | x        | x        |
|   | Analysis            | √              | √       | x            | x     | x      | √        | √        |
|   | Implementation      | √              | √       | √            | √     | √      | x        | √        |

Figure 3.1: The BDD Characteristics support from seven BDD toolkits

The resulting table represents a good basis of comparison for the validation work that will be performed later in the course of this dissertation. In section 3.2.1 a critical overview of the results and the criteria chosen to conduct this study and produce this table will be presented.

### 3.2.1 Discussion on the study by Wang and Solís

The following is a critical analysis aimed at the study conducted by Wang and Solís [70], previously described. We believe that this study can be very interesting to support the validation of our work and include in it the *BDDFramework* and the Prototype developed in this dissertation. However, we need to point out some things we do not agree with and we will do so by individually analyzing each of the criteria considered to draw the final table.

- Ubiquitous language definition:** Although this is a central concept in Behavior-Driven Development since this language is required to express behaviors in a business speaking domain, we disagree that this process should be considered in a test automation tool. This should be considered in a different phase of the process. The definition of a ubiquitous language should be taken into account when writing tests using the Gherkin syntax, but this language must be defined elsewhere, in a more initial phase of the process. However, it can be complemented during the description of the scenarios, since it is natural that new terms arise constantly. This is crucial to enhance communication among stakeholders and the main idea here is to run the tests without depending on an Integrated Development Environment (IDE) so that anyone involved can write the scenarios, and therefore this definition gains even more strength outside an automation framework. However, this factor can be seen as an extra, such as an auxiliary dictionary of terms, either accessible by the command line or as suggestions in an IDE, but we do not see this as essential, so much so that none of the tools studied in this investigation covers this point.
- Iterative Decomposition Process:** refers to the process of identifying the features that will be part of the software product to be developed. This process begins with the identification of business outcomes. In the first place, we define the behaviors

offered by the system since these are easier to express and to analyze in terms of value. The functionalities are specified in User Stories from a User point of view and then in Gherkin scenarios. The implemented features must bring value to the client and this must be taken into account in the scheduling of the development process. Again, this is an aspect that may be outside the scope of the automation tool and it is more important in a project management tool (such as *Jira*), where priorities among features can be set. The responsibility of the testing framework should be only to automate and execute the features that are passed to it (already specified in Gherkin syntax), allowing, however, complete freedom regarding the execution of scenarios. The scenarios that will run in each execution must be defined as parameters.

- **Plain Text Description with User Story and Scenario Templates:** One of the main purposes of using such tools is to write the structured textual descriptions that constitute the specification of the functionalities (the user stories from the point of view of the user and the Gherkin Scenarios, specifying the behavior of the system) to then be interpreted and converted into test code by the automation tools. This aspect is therefore very important, especially from the point of view of the scenario definition as they will form the basis of testing documentation and automation.
- **Automated Acceptance Testing with Mapping Rules:** It represents the main purpose of the analyzed frameworks: Automated generation of test code for the defined scenarios, thus making the specifications executable. Each step will be mapped into a test method. The way this is done can vary slightly from tool to tool, and some frameworks use more or less complex regular expressions (also depending on the programming languages supported) that also allow parameter detection. Mapping rules between method names and phrases in Gherkin can also vary, as well as the structure and organization of tests in classes that test the same features or not.
- **Readable Behavior Oriented Specification Code:** In BDD the test code should be part of the specification of the system (living documentation) and the methods should be self-explanatory of themselves and describe the functionality so that a person that looks at the code for the first time easily realizes what it is intended to do/test. The frameworks support this by generating methods and classes with the names of the steps and with the description of the Gherkin steps, respectively. Some also support writing scenarios as code directly, with the aid of annotations. We do not agree with Wang and Solís' rating of Cucumber on this one as it is one of the most complete tools and currently supports this, just like the rest, possibly not at the time but currently it does.
- **Behavior-Driven at Different Phases:** Since frameworks do not allow defining and specifying business outcomes, the authors state that the planning phase is not supported by any of the tools. However we consider that this may run away from the scope of what is really needed in the framework, being just a possible



extra. In the Analysis phase, some tools support the process because they allow specifying the features through scenarios described with the Gherkin syntax and some of them even with User Stories. Finally, in the implementation phase, the same that support the analysis phase also support the automated generation of test classes and methods. The three table entries concerning this topic (planning, analysis and implementation) are somewhat repetitive and summarize what was described in the other topics, clearly dividing the process into distinct phases, which may be interesting as a conclusion of the table but not that interesting to do within it.

Some of the points taken into account in this study by Wang and Solís are considered inadequate or non-priority to be relevant in a BDD practice support test automation framework, namely the first 2, which despite their importance to the process are outside the scope of a test automation framework and as such should not be considered in this classification study. In addition some of the tools presented also have a different purpose than what is intended to be achieved during this thesis and as such will not be considered, only the two we consider most important, Cucumber and SpecFlow. After a practical analysis of some commonly used frameworks, we present a new model for evaluating BDD frameworks.

### 3.3 Software evolution challenges

One of the biggest challenges inherent in BDD relates to the growth of software being developed. Over time, many projects tend to get bigger and increase the number of features. Consequently, the number of User Stories and Scenarios will also grow, which may make this process difficult to maintain, as stated in a study conducted by Binamungu et al.[9]. They found that the maintenance challenges reported are likely to be less significant if typical test cases contain a number of scenarios that can be managed by hand. When test suites are not small, individual manual inspection of all scenarios is a difficult and costly task. With the growth of software, these maintenance challenges tend to become more prominent:

1. In large test suites it can be hard to locate the origin of the faults, due to the large integrations and the large number of files (traceability becomes harder);
2. Changing specification or inserting new features;
3. Duplication detection in BDD specifications;

Some research has been done to address these problems among which the study carried out by Lucassen et al. stands out [10]. They propose the Behavior-Driven Traceability Method (BDT) that takes a different standpoint on automated traceability: “establishing ubiquitous traceability between user story requirements and source code by taking advantage of the automated accessibility tests that are created as part of the Behavior-Driven

Development process”. The proposed BDT method automatically establishes ubiquitous traceability on top of the BDD process. It relies on two features of BDD: the steps that define the Gherkin Scenarios and operationalization of these steps on the UI. They explain how the BDT Method takes advantage of these characteristics and introduce the BDT Tracer which builds a matrix that records the source code and methods called for each user story: “When a software development team creates individual BDD tests for each user story, applying BDT results in a BDT Matrix that allows a developer to request all the source code invoked to realize a given user story. By applying smart filtering techniques the BDT Matrix can then be used to produce a variety of reports, such as methods that are never called in the entire test suite to identify dead code (if the test coverage is good enough), or all the classes involved in a specific user story to inform developers modifying or refactoring a user story’s code”. On the other hand, in another investigation by L. Binamungu et al. the authors developed an algorithm to detect duplication of examples. They state that when the suites of examples are very large, they can be difficult and expensive to change. Duplication in this case is difficult to detect manually by developers. The resulting algorithm detected more than 70% of the injected duplicates in the tested systems[11].

## 3.4 Automation frameworks

### 3.4.1 How BDD frameworks work

We saw how to express behavior using the Gherkin Syntax, with the “Given,When,Then” notation in section 2.7. Many frameworks use this syntax to automate tests. These take the textual description of the steps defined for each scenario and create methods to execute the corresponding test code. Tests can be run in different programming languages and can be implemented at different levels of the application architecture (UI, API, and so on), according to the framework we are using. All the mostly known frameworks work in a very similar way which we will present below.

The following examples are based on the examples given in [69]:

Listing 3.1: Scenario Example

```
1 Scenario: Earning points from an Economy flight in TAP (scenario description)
2
3 GIVEN The flying distance between Lisbon and Faro is 300km (step)
4 AND I am a standard Flyer member (step)
5 WHEN I fly from Lisbon to Faro (step)
6 THEN I should earn 50 points (step)
```

The previous scenario will then be interpreted by the application, step by step, and the test environment will be set. For example:

**Given** the flying distance between Lisbon and Faro is 300km

A test database is configured to provide the correct distance between the two portuguese cities. This can be done in many ways like through an API call, directly through the UI or manually inserting data into the test database.

**When** I fly from Lisbon to Faro

In this step we want to record a flight from Lisbon to Faro and check how many points the member earns. Tools like Cucumber, JBehave or SpecFlow can not turn a text scenario into an automated test by themselves so we need to specify what each of these steps means in terms of our application and how it must manipulate or query the application to perform the task. This is called step definition (or step interpretation).

Step definition is essentially a piece of code that interprets the text in a feature file and specifies each step in a test method like this:

Listing 3.2: *Given* clause

```

1 @Given("The_flying_distance_between_{$departure}_and_{$destination}_is_{$distance}_km
2 ")
3 public void flyingDistance(String departure,
4                             String destination,
5                             int distance) {
6
7     //prepare the data for this trip
8
9 }
```

Listing 3.3: *And* clause

```

1 @And("I_am_a_{$status}_flyer_member")
2 public void defineMember(String status) {
3
4     //prepare the data for this member
5 }
```

Listing 3.4: *When* clause

```

1 @When("I_fly_from_{$departure}_to_{$destination}")
2 public void flightFrom(String departure, String destination) {
3
4     //add flight to member
5 }
```

Listing 3.5: *Then* clause

```
1 @Then("I should learn $points")
2 public void pointsCalculation(int points) {
3
4     //calculates the points earned
5
6 }
```

Step definitions can be implemented in many programming languages depending on the frameworks. For example, in SpecFlow step definition might look like this:

Listing 3.6: Step definition in SpecFlow

```
1 [Given(@"The flying distance between (.*) and (.*) is (.*) km")]
2 public void defineTheFlyingDistanceForATrip(String departure,
3                                             String destination,
4                                             int distance) {
5     ...
6 }
```

The step definitions do whatever it needs to perform the steps, although we can also tell the frameworks how the data extraction is done into the step definition method. It is a good practice to keep step definitions simple and meaningful[69].

### 3.4.2 Well-known examples

Cucumber is a very popular BDD test automation framework originally created for Ruby[17, 38]. It supports several languages for the definition of scenarios. An example is Cucumber-JVM which is a recent Java implementation of Cucumber that allows writing and defining steps in Java and other JVM languages. Cucumber also supports Python and JavaScript. Another cucumber extension, Cucumber-JS, lets us define scenarios in JavaScript which is gaining increasing importance in modern web development. On the other hand, if we are in a .NET environment our best option may probably be SpecFlow [72]. SpecFlow is an open-source Visual Studio extension in the .NET and Windows development system which allows us to automate gherkin scenarios. Another important tool is JBehave [44]. It was developed by Dan North, the founder of BDD, and allows to define steps in Java, Scala and other JVM languages. Although these tools are applied to different languages, their operation is very similar and is based on the steps specified in section 2.7. Some of these tools will be explored in more detail in the next chapter.

## CASE STUDY AND INITIAL CONSIDERATIONS

Throughout this section, we will detail the practical work that was done at the beginning of this dissertation, how the problem was addressed, and how we adopted our approach to face the problem. We will also summarize the information collected with the interviews made to people in the area who deal with testing in OutSystems on a regular basis to understand what is missing and what this dissertation can do to help them. Finally, we will be detailing some features that we conclude are necessary to address in the framework prototype.

As a case study for this research, we have the real example of OutSystems R&D, with which it was possible to establish direct contact since this research was carried out at its facilities. As such, our investigation has been very focused on this platform and its specific characteristics. After learning the OutSystems language, testing with the *BDDFramework* and investigating the principles of BDD, we will analyse this real case example, through the realization of the mentioned interviews. With these, we hope to realize how both the development and testing processes work and get some useful feedback in order to understand what is lacking or what can be improved.

After this process we had to decide between some implementation approaches. It will be all described in this chapter.

### 4.1 Interviews

The interview questions covered the development process and especially the testing phase that is practiced in OutSystems, either about the BDD process itself or the experience with the *BDDFramework*. The respondents include two Outsystems' Quality Owners<sup>1</sup>,

---

<sup>1</sup>Developers who have responsibilities related to the quality area, namely to organize the testing process within teams.

two product developers, an external contact in an OutSystems client that applies BDD as a development process, and a contact in the United States that works for OutSystems with important knowledge concerning both BDD as a process and the framework.

#### 4.1.1 Interview Questions

We customized the questions according to the respondent experience with BDD and the framework. For instance, the only contact that the elements of the productivity teams had with the *BDDFramework* was for test automation purposes and not with the BDD process itself and their knowledge is restricted to the use of the tool. On the other hand, Quality Owners have a more global view of all stages of the process. However, the proposed questions were very much around the testing process carried out with and without the tool for the interviewees who had only used the framework. For the ones with more knowledge about the BDD process itself, we asked more questions regarding the type of software and phases to which the BDD refers, its applicability in the context of OutSystems and also which features should be included in the framework. [Appendix 1 Interview Scripts](#) contains the interview scripts, although in most of the interviews these were deepened according to the answers, experience and the direction of the conversation.

#### 4.1.2 Interview Results

According to the results it was clear that there was a **test documentation problem**. Whenever it was necessary to look at a test (e.g. when it failed) there was a high probability that the test would be misinterpreted because people often looked at tests that they had not designed and had never seen before and it was difficult to understand them. The framework has helped developers to design and document tests. With the introduction of the *BDDFramework*, just by looking at the report of the test, developers were able to realize **how the tested functionality should work** and **what each test is supposed to verify** because of its description and the step definition in the Gherkin syntax, with plain English text. This constitutes a form of **living documentation** with **self-explained tests**. The complexity of the testing process has been substantially **reduced and the process accelerated**, with respect to **tests made with the OutSystems language** - the ones addressed by the framework. Developers in OutSystems have **no standardized way of testing**, each team tests as it works better for them, so this framework brought something new: one tool for test automation specific for this language which has the potential to become the **test standard for OutSystems**.

It was also verified that the testing phase is a little underestimated by developers, as it was described as a “boring” and time-consuming activity. Many developers have **no training in testing** and the fact that OutSystems is a Low Code language may give the (wrong) illusion that this phase is less important than in other languages (although this is true for some kinds of integration testing). In OutSystems, an Agile development approach is followed and testing is **performed by the developers**. There are no “testers”

to verify and validate software. Regarding the name of the framework (*BDDFramework*), it can also be **misleading** since it was chosen because the behavior scenarios are described with the Gherkin syntax but **not in reference to the BDD process itself**. This tool was not designed to be an assistant to the BDD process, but instead to **automate tests using the Gherkin with the OutSystems Platform**.

In relation to other tools in other programming languages, like Cucumber or SpecFlow, which are two of the most popular ones, the fact that the *BDDFramework* is in the same environment as the development platform (Service Studio) can make the task for the developer simpler, but on the other hand can make it more difficult for business people to participate in the scenario description process in a more active way, since using Service Studio might not be as adequate for their profile as a regular text editor to write the scenarios.

In the mentioned tools, the description of scenarios is done in plain text files that are automatically converted to testing methods and classes, which is good for the integration of the business and can also be interesting from the point of view of the developers, since the steps already known by the system would be possible to reuse and this is not currently supported by the *BDDFramework*. At this moment, developers need to analyze the manually defined steps, even to use them in other scenarios. Such reuse would be possible with the definition of a ubiquitous language and direct mapping from text to test code. With the *BDDFramework*, the logic that implements the steps is defined manually. Therefore, it takes the services of a developer to make the conversion.

In terms of the **negative aspects**, as already mentioned the framework name can be a little “inappropriate” and lead to misinterpretations. Not everyone knows what the BDD process is or what is the Gherkin syntax, so **when using the *BDDFramework* people may mistakenly think they are applying BDD as a process**, but in OutSystems this process is still not used in practice, although there are plans for future experiences in some development teams. In addition to this aspect, there is still a learning curve involved and some say that the process is **very laborious and that the results do not compensate the work**. The tests tend to be neglected, although in practice we find that this does not make much sense, especially from a software behavior point of view (which is precisely the basis of BDD) and functional testing.

The interviewed members of productivity teams also mentioned that the framework could offer some more complete monitoring and test coverage information, although this last issue is already a bit beyond the scope of what is intended with the framework. Integration with some tools like *JIRA*, for project management and issue tracking, was also requested. Another important thing to keep in mind is that the framework only works for Back-End testing. Speaking of scalability, interviewees also suggested the functionality that allows performing **setup** and **teardown** hooks at test suite level and not only for each Gherkin scenario individually. Another current limitation of the BDD Framework is that sometimes the way OutSystems code is done following “recurring practices” of the language can generate some problems and make it harder to test, although this is more of

a platform issue. For example, if we do an abort transaction operation in the database, the framework loses all asserts that have been made so far in that step. Within OutSystems the tool is only used by some teams. Most of the teams integrate different languages and technologies, so it is not practical to test using the *BDDFramework* and to write BDD tests. Some teams also did not adopt the tool because they were working on ongoing projects and they would have to redo the tests in BDD which would be a lot of work. These were the main reasons that made some of the teams not adopting the framework, but it should be noted that some of these suggestions are beyond the purpose of it.

One of the interviewees, who applies BDD as a process in an OutSystems' client, despite not using the *BDDFramework*, stated that the results were quite satisfactory and that **User Stories played a fundamental role in the development process**. These describe the features to implement and result from the conversation between the stakeholders. They are a form of communication that can be perceived by everyone. At the end of each Sprint, Product Owners<sup>2</sup> review what has been developed and compare it with the previously defined User Stories, which constitute a **Product Acceptance Stamp**. The example of this interviewee shows us that there is at least **one example of BDD being applied as a process that brings value to OutSystems-based development**. He said that the reason they started practicing BDD was the **value it brings to the business**. He described that at the beginning of each sprint, all stakeholders try to perceive and interpret together the values for the business. The User Stories are sent early to interpretation and are only closed and given as completed after properly tested. According to him, Product Owners are already able to describe scenarios in Gherkin syntax too. In this case, the framework used is *SpecFlow* for the description and automation of tests. *SpecFlow* allows, among other things, the integration with other software that enables business to monitor the progress of development, in this particular case Microsoft's Visual Studio Team Services. Here the client can see the state of the project, namely keep track of which functionalities are already developed. It is important to note that they are using another framework (*SpecFlow*) in an OutSystems project because all the implementation of steps is done in another technology to use *WebDriver* to interact with the User Interface (UI) of the OutSystems applications. UI tests tend to be brittle, slow and hard to maintain (see section 2.17). Implementations of *SpecFlow* with OutSystems will always be interactions with UI or APIs in OutSystems applications because of the language barrier and there is a **clear separation between scenarios and test code**. If we try to perform a test under the UI of the application (sub cutaneous testing [21, 22]) with Gherkin in OutSystems, we need to have our application fully prepared to expose the features by APIs (which is often not feasible). We want a **solution for low code implementations in OutSystems** since we do not want to jump to a non low code context to do BDD testing.

On Low Code platforms, development is fast so it is important that communication is well managed to guarantee that the speed of development is well employed and to

---

<sup>2</sup>Team members representing the business



minimize the risk of failure and regression. If the testing process is slow or very likely to detect faults then the **regressions during the development phase will be very pronounced in cases of error** since testing cannot keep up with this pace. The code that will have to be redone can be extensive because it corresponds to old phases and everything that has been done since then can potentially be incorrect.

The interviewed OutSystems' contact in the United States that works closely with the practice of BDD argues that the *BDDFramework* can become a vital vehicle for having customers doing two things: pushing testing activities closer to the moment when development is done over user stories, and establishing a standardized way for developers and testers to have a conversation over tests and behaviors. He also admits the future possibility of including the business people and thus having the full *Three Amigos* involved in the process. According to him, **most customers think in requirements first**, then development and only at the end in testing. In his opinion, it is **vital to approximate these stages** and especially the testing phase should be done alongside development. In fact, testing should be a part of the development process and developers themselves should test the software and know how to do it properly. The BDD can take a huge part in this, bringing the business to Sprints, always with the aim of "Write correctly from the first try", creating tests that check if the software is correct and if it works the way customers want. Another important aspect that was mentioned in this interview is that in OutSystems development is fast, which is very good but raises some challenges to the testing phase, hence again the need of associating testing to development and discovering discrepancies as early as possible that can have a greater impact given the speed of development. He also argues that automation is the starting point for BDD and that the *BDDFramework* is ideal for functional testing and a good tool for the practice, even though he considers that it should have more extensions that allow integration with other software.

To sum up all the information, the main strengths of the BDD Framework from the point of view of its users are:

- **Self-explanatory tests** which constitute a form of living documentation of the system;
- **Standard for test design** in OutSystems;
- Testing **process acceleration** in case of errors;
- **Tests run automatically** and can be organized into test suites;
- API testing.

And the main weaknesses are:

- The fact that scenarios are described within service studio, although it can be argued that it facilitates the developer (because it is the same development environment),

makes it difficult for business people to participate. In other tools the steps that make up the scenarios are described in text files;

- Does not support direct reuse of steps. The developer always has to manually detect if the steps are equal to reuse the same logic. With text to code mapping this is not required;
- Integration with other tools (like *JIRA* for example) could be supported to help in project management and issue tracking.

## 4.2 Analysis of BDD Automation Frameworks

In this section we will present two of the most commonly referenced automation frameworks during this dissertation. Both are well-known BDD process support tools. We focused particularly on them, as they are two of the most used today in practice and we consider that they are an excellent outline of what we intend with an automation tool. They, together with the interviews, were a valuable help in the specification of the prototype functionalities we developed during this dissertation.

### 4.2.1 Cucumber

*Cucumber* is perhaps the most widely used BDD testing automation framework. It was originally created as a command-line tool by members of the Ruby community. It has, since then, been translated into several development languages, including Java and JavaScript. When we run *Cucumber*, it reads our specifications from plain-language text files called **Feature Files**, parsing them to find scenarios and to generate the test skeleton (step definitions) automatically. *Cucumber* also supports running the scenarios against our applications. Each scenario is a list of steps for *Cucumber* to work through. *Cucumber* can understand these feature files, which follow some basic syntax rules - Gherkin. Along with the features, we give *Cucumber* a set of **step definitions**, which map the business-readable language of each step into code to carry out the action which is being described by the step. Usually in a test suite, the step definition itself will probably just be a few lines of code that call a library of support code, specific to the domain of our application. Sometimes that may involve using an automation library, like Selenium, to interact with the system itself. Note that the **Given, When and Then** annotations do not matter for pattern matching. If the code in the step definition executes without error, *Cucumber* proceeds to the next step in the scenario. If it gets to the end of the scenario without any of the steps failing, the scenario passes. If any of the steps in the scenario fail, however, *Cucumber* marks the scenario as having failed and moves on to the next one. As the scenarios run, *Cucumber* prints out the results showing exactly what is working and what is not. It is a very complete tool: we can write specifications in more than forty different

spoken languages, use tags to organize and group scenarios and we can easily integrate with a host of high-quality automation libraries to drive almost any kind of application.

In short, Cucumber was designed specifically to help business stakeholders get involved in writing acceptance tests. Each test case in *Cucumber* is called a scenario, and scenarios are grouped into features. Each scenario contains several steps. The business-facing parts of a *Cucumber* test suite, stored in feature files, must be written according to Gherkin syntax rules so that *Cucumber* can read them [63].

The cucumber process can be summarized with the scheme of figure 4.1.

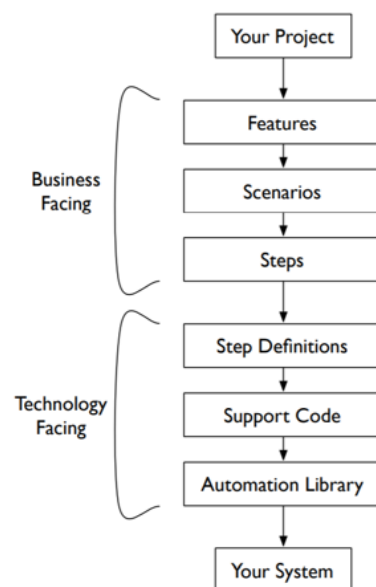


Figure 4.1: The Cucumber Process, as described in *The Cucumber For Java Book*[63]

To better understand how the framework works, a small example problem was used to perform Java tests using the IDE *IntelliJ*. The problem is a variant of the well-known *FizzBuzz* problem. This small project contains a function that takes a number: if it is multiple of 3 it returns Fizz. If it is multiple of 5 returns Buzz, as you can see in figure 4.2.

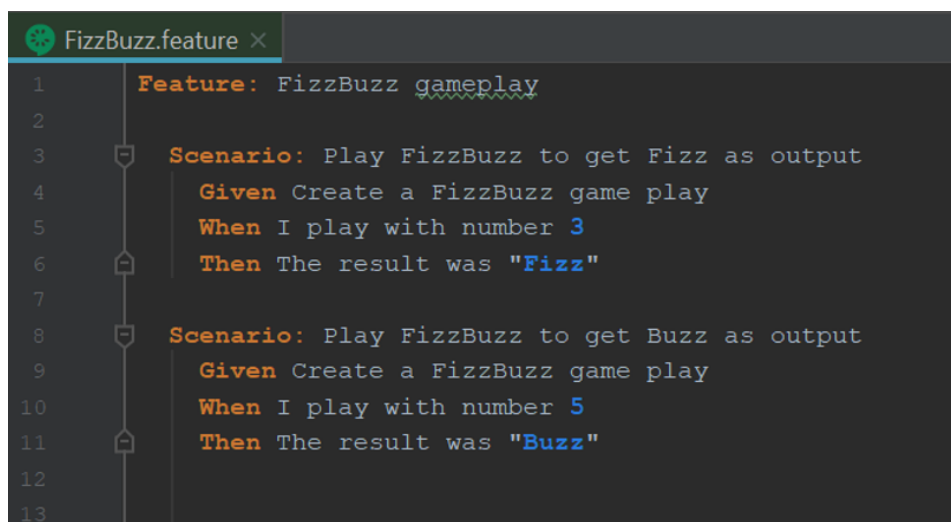
```

c FizzBuzz.java x
1  public class FizzBuzz {
2
3      public String play(int number) throws IllegalAccessException {
4          if(number == 0) throw new IllegalAccessException("Number must not be null");
5          if(number % 3 == 0) return "Fizz";
6          if(number % 5 == 0) return "Buzz";
7
8          return String.valueOf(number);
9      }
10
11 }
12
  
```

Figure 4.2: The Java FizzBuzz method under test

Two test scenarios were created: one to test "Fizz" and the other "Buzz" in a text file as shown in figure 4.3.

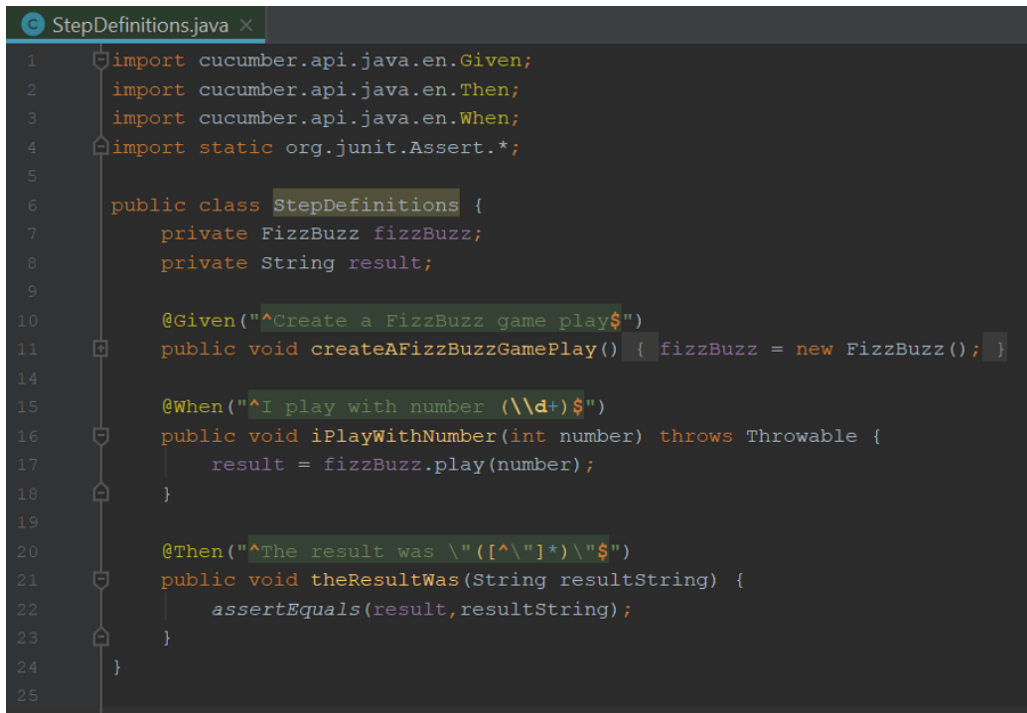
Note: For demonstration purposes we only created 2 test scenarios. In a real test scenario we could (and should) create more and also test the case if it is neither *Fizz* nor *Buzz*. We can see that Cucumber detects numbers and words in quotation marks as parameters.



```
FizzBuzz.feature x
1  Feature: FizzBuzz gameplay
2
3  Scenario: Play FizzBuzz to get Fizz as output
4    Given Create a FizzBuzz game play
5    When I play with number 3
6    Then The result was "Fizz"
7
8  Scenario: Play FizzBuzz to get Buzz as output
9    Given Create a FizzBuzz game play
10   When I play with number 5
11   Then The result was "Buzz"
12
13
```

Figure 4.3: Feature File with the plain text description of 2 Gherkin Scenarios, one for the *Fizz* case and the other for the *Buzz* case, as expected results

From this scenario descriptions it is possible to automatically generate the class with the definition of the steps, not initially implemented, but with the methods corresponding to the Gherkin sentences ready to be implemented. As you can see in the 4.4 figure Cucumber recognizes the same sentences (which have the same implementation) and reuses, without duplicating the methods.



```

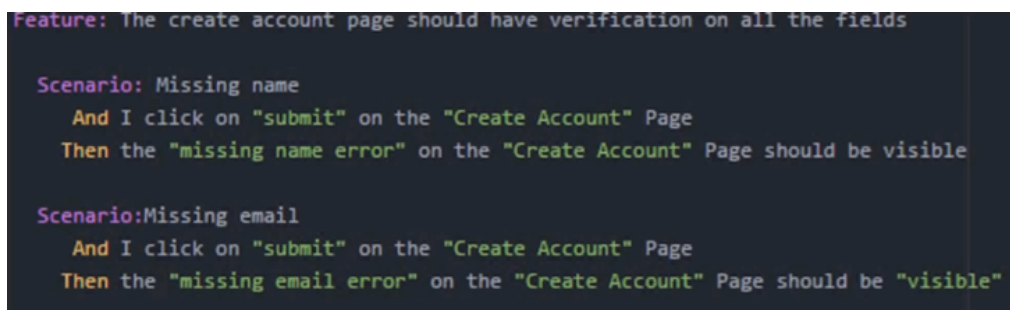
1  import cucumber.api.java.en.Given;
2  import cucumber.api.java.en.Then;
3  import cucumber.api.java.en.When;
4  import static org.junit.Assert.*;
5
6  public class StepDefinitions {
7      private FizzBuzz fizzBuzz;
8      private String result;
9
10     @Given("^Create a FizzBuzz game play$")
11     public void createAFizzBuzzGamePlay() { fizzBuzz = new FizzBuzz(); }
12
13
14
15     @When("^I play with number (\\d+)$")
16     public void iPlayWithNumber(int number) throws Throwable {
17         result = fizzBuzz.play(number);
18     }
19
20
21     @Then("^The result was \"([^\"]*)\"$")
22     public void theResultWas(String resultString) {
23         assertEquals(result, resultString);
24     }
25 }

```

Figure 4.4: Test class generated from the feature file with the step definitions already implemented

In addition, Cucumber also supports some features that can complement the description of scenarios and aid the testing process, speeding it up, including test data manipulation mechanisms:

- **Scenario Outlines:** replace each variable in the scenario step with the value from the examples table. Each row in the table is considered to be a scenario. The unique scenario will be executed for each line. In figures 4.5 and 4.6 we have an example scenario with and without the scenario outlines;



```

Feature: The create account page should have verification on all the fields

Scenario: Missing name
  And I click on "submit" on the "Create Account" Page
  Then the "missing name error" on the "Create Account" Page should be visible

Scenario: Missing email
  And I click on "submit" on the "Create Account" Page
  Then the "missing email error" on the "Create Account" Page should be "visible"

```

Figure 4.5: In this two scenarios the values (“missing name error” and “missing email error”) in the *Then* clause are introduced manually. These scenarios could be compressed into a unique scenario using a scenario outline with the examples table containing the values to replace the variable.

```

Feature: Amazon create account page should have verification on all the fields

Scenario Outline: All of the fields should display an error when not populated on form submission
  And I click on "submit" on the "Create Account" Page
  Then the "<error>" on the "Create Account" Page should be "visible"

Examples:
| error |
| missing name error |
| missing email error |

```

Figure 4.6: Using the scenario outline we compressed the 2 scenarios into one unique scenario outline that will execute as many times as there are lines in the examples table. In this particular case 2 times, one for each value of the variable “Error”

- **Steps Table:** step tables or data tables differ from scenario outlines since their data is all used together in each execution. Data tables are not looped through. Their data is all used in at once. In figure 4.5 we have an example scenario with and without the scenario outlines;

```

Feature: Amazon search results should contain all required data

Scenario Outline: Description of scenario outline
  Given I am on the site homepage
  When search for "<search_string>"
  Then each result should have the following information:
  | information | data_type |
  | title       | string    |
  | price       | currency  |
  | number of ratings | integer   |
  | rating value | string    |
  | item image  | image     |

Examples:
| search_string |
| cucumber book |
| java book     |
| cucumber     |
| java         |

```

Figure 4.7: The immediate table below the scenario specification represents a data table. This data is all used within an execution of the scenario and represents the information retrieved when we search for some word which is in the scenario outline table below

- **Background:** background is used when the scenario’s first step (or steps) are equal. It executes the step (or steps) before the other scenario steps are executed and it is abstracted from each scenario individually. Only described once but executed for all scenarios. In figures 4.8 and 4.9 we have an example with a small scenario file with and without background to group similar starting steps, respectively. Background is very similar to hooks but at the test suite level and not each scenario individually.

```
Feature: Insert detailed description here

Scenario:Examples 1
  Given I sign in as "user1"
  When I click on "My Account" on the "Home" Page
  Then I should be on the "My Account" Page

Scenario:Examples 2
  Given I sign in as "user1"
  When I click on "Log out" on the "Home" Page
  Then I should be on the "Login" Page

Scenario:Examples 3
  Given I sign in as "user1"
  When I click on "My Account" on the "Home" Page
  And I click on "Log out" on the "My Account" Page
  Then I should be on the "Login" Page
```

Figure 4.8: The first step of the three scenarios displayed is equal.

```
Feature: Insert detailed description here

Background: Insert background description here
  Given I sign in as "user1"

Scenario:Examples 1
  When I click on "My Account" on the "Home" Page
  Then I should be on the "My Account" Page

Scenario:Examples 2
  When I click on "Log out" on the "Home" Page
  Then I should be on the "Login" Page

Scenario:Examples 3
  When I click on "My Account" on the "Home" Page
  And I click on "Log out" on the "My Account" Page
  Then I should be on the "Login" Page
```

Figure 4.9: In this case it is possible to group the equal steps in a background step that will be executed for all scenarios considered.

Cucumber allows the use of reporter plugins to produce reports containing the information about which scenarios have passed or failed during an execution, about execution times, among other metrics. Some of these plugins are built-in, others have to be installed separately, like third-party plugins. These vary with the programming language we are using. Although *Cucumber* does not have an integrated sophisticated reporting mechanism as executions also depend on the environment where tests are being run, it offers ways to enable the generation of more complete visual reports more complete than simple ones presented to the user in the execution runs. Since these are very basic reports, using their output, we can build more detailed HTML reports. In figures 4.10 and 4.11 we can see examples of visual reports obtained from Cucumber's integration with *Jenkins*.

### Features Statistics

The following graphs show passing and failing statistics for features

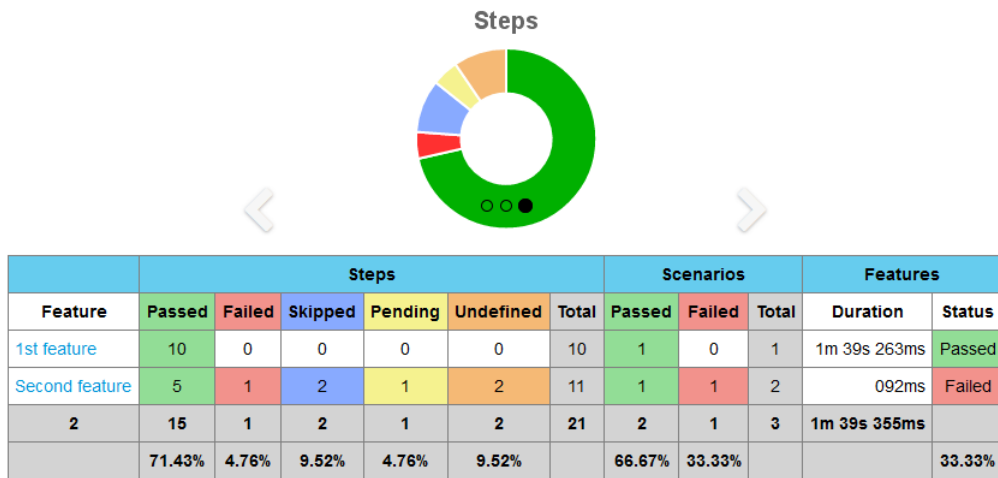


Figure 4.10: Example of a more complete visual report, obtained with the execution information generated by Cucumber, integrated with Jenkins.

### Feature Report

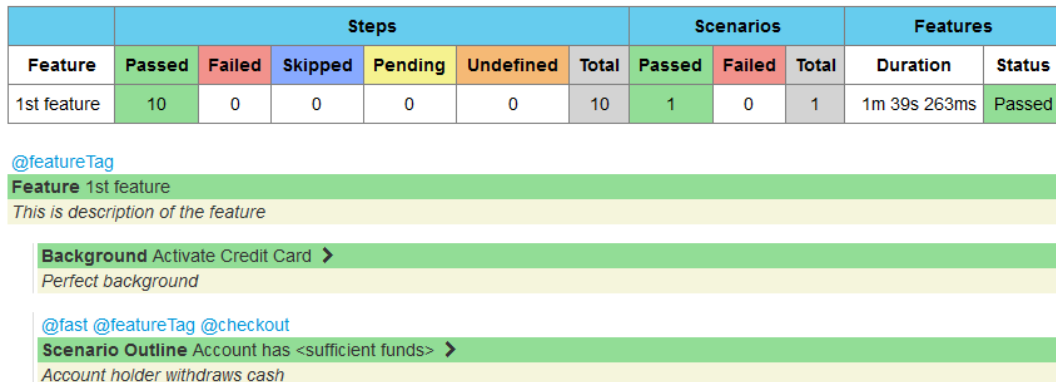


Figure 4.11: Another example of a more complete visual report, obtained with the execution information generated by Cucumber, integrated with Jenkins

### 4.2.2 SpecFlow

*SpecFlow* is *The Cucumber for .NET*. It is a “port” of Cucumber for *.NET* that also uses Gherkin syntax files but wires them up to C# code. *SpecFlow* builds on existing unit testing frameworks like *NUnit* and *MsTest*. *SpecFlow+ Runner* is a dedicated test runner for *SpecFlow* (Windows only) and integrates directly with *Visual Studio*. *SpecFlow+* replaces general purpose testing frameworks with a dedicated solution and introduces additional features, such as enhanced test execution options and execution reports. However, and despite the similarities, both frameworks are different. First of all *Cucumber* has a global namespace (Given, When and Then are interchangeable) as mentioned in section 4.2.1. In *SpecFlow* namespaces are independent (Given, When and Then are treated differently).



SpecFlow also features a greater diversity of Hooks (setup/teardown steps), with different levels: before and after steps, before and after scenarios, before and after features, before and after execution, among others. Cucumber Hooks are more limited.

As with the previous tool, we used the same small example problem (a variant of *FizzBuzz*) to better understand how the tool works in practice. The code is in figure 4.12.

```
namespace FizzBuzz
{
    2 references
    public class FizzBuzz
    {
        1 reference
        public string play(int number)
        {
            if (number == 0) return "Null number";
            if (number % 3 == 0) return "Fizz";
            if (number % 5 == 0) return "Buzz";
            else return "Neither Fizz or Buzz";
        }
    }
}
```

Figure 4.12: The C# FizzBuzz method under test

A feature file was created: one scenario to test "Fizz" and the other "Buzz" in a text file as shown in figure 4.13.

```
Feature: Play FizzBuzz
  In order to play FizzBuzz
  As regular player
  I want to receive the result "Fizz" whenever I enter an integer multiple of 3 and "Buzz" when I

Scenario: Test Fizz Output
  Given Create a FizzBuzz gameplay
  When I play with number 3
  Then The result is "Fizz"

Scenario: Test Buzz Output
  Given Create a FizzBuzz gameplay
  When I play with number 5
  Then The result is "Buzz"
```

Figure 4.13: Feature File with the plain text description of 2 Gherkin Scenarios, one for Fizz and the other for Buzz, as expected results

SpecFlow also detects parameters in the steps as we can see in the automatically generated step definitions already implemented in figure 4.14

```

namespace FizzBuzz
{
    [Binding]
    0 references
    public class PlayFizzBuzzSteps
    {
        private FizzBuzz fizzbuzz;
        private string result;

        [Given(@"Create a FizzBuzz gameplay")]
        0 references
        public void GivenCreateAFizzBuzzGameplay()
        {
            fizzbuzz = new FizzBuzz();
        }

        [When(@"I play with number (.*)")]
        0 references
        public void WhenIPlayWithNumber(int number)
        {
            result = fizzbuzz.play(number);
        }

        [Then(@"The result is ""(.*)""")]
        0 references
        public void ThenTheResultIs(string number)
    }
}

```

Figure 4.14: Test class generated from the feature file with the step definitions already implemented

As with Cucumber SpecFlow also enables detailed execution reports with information about the scenarios and their steps. These can be generated in a variety of ways and formats, either through direct SpecFlow integrations (with *NUnit* for example) or third party services.

In an OutSystems context SpecFlow may be the most interesting framework since its implementation works with C# code, which is the basis of the OutSystems language and under which the platform is built.

### 4.2.3 Framework Evaluation Proposal

Following the review of the study by Wang and Solís in section 3.2.1, we decided to create a new comparative table, with some aspects that we consider to be important to evaluate in a BDD automation framework. These were chosen on the assumption that a supporting test automation tool for the BDD process serves only for this purpose: to support the conduct of BDD by allowing test automation but not covering all Behavior-Driven Development procedures since the most important part of it (conversations and discussion around functionalities) should be taken beforehand, involving all stakeholders, and out of this automation context. As such and from the interviews conducted and the study of the main tools we culminate with the proposal of the following features that we consider appropriate to figure in this type of framework:

- **Feature and Scenario description in textual files:** description of functionalities and scenarios in a structured and simple textual language such as Gherkin, to describe product behaviors. These descriptions constitute a form of documentation

and are the basis for automation and it must be possible to write them in textual files usually called *feature files* and import them feature from outside an IDE;

- **Step Definition Generation** (Testing classes and methods): interpretation and automatic generation of test methods for each step present in the Gherkin scenarios, in a structured and standardized way, mapping the scenarios (documentation) to implementation.
- **Parameterization of Inputs in the Gherkin sentences**: the tools must be able to automatically detect parameters in the Gherkin sentences and they should create input variables for the these parameters (parameterization of testing data);
- **Scenario Outlines**: Prepare groups of testing data for each scenario;
- **Hooks (Setup/Teardown)**: Extra steps designed to prepare/remove the configurations and data necessary for a scenario or a set of scenarios;
- **Step Reuse Detection**: Detection and use of equal steps by the framework, having a centralized implementation for a step that can be used in several different places, reducing the manual work of the developers in relation to already implemented steps;
- **Automated Execution**: BDD scenarios must run automatically when executed and this execution must be fully customizable and parameterizable in relation to the scenarios to be executed;
- **Integration with Management/report Tools**: It should be possible to integrate with project management tools to import/export the feature files and to provide more traceability to the functionalities that are being developed.

Other criteria, considered less important:

- **Parameter Type Inference**: refers to inferring the type of the variables present in the description of the scenarios in order to facilitate the work of the developer and automate the process. Although this is a good feature, the most important aspect is actually finding the parameters in the sentences and creating variables to support them. If the type is immediately right, better but if not, it can still be easily changed.
- **Step Aggregation**: Aggregation of common steps within a feature to avoid redundancy and repetition of equal steps. It avoids some repetition of steps in the descriptions.

This brings us to the following evaluation model (which already fits the *BDDFramework*):

| Feature   | Cucumber | SpecFlow | BDDFramework |
|---|----------|----------|--------------|
| Feature and Scenario description in text files (Outside an IDE) | ✓        | ✓        | ✗            |
| Step Definition Generation                                      | ✓        | ✓        | ✗            |
| Automated Parameterization of Inputs in the Gherkin Sentences   | ✓        | ✓        | ✗            |
| Parameter Type Inference  | ✓        | ✓        | -            |
| Scenario Outlines   | ✓        | ✓        | *1           |
| Step Aggregation  | ✓        | ✓        | ✗            |
| Hooks (Setup/Teardown)  | ✓        | ✓        | ✓            |
| Step Reuse Detection  | ✓        | ✓        | ✗            |
| Automated Execution   | ✓        | ✓        | ✓            |
| Integration with Management/Reporting Tools                     | ✓        | ✓        | *2           |

Figure 4.15: The most used frameworks compared with the *BDDFramework* in the new proposal of evaluation model. Attributes marked with a check mark are present in the frameworks. Attributes marked with a X mark are not present in the frameworks.

**Notes about the table:** Parameter type inference is not supported in the *BDDFramework* but this attribute is a direct consequence of the previous one (we can not infer parameter types if we have no parameter detection). Regarding scenario outlines (\*1), the *BDDFramework* does not support them directly but the way the OutSystems language is built and the way it works it is possible to perform these kind of tests with relatively ease in the *BDDFramework*. There is even some documentation teaching how to perform Data-Driven Testing in OutSystems with the *BDDFramework*[59]. Integration with other tools is also partially supported (\*2). Although the *BDDFramework* lacks more complete reporting mechanisms about the detailed current state of the tests in a project, we have some ways available (with the support of the REST API *BDDFramework*[58]), to manage some information reporting how many test scenarios failed and passed in the executions (with detailed information about the failing ones).

### 4.3 Prototype Features Identification

In this section we will present the features we identified as being important to implement in the developed prototype:

- **Conversion from plain textual feature file specifications using Gherkin to Out-Systems' logic testing code:** this will make it easier for both business and developers to get involved in the process because the scenario description phase can be done outside Service Studio, and text specifications are mapped directly into test code by the framework, leaving just the test logic implementation to be done by the developer in the test implementation phase, automating the process and facilitating his work;
- **Allow the automated generation of test screens and scenarios in a structured manner and following the recommended practices for test organization:** individual scenarios are placed within *Web Blocks* (to allow reuse and association of variables/parameters) and all the scenario blocks that test the same feature are placed inside the same Web Screen . Like this, the screens are divided by features, each of which will contain the scenarios that implement that feature. This allows developers to follow a test design standard that previously had to be done manually, with risk of test architecture disorganization;
- **The product to be developed should be able to recognize and reuse equal steps automatically:** this will allow reusing implementations and saving the developer some time, especially when a lot of test code is already implemented and many of the upcoming steps are repeated. In the BDD Framework, as it is at this moment, this is not supported and developers need to manually identify identical steps and reuse them by hand;
- **Parameterization of the implementation from placeholders in the Gherkin steps:** currently in the *BDDFramework* to do this it is necessary developer involvement. In other existing tools variables can be parameterized immediately in the feature file description. For instance, we may have a sentence that checks whether a book exists in a database. The book title can be parameterized in the scenario description (e.g. "Given There is a book whose title is <title>"). This, once again, can encourage the participation of business people because they can specify explicit variables in a textual way ("Given There is a book whose title is *Don Quixote*", for example, and the implementations validates if the book with the name Don Quixote exists in the database) and this also enables a more powerful reuse of steps;
- **Integration with reporting and project management tools** (like *Jira* [5]) for project management and issue tracking. It will allow to list all the Gherkin scenarios implemented in the context of a user story. Business people would be able monitor the

implementation of features through this software and check when a user story is implemented, validating the scenarios to accept the feature or not. This integration is already possible in frameworks like Cucumber[18] with extensions like Xray[76] and was pointed out as something very valuable by the interviewee that spoke to us about his experience of practicing BDD in an OutSystems project.

## 4.4 Prototype Alternatives

After identifying the prototype desired features, we needed to analyze how we would implement those, the pros and cons of each alternative, and immediately came up with 2 prototype options:

- **Alternative 1: Start from the existing *BDDFramework* and build a new component around it.** It will use the features already provided by the *BDDFramework*, such as the Gherkin scenario templates and all the blocks for the description and implementation of BDD steps, as well as the *BDDFramework AssertSteps* library and information about test execution. With this alternative we would still be in the Low Code OutSystems domain and would have already implemented the basis for the description of scenarios as it already exists in the *BDDFramework*. The challenge here would be more focused on automating the screen test logic generation (step definitions), as well as reusing actions, all from feature textual files. The strengths of this alternative would be: the previously mentioned fact that we remain in the Low Code domain, without having to leave it to perform BDD testing. This can be a crucial aspect, as it would allow developers to stay in the development environment (the test specification and implementation would be very close to the code under test) and we would use the already known *BDDFramework* that would be leveraged by the component but will continue to exist on its own, just like before. It would not be replaced. The learning curve would therefore be supposedly shorter for developers who already know OutSystems and the *BDDFramework* and also by allowing scenarios to be described in text files that automatically populate the scenario templates (before we had to drag the steps, type in the descriptions and place the blocks in their corresponding placeholder within Service Studio). This would significantly simplify the entire process. Anyone can easily use a simple text file, including business people, who previously had to install Service Studio to participate in the scenario description process. Moreover this tool would be unique and developed specifically for the language. We can find a schema that represents this solution in figure 4.16 and the test eSpace generated general structure and organization inside Service Studio in figure 4.17.

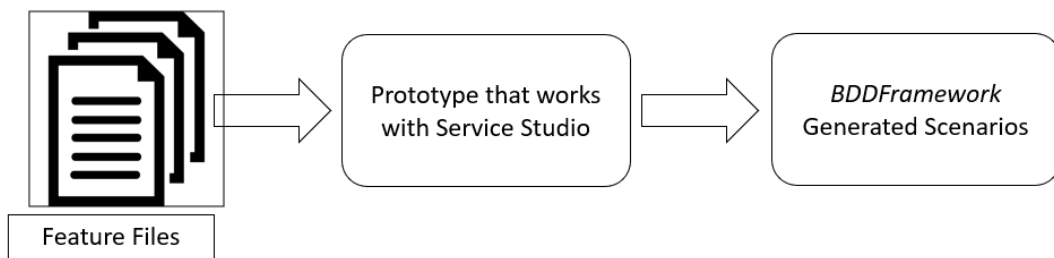


Figure 4.16: Automated generation of the test logic and filling of the BDDFramework scenarios and steps process, from the external feature files by the prototype component.

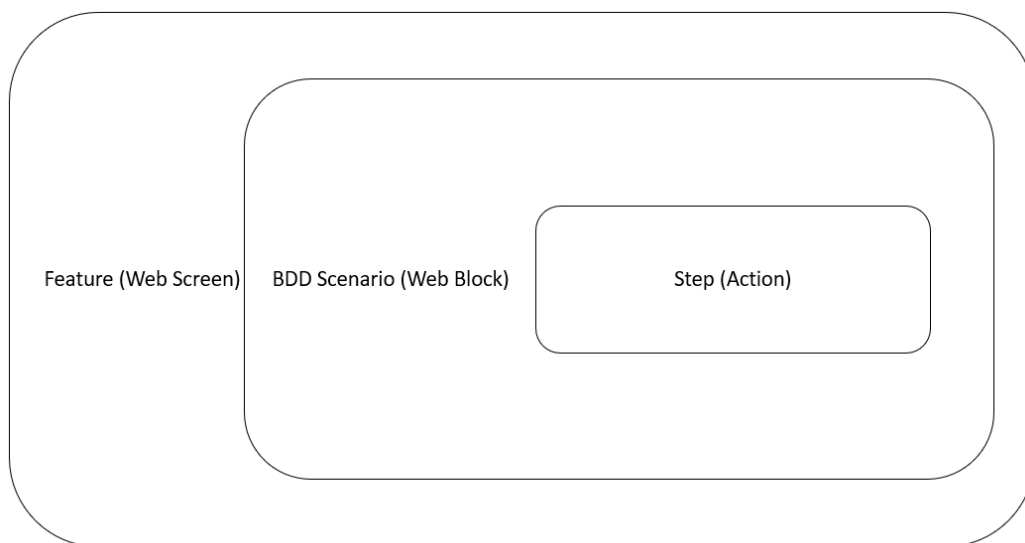


Figure 4.17: *eSpace* organization inside Service Studio.

- Alternative 2: Integrate SpecFlow with OutSystems.** The idea is to take advantage of an already developed engine and apply it to perform BDD tests. This approach involves the development of a component that enables the integration between scenarios interpreted by SpecFlow and their implementation in OutSystems logic. SpecFlow engine already supports the automated generation of test classes and step definitions for the corresponding feature files and scenarios, respectively. It also supports the parameterization of inputs with type inference as well as various mechanisms of step aggregation and data manipulation for testing. SpecFlow also supports step reuse for equal steps and it supports integration with other tools (although like in Alternative 1 we will need it anyway for an OutSystems domain). It also can be executed from the terminal without any IDE to generate the step definitions and return the test results (with the *generateall*<sup>3</sup> command). To do this,

<sup>3</sup>The *generateall* command can be used to re-generate outdated unit test classes based on out feature files. This can be useful when upgrading to a newer SpecFlow version, or if feature files are modified outside of *Visual Studio*. This does not apply to the newest version, SpecFlow 3.

the idea would be to have SpecFlow C# classes invoking actions of an OutSystems eSpace directly (in runtime). The SpecFlow C# code would just be wrappers for actions to be invoked in an eSpace, thus the step implementation would be done in OutSystems code. To achieve this, we need to built: (1) a code generation mechanism (close to Alternative 1) just to create an eSpace with Server Actions where the step implementation would be done; (2) a C# library that would allow us to easily call the eSpace server actions in runtime; (3) a code generation application that would populate the SpecFlow generated C# classes (with the step implementations) with the appropriate calls to the library in item 2. In short, the idea would be to take advantage of the SpecFlow features as they are, to be able to generate the OutSystems project with the test code. The main advantage of this approach is that the main engine is already built and quite complete with lots of features, but also the fact that the SpecFlow community is large, and it allows integration with other tools as well as command line execution. We would have found to be a way of generating output between SpecFlow, that deals with the feature files and scenarios, and Service Studio, where the test code is implemented. The main challenges would be: beyond the crossover between both platforms, the already mentioned code generation mechanism (which we have already found applicable in this context) to insert code invokers to OutSystems actions in the generated .cs files with the step definitions and a library to handle those files with the inserted parameters and invoker functions to allow the automated generation of eSpaces, screens, actions and parameters in an OutSystems domain. Finally, a way to generate the test outputs across platforms. Figure 4.18 represents this approach.

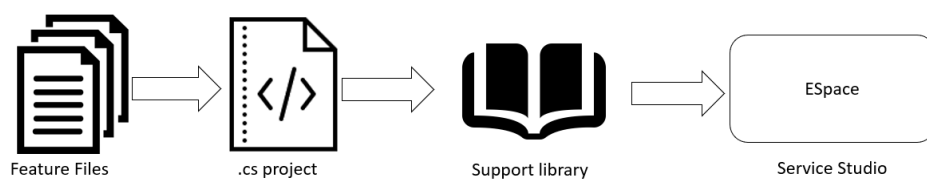


Figure 4.18: Schema that represents Hypothesis 2.

## 4.5 Decision making and strategy adopted

Then, after carefully analyzing the implications and feasibility of each approach, we made another round of interviews, this time only to 2 people, both previously interviewed. One of them works with the *BDDFramework* regularly and is very used to this tool, either to create or debug BDD tests. The other interviewee is the external contact who uses SpecFlow, successfully, to perform BDD testing on OutSystems projects. However, he uses Selenium to do UI layer testing (which is not what we intend). We want a framework that works for other types of testing (including functional testing), without having to



prepare our code for UI testing, which has its problems as we could see in section 2.17. We considered that the contribution of these two people could be important in order to make a decision, so, we presented both hypotheses to them and discussed in more detail their testing processes, to better understand their daily **testing and debugging experience**, something that was not possible to realize in detail in the first interview, since it was not its purpose back then.

With the interview that was done to the *BDDFramework* user and the research that had been done beforehand, it was possible to realize that **the developer experience for those who use the BDDFramework is good: it is easy to establish a connection between the sentences that make up the Gherkin scenarios** and its corresponding implementation in OutSystems' logic, since there is a direct mapping between them, thus making it **easy to navigate through feature files and quickly access the code as they are both in the same place (Service Studio)**. This was precisely one of the most important aspects to understand from the external contact interview: how important is the navigation from the Gherkin scenarios to the step definitions code, when debugging the failing scenarios using SpecFlow with OutSystems? We are afraid that this approach we are proposing might **break this experience**, since the Gherkin scenarios are in Visual Studio, outside the OutSystems development tool (Service Studio).

According to this interviewee, the ability to immediately identify what is failing in a test, from the report generated by SpecFlow and where the failure occurred depends on the **granularity of the Gherkin scenarios** and the **experience of the developer**. Usually, if they are very generic sentences they become harder to analyze. However, if the developer is experienced and knowledgeable about the code he usually debug it directly. Many of the bugs and problems he faces also relate to the UI layer (objects not found) and the very nature of the tests that are performed using Selenium. The overload of the test tool itself also plays a relevant role in this domain and can be a determining factor in its execution. Most problems are due to changes that are being made in development that are not yet reflected in testing.

Regarding the debugging process itself, the first step the interviewee takes is to check whether or not the failing object is found by SpecFlow (most recurring error) and if the error does not come from it, a breakpoint <sup>4</sup> is placed in the implementation of the step that is failing (isolating the code, commenting the unnecessary lines). **If the error is not found the developer will see if the step in question is dependent on any previous steps by navigating through the code**. Data is passed and shared between steps through global variables, not via the scenario we are running.

We presented the possibility of developing a tool that uses SpecFlow for OutSystems in this dissertation and that is not designed specifically for UI testing using Selenium, but instead also allows functional/logical testing under the UI layer of applications. This

---

<sup>4</sup>A *breakpoint* is an intentional stopping or pausing place in a program, put in place for debugging purposes

component should be able to read feature files and transform them straight into test specifications in the OutSystems language (Alternative 2). As it is currently the case with SpecFlow, the developer will only have to worry about implementing the test code for the new steps, and the application itself would allow the reuse of equal steps existing steps. The respondent noted that this could be interesting as it would take the load off the UI and could make the testing process faster and more efficient, both in development and in execution. The interviewee also pointed out that this would **broaden the test spectrum and make it easier to test the logic of applications**, although he also considered that allowing testing under the UI layer of applications might not be as interesting from a business point of view as we thought. All of this could also be achieved with alternative 1: extending *BDDFramework* to use a mechanism similar to SpecFlow, capable of importing Gherkin scenarios described in text files outside of Service Studio and filling the *BDDFramework* templates with them, as well as allowing automatic generation of step definitions in OutSystems' logic. Regarding the two prototype approaches presented, both respondents agreed that if SpecFlow is used to read the feature files and used to invoke the OutSystems' logic, it may be **difficult and impractical to navigate from one tool to another** (SpecFlow kept the feature files and Service Studio the OutSystems' code), in **separate domains**. There could be some **lack of traceability**. This is the problem with this approach: it can **degrade the development and troubleshooting experience**. On the other hand with the other approach of extending *BDDFramework* this problem no longer exists as this experience of browsing through feature files and direct mapping to code is preserved and can be done directly within Service Studio, in the development environment. With SpecFlow we have an engine that works outside of OutSystems. **If we do not have a Gherkin editor in service studio it becomes more difficult to relate scenarios to code and hence, the debugging experience.**

As we have already mentioned, the developer experience of those who use the *BDDFramework* is good (we see things happening). Since SpecFlow works outside Service Studio, making the bridge between the description of scenarios and their implementation can be difficult, which complicates the development and debugging processes for less experienced developers and people looking at test they did not write. With the *BDDFramework* it is all in one single place without degrading the developer experience as we have Gherkin scenarios and their implementation side by side in Service Studio, making code and testing progress **easier to track and maintain** and this is the **main reason for choosing Alternative 1** over Alternative 2. In addition, this solution can possibly establish itself as a test standard made specifically for OutSystems and using the existing *BDDFramework* that many users already know, without changing it or changing the OutSystems product itself.

## PROTOTYPE IMPLEMENTATION

In this chapter we will describe the whole implementation process of the Prototype produced during this dissertation. In a first step, we will start by describing its features and architecture, and then we will explain all the procedures, giving an overview of the implementation and how it was done, present the main algorithms used, justifying our choices and describing the difficulties we faced along the implementation phase.

### 5.1 Prototype Description

The software component produced during this dissertation was **implemented incrementally**, in small development sprints, and it works as a **Service Studio command** that automatically generates the step definition screens and actions from text files containing the BDD scenarios, just like any other well-know BDD tool, but in OutSystems. This component uses the existing *BDDFramework*, with the difference that, through the prototype, its templates are now **automatically** filled, based on external feature files that contain the scenario descriptions, organized by features, as you can see in figure 5.1. This will allow business people to participate in the scenario description process more easily and actively, and also make it easier for developers, who can now integrate new scenarios in the test suites, which came from project management tools, by simply running a command to enter them into Service Studio.

```

1 Feature Withdraw cash from bank account
2
3 Scenario When the user withdraws a certain money quantity (20) from his account the balance should be correctly updated
4 Given I have a valid card
5 And I have "30" euros in the account
6 When I click in option to withdraw "20" euros
7 Then The account balance is correctly updated
8
9 Scenario When the user tries to withdraws a certain money quantity (50) from his account he should receive an error message if he does not have enough money
10 Given I have a valid card
11 And I have "40" euros in the account
12 When I click in option to withdraw "50" euros
13 Then A "not enough money" error message should be displayed
14
15
16 Feature Make a deposit in the bank account
17
18 Scenario When the user deposits a certain money quantity (30) in his account the balance should be correctly updated
19 Given I have a valid card
20 When I deposit "30" euros
21 Then The account balance is correctly updated
22

```

Figure 5.1: Example of a feature file containing 2 features and 3 scenarios. The first feature contains two test scenarios and the second only one. The component parses the file and the highlighted words identify special keywords where new Features, Scenarios and Gherkin steps begin.

The description placed in front of the *Feature* keyword will serve to name a Web Screen that will contain scenarios testing that feature. The scenario description, like all descriptions placed in front of the Gherkin syntax keywords (**Given**, **When**, **Then** and **And**) will be used to fill the *BDDScenarios* and will name the implementation actions.

In addition to enabling this automation and consequent acceleration in the process of importing and populating the scenarios, this component also automatically generates:

- **Test Screens:** For **each feature** under test described in the feature file, a **Web Screen** is created. It will contain inside **all scenarios corresponding to that same functionality**. At the bottom of each feature screen is placed a **Final Result** block, which reports how many scenarios are failing from the current screen;
- **Scenario Blocks:** each *BDDScenario* is properly filled with the scenario description and the Gherkin steps from the feature file, and it is placed inside a **Web Block** (reusable) within the Web Screen of the corresponding feature;
- **OutSystems actions** that will contain the implementation of the Gherkin steps: for **each sentence** in the Gherkin text, a **Screen Action** that is associated with it is created, and it calls the corresponding **Server Action** (with the same name) that will contain the logic of the implementation for that respective sentence.

The structure and organization of the test eSpace (which we can see in figures 5.2 and 5.3) was thought according to some recommended practices given by OutSystems' quality experts, to standardize the way BDD tests are conceived and designed, as we can remind in section 2.18.1.

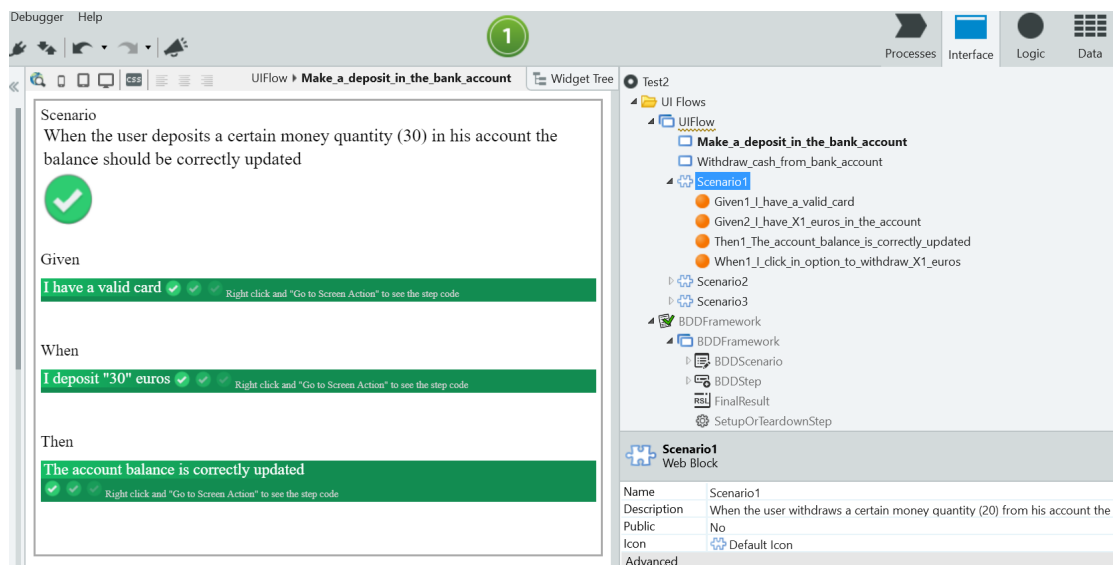


Figure 5.2: The following test module is obtained when we execute the BDD command using the previously presented feature file example.

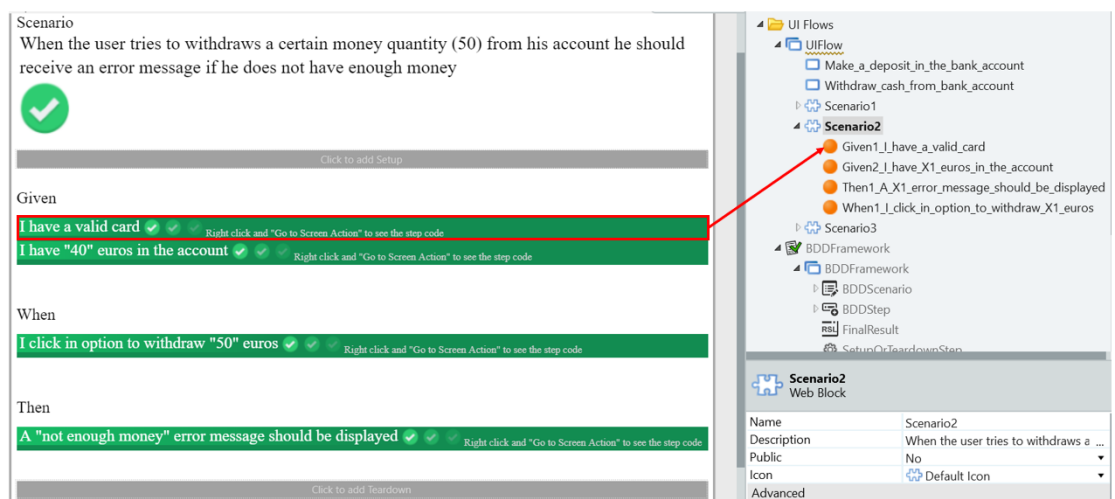


Figure 5.3: Each sentence in the Gherkin scenario is connected with a screen action containing its implementation.

We will now present some fundamental features considered for the developed component, related to test implementation efficiency:

- **Step reuse:** As the complexity of applications grows, it is expected an **increasing number of test scenarios** and a resultant **redundancy of steps**. As such, it is critical that our component can **detect equal steps** and be able to **reuse their implementation code**, which must be implemented in a single place, avoiding redundancy of actions and making the developer work as facilitated as possible, preferably without having to worry about this reuse process. In the past, with the *BDDFramework*, step reuse had to be done manually by developers. We found a way to automate this:

each Screen Action (figure 5.4), instead of directly containing the corresponding step implementation code, will instead call a (centralized) Server Action with the same name (figure 5.5). If the Server Action does not already exist in the system (meaning that the step is new) our component creates a new one. Otherwise, the existing one is called inside the Screen Action connected to the step. Therefore, the Server Actions will contain the step implementation logic and so, we guarantee that once the sentence logic is defined the first time, from now on all the equal steps that come up will be already implemented in a single centralized action. For people writing scenarios to take advantage of this reuse they should write equal steps with exactly the same description.

- **Parameter Detection and Type Inference:** The component detects what is enclosed in **quotation marks** as a parameter, and then creates corresponding **input parameters** in the server actions that implement those steps (figures 5.6 and 5.7). Note: For reuse purposes, the sentences: *The user removes "10"bananas from the cart* and *The user removes "15"bananas from the cart* are equal since the parameter is ignored and is not taken into account when comparing the sentences. The parameter type is also automatically inferred by the application.

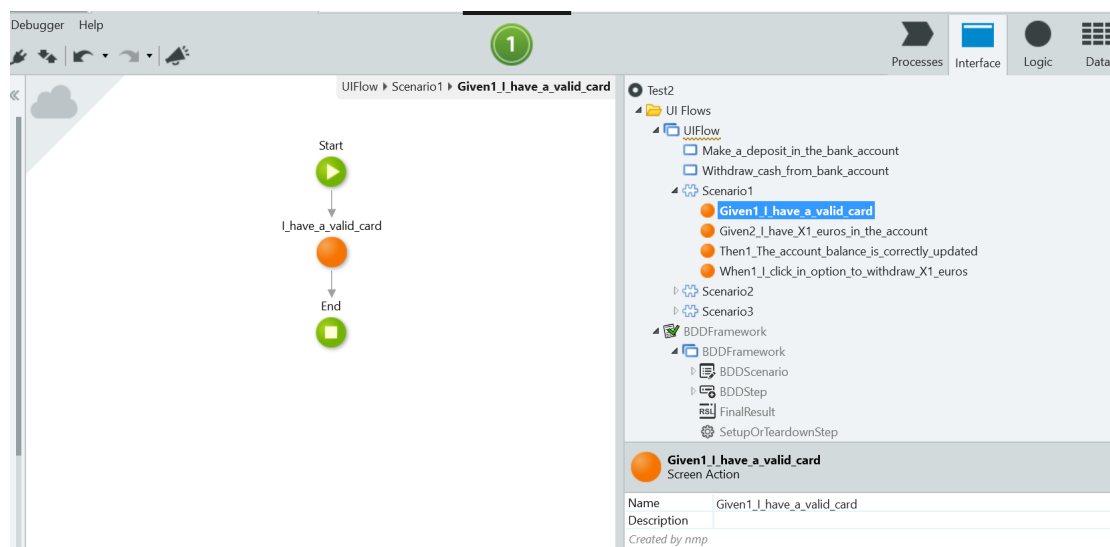


Figure 5.4: The Screen Action associated with a Gherkin step calls a Server Action (centralized) with the same name containing its implementation. There is *one Screen action per Gherkin Step*.



Figure 5.5: The Server Actions hold the centralized implementations of the BDD steps. There is only one Server Action *per different Gherkin step*, avoiding action redundancy. Every Gherkin step “I have a valid card” will call the same Server Action, since their implementation is the same (equal steps). As we can see, there are 11 Gherkin steps in the example feature file (figure 5.1) but only 6 Server Actions in the generated eSpace, since some of the steps are the same. In the second and third scenarios, only one of the steps is new.

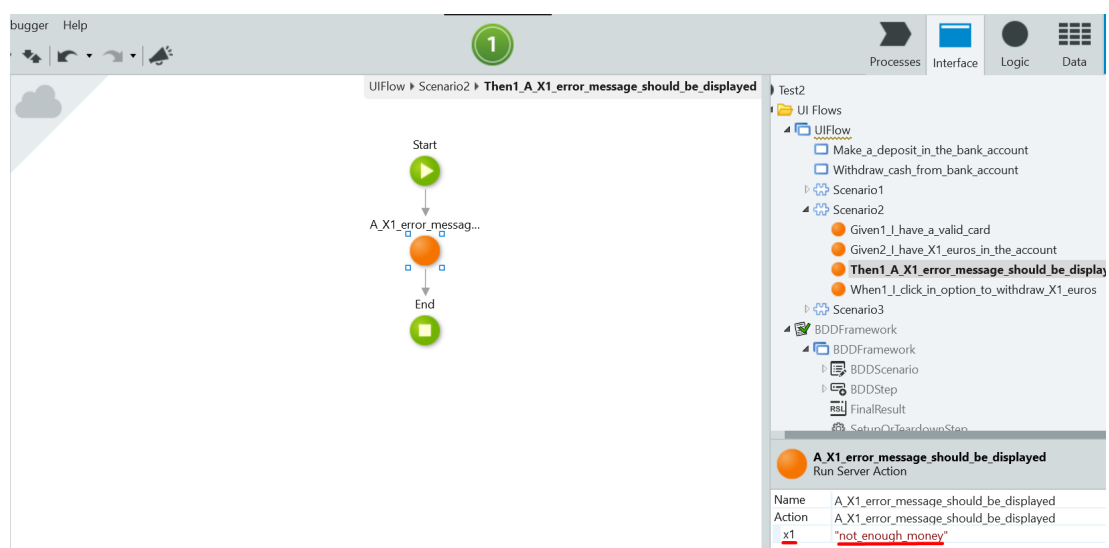


Figure 5.6: The textual parameter is replaced by a variable in the sentence. It is ignored when we compare sentences to check for equality. Its value is passed as an input parameter for the server action that contains the corresponding step implementation.

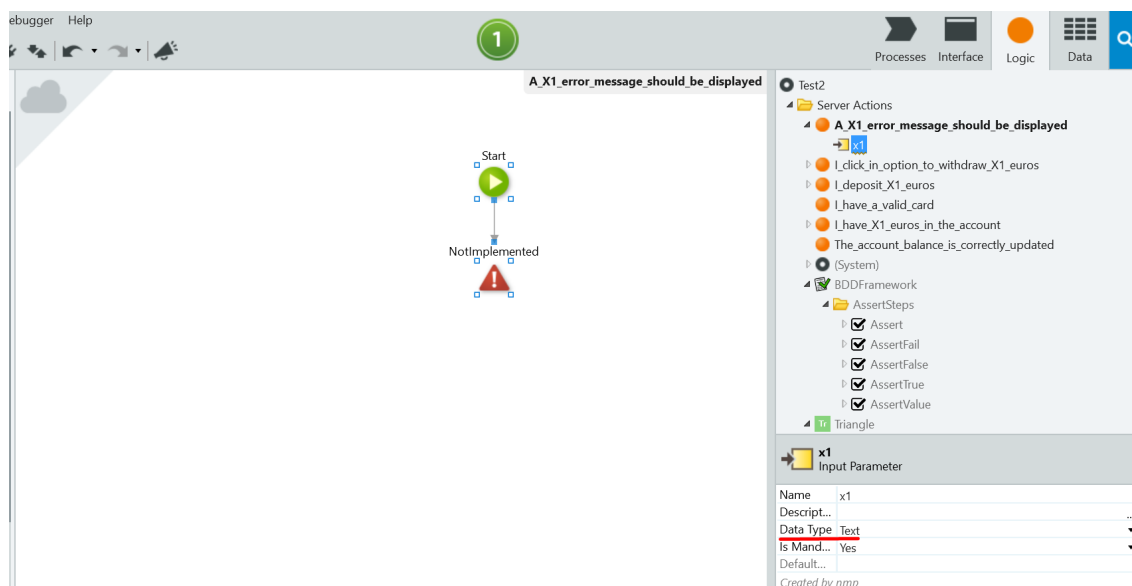


Figure 5.7: The Server Action (not yet implemented) receives the parameter as an input and it automatically infers its type, in this case a text containing the error.

As expected, the Server actions that implement the Gherkin steps are not initially defined (except for already implemented sentences, where the implementation is automatically reused by the application as we previously saw) and this work is later up to the developer. As such, we **automatically generate an OutSystems exception** (*NotImplementedException*) which is the only logic initially present in the Server Actions, to demonstrate that they have not yet been implemented, as we can see in figure 5.8. These exceptions should then be replaced by the test logic. It is recommended that developers implement applications with this type of testing in mind and taking in consideration that the core logic of applications must be implemented in Server Actions so they can be exported and used across the test projects.



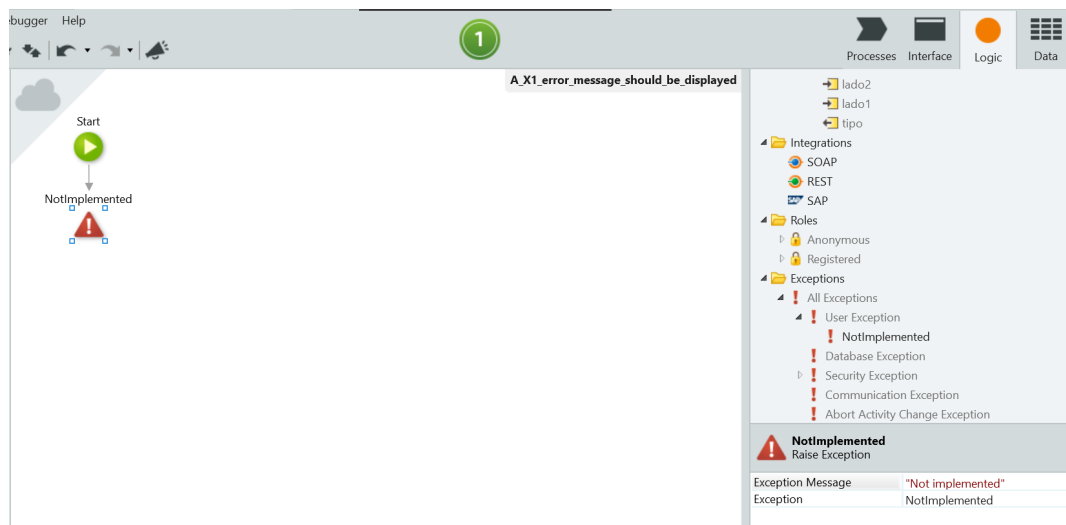


Figure 5.8: The *NotImplementedException* is defined for the generated Server Actions, when those are created.

Regarding the operation of the Prototype, the command is executed by **right-clicking the UI Flow** (inside Service Studio's Interface tab) and clicking the **"Generate BDD Scenarios"** option. The test module must have the *BDDFramework* dependencies already imported. To do this, it is necessary to first download the component in its OutSystems forge page <sup>1</sup> and set up its dependencies (in the *Manage dependencies* menu) on the module where tests will be made, within Service Studio. In order to see the results of the test executions we need to **publish the module** and open it in the browser, where we can see each scenario screen with the corresponding scenarios individually displayed along with the *BDDFinalResult* block at the bottom of each screen (figure 5.10).

<sup>1</sup><https://www.outsystems.com/forge/component-overview/1201/bddframework>

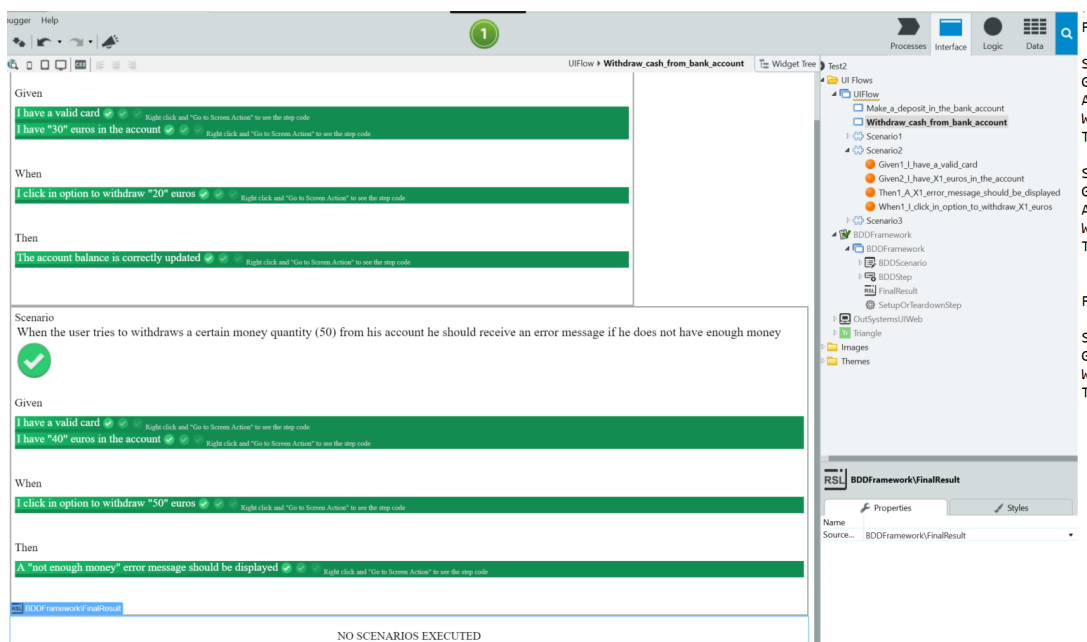


Figure 5.9: The *BDDFinalResult* block shows how many scenarios have failed during the execution.

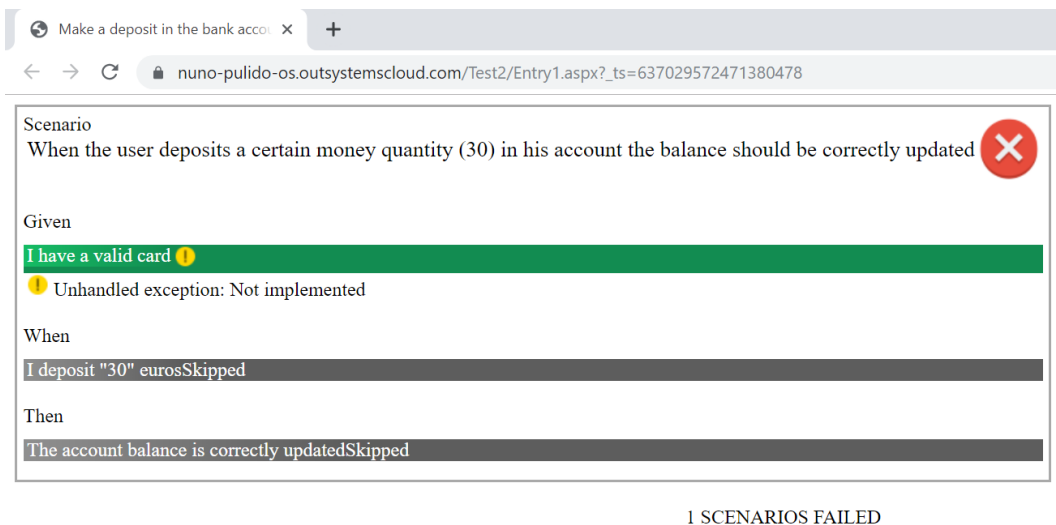


Figure 5.10: The result of publishing and displaying in the browser the scenario results from the *Withdraw cash from bank account* screen. The (single) scenario was not implemented and as expected the unique scenario fails right on the first step, where an exception is raised and the other steps are skipped.

In order to **edit**, **add** or **remove** scenarios we must make the desired changes **directly on the feature file**. The file must be saved for the changes to take effect, then we need to simply re-run the command again in Service Studio. The command can be executed successive times and nothing that was already in *eSpace* will be duplicated. Only scenarios that were removed from the feature file, before the execution, will be deleted, as well

as any screens and actions that were left behind that are no longer needed (for example screens of features from which all test scenarios were deleted and they are now empty, or actions that are unused since the scenarios containing their sentences got deleted). New scenarios that were added into the file will also be introduced into the project. Regarding the edition of scenarios, developers have complete freedom to edit the test implementation code as many times as they need. Nothing that is modified in the implementation will be lost and actions will still be properly linked to the respective steps. It is also possible for the developer to add more logic to the blocks by adding more actions, whether these are BDDFramework setup or teardown actions or other types of actions within Service Studio, and these will be saved successfully as scenarios are kept and maintained in the *eSpace* linked to the corresponding Web Block even when we rerun the command (if they are kept in the feature file by the user). The idea was to make the command as efficient as possible and so, we needed to avoid regenerating the components. When the user changes a Gherkin sentence in the feature file, the idea would be to change it within Service Studio scenarios but without touching the implementation since with the current implementation a new action is created (because the sentence is different) and as such, the scenario is considered different. However, this presents a minor problem as only this sentence will be reinitialized and not the entire scenario. we have the infrastructure for this but we are not currently doing it: if the scenario description remains the same (which means the scenario is the same) and a sentence changes then we just **change the text of it**, regardless of whether we keep the old one and create a new one with the same code. In turn, the scenario description (the identifier) must be unique as we assume that two scenarios with the same description are equal and such duplication is not allowed. So, if we edit the description we are creating another scenario that is of course the same if the Gherkin steps remain untouched, and due to the reuse mechanisms we have it will be reused in its entirety if the actions are not deleted.

## 5.2 Implementation Analysis

### 5.2.1 Architecture Overview

The Prototype was developed in .NET and integrated with the platform code, as a Service Studio command component, without changing the platform structure and organization. Figure 5.11 represents the class diagram of the component. This was generated using the Class Designer of Visual Studio 2019, the editor in which this project was developed.

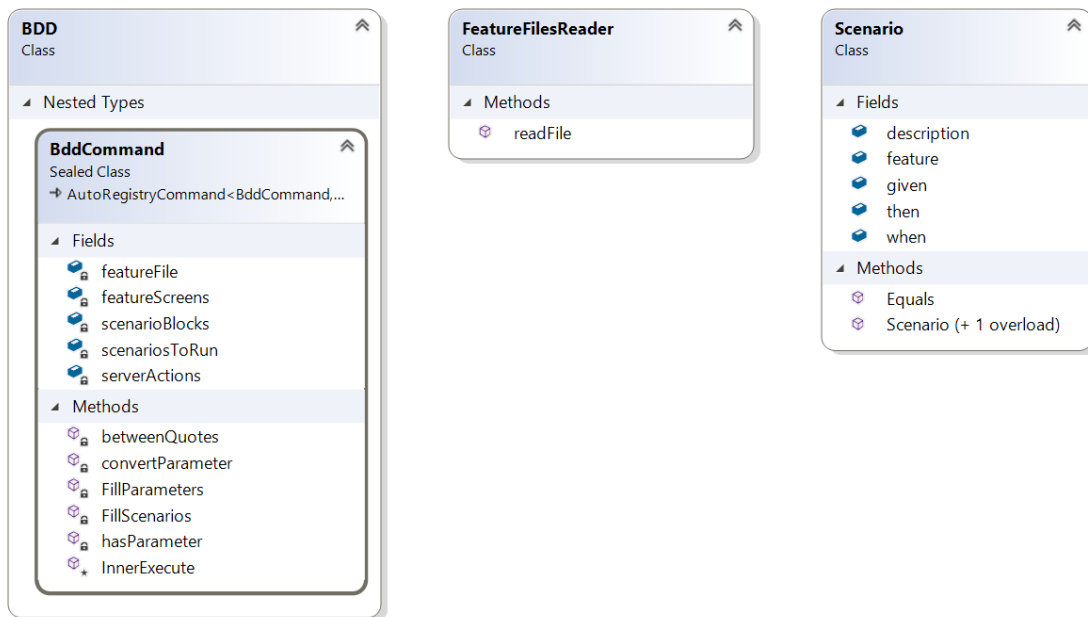


Figure 5.11: The Class Diagram representing the 3 classes which compose the Prototype.

Now we will give some more insight about the classes that make up our solution:

- **Scenario Class:** As its name suggests, this class represents a BDD scenario which is composed of a description, a set of **given**, **when** and **then** classes, and tests a specific feature. This is a very simple class and it is the basis for the creation of scenarios that come from the feature file.
- **FeatureFilesReader Class:** this class reads the feature file. Basically, it works as a parser of files written in Gherkin. The main method of the class (`readFile`) ensures the logic responsible for identifying in the sentences the Gherkin keywords (**Given**, **When**, **Then**) as well as the keywords that indicate the beginning of a Scenario or a new feature (**Scenario** and **Feature**, respectively). The sentences (or textual descriptions) that are placed in front of the keywords are saved in the Scenario objects. Each scenario is then placed in a dictionary representing the feature file, where the Key is the feature name and the value is the list of scenarios that implement the feature. This class iterates over the entire text file until it completely populates the dictionary that will be used to read the scenarios when we execute the command.
- **BDD Class:** This is the main class of our program. This is where most of the computation related to the entire screen generation process is centered, and it is where most of the data structures responsible for storing screens, nodes, and actions are located. It consists of a major method (`InnerExecute`) that contains the code responsible for the command execution in Service Studio and a set of auxiliary methods responsible for filling the list of scenarios, data structures and the eSpace objects to be considered in the next execution, already filtered according to what

was present in eSpace and read at the beginning of the method. The logic containing parameter reading and type inference is also present in this class as well as the entire process of programmatic search and completion of the *BDDFramework* templates in Service Studio.

### 5.2.2 Development process

As it was already mentioned in previous sections, the prototype was produced incrementally, in **developmental sprints of approximately 3 weeks**. The order between tasks was established by value of features. In the first phase we focused on understanding the programmatic generation of test screens and blocks, which we considered to be the most valuable feature initially. For this we used dummy scenarios, statically placed in the code, since we did not have the file reader that was developed in a second phase. At this moment, the scenarios were read from the text file and it was already possible to create the test Web Blocks and Web Screens containing the *BDDFramework* templates, with the corresponding Screen Actions that implement the Gherkin steps. Then, we developed the centralized actions that came at the top of our priorities: after being able to generate the tests from a scenario file, our priority became the reuse of equal steps. This was achieved by introducing Server Actions to store the steps' implementation. The next step was the parameterization and the development of mechanisms to allow detecting the parameters present in the Gherkin sentences. Finally, our focus was on reusing nodes between executions: at an early stage we deleted all the blocks and screens and regenerated each time the command was executed, as this simplified the reading logic of eSpaces (because we did not need to worry about which nodes were already present in there or what would be deleted, as we just read the feature file all over again). Server Actions were the only ones that could be saved, allowing us to reuse their implementation code. The goal now was **not to delete nodes and screens that were reused, both between runs and when Service Studio was closed and all variable data in memory was lost**. This was achieved with some logic and test eSpace interpretation mechanisms. Finally, we left to the end some details like some **nomenclature** associated with the actions and scenarios (to be able to edit the name of the directly in the feature file), **the visualization of the execution results** and some editing aspects of the scenarios, which are considered secondary aspects or that the idea came up after the development phase.

### 5.2.3 Implementation Analysis

In this section we will analyze the execution flow that the test eSpace gets through when the command is executed, analyzing in detail the crucial phases of the execution and all the main mechanisms and algorithms used for the solution implementation, using pseudo-code to demonstrate our choices and justify the correctness of our decisions.

Figure 5.12 represents in a small scheme with main stages of the execution flow happening when the command is executed.

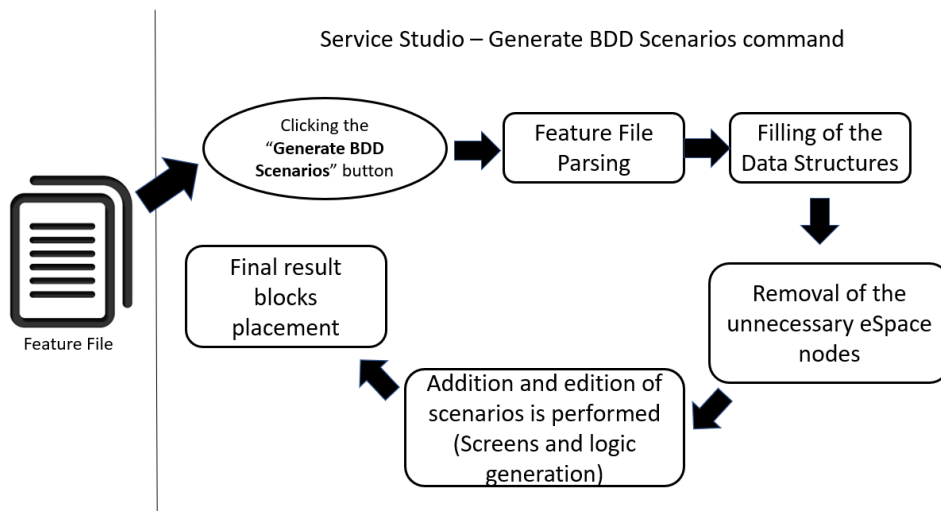


Figure 5.12: The **Generate BDD Scenarios** command execution flow.

When the command is executed, the first step performed in the process is the parsing of the text file containing the features. The `readFile` method of the `FeatureFilesReader` class is called, which works as presented in algorithm 1.

The idea behind this algorithm is to read the text file, line by line, and build the Scenario objects from it. By finding the Gherkin keywords (*Given*, *When*, *Then* and *And*), along with the *Feature* and *Scenario* keywords, which represent a new feature and a new Scenario, respectively, we fill in their descriptions with the rest of the sentence (substring) which is in front of the keyword. Of course we have to keep in mind a few details: If we find the keyword *Feature* (which marks the start of a new feature) and this word was not found for the first time (which means this is at least the second feature found) in the file, then we can add the scenario that was being built to that feature’s list of scenarios and put the list at the respective dictionary position, thus closing the previous feature and restarting the current feature and its scenario list; If we find the *Scenario* keyword we add the scenario that was being built to the scenario list and start a new one with the current description; If we find the keyword *And* we have to be aware of the current type, hence the need to store it in a variable, which is updated when we arrive at a new type (**Given**, **When** or **Then**). Finally, after the loop it is necessary to close and save the last scenario because we will not go back to the cycle and at the end of reading the file we will have filled the dictionary representing the feature file, organized by features and respective scenarios, ready to use by the component within Service Studio.

After completing the dictionary that represents the user’s feature file, some logic that will populate the data structures that store the screens, blocks, and actions is executed. This is necessary since, **once Service Studio is restarted, these structures are also reset**, but **nodes and flows can be stored in the test module** and as such it is necessary to fill these structures before generating the new ones, to avoid duplicates. The logic behind this mechanism is presented in the algorithm 2.

**Algorithm 1** Feature File Reading

---

```

1: procedure READFILE()
2:   featureFile ← dictionary that will represent the feature file, where the key is
3:   the feature name and the value is the list of feature scenarios;
4:   currentFeature ← current feature being read;
5:   currentType ← type of the last Gherkin clause read, to use in the And clauses;
6:   currentScenario ← scenario currently being built;
7:   scenarios ← list of scenarios to add to the current feature;
8:   line ← represents a line of the file;
9:
10:  for each line in textFile do
11:    if line starts with "Feature" then
12:      if currentFeature is not null then
13:        scenarios ← scenarios ∪ currentScenario;
14:        featureFile ← featureFile ∪ (currentFeature, scenarios);
15:        currentScenario ← null;
16:      end if
17:      scenarios ← {};
18:      currentFeature ← featureName;    ▶ Where featureName represents the
    textual description in front of the word "Feature"
19:    else if line starts with "Scenario" then
20:      if currentScenario is not null then
21:        scenarios ← scenarios ∪ currentScenario;
22:      end if
23:      currentScenario ← new Scenario();
24:      currentScenario.description ← description; ▶ Where description represents
    the text in front of the word "Scenario"
25:      currentScenario.feature ← currentFeature;
26:    else if line starts with type then ▶ Where type represents a Gherkin keyword
    (Given, When, Then)
27:      currentScenario.type ← currentScenario.type ∪ sentence;    ▶ Where
    sentence represents the textual description in front of the type
28:      currentType ← type;
29:    else if line starts with "And" then
30:      if currentType is type then    ▶ Where type represents a Gherkin keyword
    (Given, When, Then)
31:        currentScenario.type ← currentScenario.type ∪ sentence;    ▶ Where
    sentence represents the textual description in front of the type
32:      end if
33:    end if
34:    if currentFeature is not null then
35:      scenarios ← scenarios ∪ currentScenario;
36:      featureFile ← featureFile ∪ (currentFeature, scenarios);
37:    end if
38:  end for
39: end procedure

```

---

**Algorithm 2** Filling Data Structures

---

```
1: procedure FILLSTRUCTURES(Flows.WebFlowtarget)
2:   scenarioBlocks ← set of Web Blocks identified by Scenario description;
3:   featureScreens ← set of Web Screens identified by Feature description;
4:   serverActions ← set of Server Action flows identified by step description;
5:
6:   for each node in Nodes.WebBlock do   ▶ Where Nodes.WebBlock is the set of Web
   Blocks in the module
7:     if node ∉ scenarioBlocks then
8:       scenarioBlocks ← scenarioBlocks ∪ (node.description, node); ▶ Where the
   description of Nodes.WebBlock nodes represents a scenario description
9:     end if
10:  end for
11:  for each node in Nodes.WebScreen do ▶ Where Nodes.WebScreen is the set of Web
   Screens in the module
12:    if node ∉ featureScreens then
13:      featureScreens ← featureScreens ∪ (node.description, node); ▶ Where the
   description of Nodes.WebScreen nodes represents a feaure name
14:    end if
15:  end for
16:  for each action in Flows.UserActions do   ▶ Where Flows.UserAction is the set of
   Server Actions in the Logic tab
17:    if action ∉ serverActions then
18:      serverActions ← serverActions ∪ (action.description, action); ▶ Where the
   description of Flows.UserAction flows represents a Gherkin step description
19:    end if
20:  end for
21: end procedure
```

---

Before processing the scenarios we got from the user at the beginning of the command execution, we first need to consider a few things: not everything that comes from the Feature File needs to be processed. Some scenarios may already be in the test module and as such they do not need to be processed again. Others can be there but they might not be exactly the same, having suffered some kind of edition in the sentences and as such we need to deal with it, replacing the old ones with the new ones (in fact editing the differences). We must also remove from the list of scenarios to execute, those that are not in the feature file. This results in a list of scenarios to process (*scenariosToRun*), which comes from what was already in it and from the dictionary we had previously obtained. This requires a dictionary to list conversion which simplifies our process because reading the feature file was easier to do in the form of a dictionary (due to the text file structure) but processing scenarios with a list is more practical and is sufficient as Scenario objects have their respective feature field, which allows us to identify it without needing the key. The pseudo-code for this part is presented in algorithm 3.

We could read the scenarios from the feature file as it came and it will still be correct because when the screens themselves are actually generated, mechanisms that avoid



**Algorithm 3** Scenarios to run preparation

---

```

1: procedure FILLSCENARIOS()
2:   scenariosToRun ← List of scenarios to use in this execution;
3:   featureFile ← set of scenarios read with the readFile method;
4:
5:   for each scenario in featureFile do
6:     if scenariosToRun is null then
7:       scenarios ← {};
8:     end if
9:     if scenario ∉ scenariosToRun then
10:      scenarioToReplace ← scenariosToRun.Where(i => i.description ==
        scenario.description);
11:      if scenarioToReplace is not null then
12:        scenariosToRun ← scenariosToRun / scenarioToReplace; ▶ We replace
        the old scenario with the new (edited) one
13:      end if
14:      scenariosToRun ← scenariosToRun ∪ scenario;
15:    end if
16:  end for
17:  scenariosToRun ← scenariosToRun / scenariosToRun.Where(i => i ∉
        featureFile);
18: end procedure

```

---

duplication are taken into account but the editing of scenarios and the overwriting of existing ones is taken into account at this stage hence the importance of it. For that we developed a Scenario comparator that does not simply compare scenario descriptions (which are the identifiers) but also the Gherkin step arrays and the main idea is to detect those cases, in which a sentence is modified, to replace only what changed (keeping the scenario skeleton).

Later, all Web Screens that are no longer needed (i.e. no longer contain scenarios) and Web Blocks with scenarios that are no longer in the feature file, do not need to be in eSpace and as such, can be removed. For that, we iterate over the eSpace Nodes and for each one we will see if it is present in the *scenariosToRun* list. If so, we mark the node as not to delete. If not, then we first delete all references to it on both screens and data structures and finally remove the node. This is done first for Web Blocks and then for Web Screens, in ascending order of screen hierarchy, as it is possible to see in algorithm 4.

The logic for deleting unused Server Actions (because the scenarios that used them were deleted) will only be taken into account at the end of the command execution flow, contrary to what we just saw with the Nodes (Screens and Blocks). We opted to delete this nodes, although it can be argued that it can be safe to keep this actions, even if it demands manual work by developers if they want to erase the action and can cause some disorganization in the eSpace, because, for example, in the case of someone making a mistake in a feature file, renaming one of the steps and the corresponding Server Action. Then, the real action gets erased if we do not save it and if it has no other referers and

**Algorithm 4** Unused Screens and Blocks removal

---

```
1: procedure REMOVEUNUSEDNODES()
2:   scenariosToRun ← list of scenarios to process in the current execution
3:   scenarioBlocks ← set of Web Blocks identified by Scenario description;
4:   featureScreens ← set of Web Screens identified by Feature description;
5:   existsScenario ← boolean representing whether or not a Web block contains a
6:   scenario to be processed (contained in scenariosToRun);
7:   existsScreen ← boolean representing whether or not a Web Screen contains a feature
8:   present in at least a scenario of scenariosToRun;
9:
10:  for each node in Nodes.WebBlock do ▶ Where Nodes.WebBlock is the set of Web
    Blocks in the module
11:    existsScenario ← false;
12:    for each scenario in scenariosToRun do
13:      if scenario ⊂ node then
14:        existsScenario ← true;
15:      end if
16:    end for
17:    if existsScenario is false then
18:      for each referer in node.Referers do
19:        referer.Delete();
20:      end for
21:      if node ∈ scenarioBlocks then
22:        scenarioBlocks ← scenarioBlocks / node;
23:      end if
24:      node.Delete();
25:    end if
26:  end for
27:
28:  for each node in Nodes.WebScreen do ▶ Where Nodes.WebScreen is the set of Web
    Screens in the module
29:    existsScreen ← false;
30:    for each scenario in scenariosToRun do
31:      if scenario.feature == node.description then
32:        existsScreen ← true;
33:      end if
34:    end for
35:    if existsScreen is false then
36:      if node ∈ featureScreens then
37:        featureScreens ← featureScreens / node;
38:      end if
39:      node.Delete()
40:    end if
41:  end for
42: end procedure
```

---

so we lose the code. We chose to build this in a way that takes a little of both opinions into account. We only delete the unused Server Actions at the very end of the command execution and this will allow that, even if some actions appear to be unnecessary at some point in the scenario processing progress, if later in the scenario list appears one that implements some of these actions, those actions are still in the system and can be reused, before they are deleted. Otherwise, if they were initially deleted, then this specific case of reuse would not be taken into account. So, in the end, we check if the actions are used somewhere (if they have referers) and if they do not have, they are deleted from the flow and from the structure that keeps them (*serverActions*). If they have referers they are not deleted and are added to the (*serverActions*) structure if they are not already there. The pseudo-code for this mechanism is presented in algorithm 5.

---

**Algorithm 5** Server Actions Removal
 

---

```

1: procedure REMOVEUNUSEDACTIONS()
2:   serverActions ← set of Server Actions, identified by Scenario description;
3:   Flows.UserActions ← set of Server Actions present in the Test ESpace at the moment;
4:
5:   for each node in Flows.UserActions do
6:     if node.Referers is empty then
7:       Flows.UserActions ← Flows.UserActions / node;
8:       if node ∈ serverActions then
9:         serverActions ← serverActions / action;
10:      end if
11:     else
12:       if node ∉ serverActions then
13:         serverActions ← serverActions ∪ node;
14:       end if
15:     end if
16:   end for
17: end procedure

```

---

We now move on to the eSpace generation phase, with the creation (or reuse) of screens (algorithm 6) and blocks (algorithm 7).

At this stage we will go through the *scenariosToRun* list and process each scenario individually, starting with the feature, which should generate a new Web Screen if it does not already exist, and then moving on to the Web Block generation that should contain the properly filled BDD templates inside. Also, it should also be generated only if it does not already exist. During the Web Block generation phase we deal with the generation of Screen and Server Actions. We link the last ones to the BDD steps, already with the respective parameters on both sides. The presented pseudo-code represents a very simplified view of this extensive process.

Some decisions, concerning the nomenclature used for screens, blocks and actions, had to be taken into account at this stage. All the suggestions and best practices that the users of the *BDDFramework* presented in the interviews were considered in this phase.

Although, some had to be adapted given that now the generation is automated and there are some limitations of nomenclature in the display names of nodes and widgets in OutSystems. Some text boxes can not have white spaces, others have length or character limitations, but the general idea was to name Web Screens according to the tested feature (placing underscore characters where no spacing is allowed), to name scenarios in numerical order (there is a future possibility of allowing *a priori* scenario name editing directly in the feature file by placing an optional tag below the description, but we will talk about it in the future work) and assigning Screen Actions the corresponding Gherkin phrase type along with a number representing their order in the scenario structure and the sentence description. (For example if we have 2 Given clauses, one will be Given1\_sentence and the other Given2\_sentence, respectively). These numbers in the Screen Actions only refer to the one scenario inside each Web Block and as such will not cause phrase identification problems as they do not mix with the actions of the other blocks, unlike Server Actions which are all mixed in the Logic tab (but are not numbered). In these, the idea is that they are named according to the sentences they implement and as such can be identified and possibly grouped within folders that organize the actions by test component.

---

**Algorithm 6** Web Screens Generation

---

```
1: procedure SCREENSGENERATION()
2:   scenariosToRun ← list of scenarios to process in this execution
3:   featureScreens ← set of Web Screens identified by feature description;
4:
5:   for each scenario in scenariosToRun do
6:     if scenario.feature ∉ featureScreens then
7:       testScreen ← new WebScreen(scenario.feature);           ▷ The argument
       represents the screen name and it will be named after the feature it tests.
8:       featureScreens ← featureScreens ∪ (featureName, testScreen);
9:     end if
10:    testScreen ← featureScreens[featureName];           ▷ In case the screen already
       exists, we will use it
11:   end for
12: end procedure
```

---

Parameter detection is achieved through a set of string operations. This process is performed for each sentence individually and starts before the generation of Screen and Server Actions, since in the mentioned sections the Gherkin sentences are used as identifiers for the data structures and as a display name for actions in Service Studio. Therefore, it is necessary to first deal with the parameters since they must be ignored for the purpose of comparing and reusing sentences. If the sentences are detected as having parameters (by the use of quotation marks), we first extract them into a list, recursively iterating over the sentence and applying the extraction method to the substring that begins at the end of the last parameter found, until we find all the parameters of the sentence. This will result in a list in which parameters are first stored as strings. Later in the process we will

**Algorithm 7** Web Blocks Generation

---

```

1: procedure BLOCKSGENERATION()
2:   scenariosToRun ← List of scenarios to use in this execution
3:   scenarioBlocks ← dictionary of Web Blocks identified by Scenario description;
4:   screenActions ← set of Screen Actions, identified by Scenario description;
5:   serverActions ← set of Server Actions, identified by Scenario description;
6:   BDDScenario ← reference BDDScenario block from the BDDFramework
7:   scenarioCounter ← number that represents the number of scenarios
8:
9:   for each scenario in scenariosToRun do
10:    if scenario ∉ scenarioBlocks then
11:      testBlock ← new WebBlock(scenarioCounter); ▷ The argument represents
the Web Block displayed name in this case it will be named according to the scenario
number it represents.
12:      bddScenario ← new BDDScenario(testBlock); ▷ The argument represents
the parent Web Block for the BDDScenario, created in the previous line.
13:      (Section to get and fill the placeholders concerning the scenario
description, in the BDDScenario block.)
14:      for each step in scenario.steps do
15:        stepCounter ← integer that represents the number of steps from the
specified Gherkin type
16:        (Section to verify if the step has any parameter.)
17:        screenAction ← new ScreenAction(testBlock, Type + counter);
        ▷ The arguments are the parent block and the action name, represented by the
Gherkin type concatenated with the counter.
18:        (Logic that defines the content of the screen action created, composed
of Start and End nodes and a Run Server Action node, which will call the action
(Server) that implements the step we are processing.)
19:        if step ∉ serverActions then
20:          serverAction ← new ServerAction(step.name);
21:          (Logic that defines the content of the server action created,
composed of a Start node and a Raise Exception node, representing that the action
is not yet implemented - NotImplementedException.)
22:          action ← serverAction ▷ The created Server Action is assigned to
the Run Server Action present in the Screen Action previously defined.
23:          serverActions ← serverActions ∪ (name, action);
24:          (Section with the logic that implements the parameter type
inference and applies it to the generated Screen and Server actions generated, to
allow parameters to pass between actions with the correct type, avoiding type
errors on the platform side.)
25:        end if
26:        bddStep ← new BDDStep(bddScenario);
27:        (Section with the logic that implements the placeholder search in the
BDDScenario template and the definition of the BDDstep.)
28:        bddStep.Destination ← screenActions[step.name]; ▷ The
BDDStep template within BDDScenario is connected to a Screen Action that contains
its implementation.
29:      end for
30:    end if
31:    scenarioBlocks ← scenarioBlocks ∪ (testBlock.name, testBlock);
32:  end for
33: end procedure

```

---

discover their type, but for now anything between pairs of quotation marks is considered a parameter and it is stored in the list. After that, we will replace the parameters in the sentences with variables with the help of a counter. These variables denote the presence of the parameters in the implementation of the sentences. For example, in the sentence - I have "10" bananas and "12" lemons - will be stored as - I have X1 bananas and X2 lemons - and that will be the name of the corresponding actions. With this, we made our implementation extensible and test code easy to reuse in Server Actions, regardless of the input values that are passed. These are stored in the list of values and will not be ignored: they will be passed through the Screen Actions as input parameters expected in the Server Actions. This is possible because Screen Actions are directly linked to the Gherkin sentences (in a one-to-one fashion) and correspond to a specific instance with values, which are then passed to the Server Actions that implement those steps (in a many-to-one fashion). These, automatically receive these values and use their values for each scenario individually, during execution.

For the inference of the parameter types in OutSystems, we loop through each value in the parameter list of each step and create an input for each list entry while we see if the value matches any of the following OutSystems types: **Phone**, **Decimal**, **Date**, **Integer**, **Long**, **Boolean** or **Text**. We check it in this order as some parameters can fit more than one type so we first check the most specific types like the phone number since one can also be stored in an integer (for example) but the opposite might not be verified, before moving on to the most generic types. These validations are made through tests that verify if it is possible to convert the parameter to a certain format, either with the help of the functions provided by the basic types of the programming language used and also with the help of regular expressions for the compound and structured types, such as phone numbers which have length restrictions and dates that have formatting restrictions. If the conversion attempt for a given type fails, we move on to the next, and so on. Finally, if the value does not "fit" in any type, we store it with the **Text** type (default).

At the end of each Web Screen is placed a *BDDFramework* block (*BDDFinalResult*) which, as the name implies, summarizes the test execution results for the scenarios in the Web Screen where the block is placed. It shows how many scenarios have failed from the ones which were run. This allows us to see the overall result of running the BDD tests for each feature and having this block at the end of the screen is a requirement for the *BDDFramework* Test Execution API to work properly. However, this process needs to take into account some aspects, such as: this block is inserted just before we change the feature we are processing, that is when we are processing the scenarios and one is going to be placed in a different Web Screen. In this case we put the block at the end of the screen we were processing and this will not cause any problem in the same run since the scenarios are grouped under the feature they belong in the feature file and they will be processed in the same order. However, the command may be executed over and over again and as such it should be noted that the feature file may change and more scenarios may be added. So, for screens that are already defined with scenarios inside, if new scenarios are introduced

in them, it is necessary to avoid duplication of test blocks. We managed this by changing the position of the block with the final result to the bottom of the page, after the last test scenario. Therefore the blocks are not duplicated and they always appear at the bottom of the page, where they belong. This, and certain special cases: like just having a single test screen (feature) - we can not just insert the block when the feature changes (because it never will, like when we have multiple features) - had to be considered in the algorithm implementation. This also applies to when we are about to insert the *BDDFinalResult* Block at the bottom of the last test screen (when we have multiple features): we need a mechanism to deal with this and insert the block at the end of the last screen. The pseudo-code for this is presented in algorithm 8.

---

**Algorithm 8** Final Result Block Placement
 

---

```

1: procedure FINALRESULTBLOCK()
2:   testScreen ← the current Web Screen being processed;
3:
4:   for each scenario in scenariosToRun do
5:     (...)
6:     (In the end of processing each scenario...)
7:     if BDDFinalResultBlock ∉ testScreen then
8:       finalResultBlock ← new BDDFinalResultBlock(testScreen);
9:     else
10:      finalResultBlock.MoveToEnd(); ▷ If a Final Result Block already exists in
      the current screen we just move it to the end of the Web Screen page.
11:    end if
12:  end for
13: end procedure

```

---





## VALIDATION AND RESULTS

In this chapter we will describe the testing process that was performed for the Prototype validation and analyze in detail the obtained results. We conducted a quasi-experiment with real users, that we describe in this chapter. This quasi-experiment makes a direct comparison across performance, correctness and usability metrics with the existing approach (*BDDFramework*), among others, since the developed prototype is not supposed to be direct competition or even replace *BDDFramework*. It is just a component that uses the *BDDFramework* and enables other purposes, namely the BDD practice, allowing more automation in the testing process and broadening the user audience (as it enables the introduction of people without the same technical skills as the OutSystems developers, through facilitating the task of writing and filling in the scenarios), and potentiating the BDD process from a technological point of view. This is what we want to assess with this quasi-experiment. So, the formal and informal user feedback will be very important in understanding whether or not the objectives have been truly achieved. We will remember our initial goals and answer the research questions we asked ourselves in the beginning of this dissertation, illustrating, analyzing and discussing all the results.

In this chapter we present the experimental protocol used in all the conducted quasi-experiments, following Jedlitschka et al. guidelines [29] on how to report quasi-experiments in Software Engineering.

We will also go back to the framework comparison model presented in section 4.2.3 and introduce our prototype in it. This analysis will allow us to frame our component alongside its true competition (other BDD test automation frameworks like Cucumber and SpecFlow). The validation phase was preceded by a (functional) verification process of the framework that was done in parallel with production to detect bugs and failures, as well as the integration of all features with Service Studio.

## 6.1 Planning

With this quasi-experiment, we tried to understand whether or not the developed component can bring advantages to the OutSystems' testing process, **compared to the existing baseline - the *BDDFramework***. A user quasi-experiment has been set up with real-world test scenarios that are designed to be simple enough to be able to accomplish in a short period while utilizing the key features of the frameworks and putting into practice the fundamental principles of BDD, using the capabilities of the tools to make the process as fast and correct as possible. Some metrics were measured, such as task completion times, ease of the tools and correctness of solutions, but also some informal final feedback collected in the form of questionnaires.

### 6.1.1 Goals

Recalling the objectives mentioned in section 1.3, we started from the premise that BDD could be valuable to apply in an OutSystems context. We wanted to study and understand how to produce a testing tool to assist in this process. OutSystems has the *BDDFramework*, but it has some limitations as we can recall from section 4.1.2, and it can be improved to support the BDD process. With this in mind, we have opted for an approach that uses the *BDDFramework* to produce test scenarios in OutSystems but in a more automated way, focusing on facilitating the BDD process and close to what is done with other frameworks in other languages. As such, we want to understand if this goal was successfully achieved, comparing the testing processes with both approaches, the new and the old one.

We used the GQM research goal template to define the goals of our quasi-experiment[6]. The high level goal of our quasi-experiments can be defined as follows:

**Generic Goal:** *Analyze* the effect of the developed Prototype in the BDD testing phase (in comparison with the *BDDFramework* approach), *For the purpose* evaluation, *With respect to* the creation and reuse of BDD test scenarios, *From the view point of* researchers, *In the context of* quasi-experiments conducted in the OutSystems R&D<sup>a</sup> office, by professional OutSystems developers.

---

<sup>a</sup>Research and Development

This objective can then be divided into the following two sub-objectives we we aim to achieve during this quasi-experiment:

**Goal 1:** *Analyze* the effect of the developed Prototype in the BDD testing phase (in comparison with the *BDDFramework* approach), *For the purpose* evaluation, *With respect to* the creation of new BDD test scenarios, *From the view point of* researchers, *In the context of* quasi-experiments conducted in the OutSystems R&D<sup>1</sup> office, by professional OutSystems developers.

**Goal 2:** *Analyze* the effect of the developed Prototype in the BDD testing phase (in comparison with the *BDDFramework* approach), *For the purpose* evaluation, *With respect to* the reuse of existing BDD test scenarios, *From the view point of* researchers, *In the context of* quasi-experiments conducted in the OutSystems R&D<sup>2</sup> office, by professional OutSystems developers.

**Goal 3:** *Analyze* the effect of the developed Prototype in the BDD testing phase (in comparison with the *BDDFramework* approach), *For the purpose* evaluation, *With respect to* the usability, *From the view point of* researchers, *In the context of* quasi-experiments conducted in the OutSystems R&D<sup>3</sup> office, by professional OutSystems developers.

**Goal 4:** *Analyze* the effect of the developed Prototype in the BDD testing phase (in comparison with the *BDDFramework* approach), *For the purpose* evaluation, *With respect to* the ease of utilization, *From the view point of* researchers, *In the context of* quasi-experiments conducted in the OutSystems R&D<sup>4</sup> office, by professional OutSystems developers.

**Goal 5:** *Analyze* the effect of the developed Prototype in the BDD testing phase (in comparison with the *BDDFramework* approach), *For the purpose* evaluation, *With respect to* the correctness of the BDD tests, *From the view point of* researchers, *In the context of* quasi-experiments conducted in the OutSystems R&D<sup>5</sup> office, by professional OutSystems developers.

---

<sup>1</sup>Research and Development

<sup>2</sup>Research and Development

<sup>3</sup>Research and Development

<sup>4</sup>Research and Development

<sup>5</sup>Research and Development

### 6.1.2 Participants

The quasi-experiment was performed on **14 subjects**<sup>6</sup>, divided into **2 test groups** (with **seven users** each). One group experimented with the existing approach, using the *BDDFramework* only (control group<sup>7</sup>), and the other tested the developed Prototype (experimental group<sup>8</sup>). The subjects had experience with the OutSystems language, and they were recruited voluntarily from the R&D staff of OutSystems (in the office of Linda-a-Velha, Lisboa, local where this dissertation was carried out). All subjects participated with full consent and attended to help evaluating the tool developed in the scope of this dissertation, without any kind of reward. Participants were guaranteed the anonymity of their identity and responses, through a verbal agreement made prior to the start of the test.

Knowing the OutSystems language was a **necessary condition** for participating in the quasi-experiment since implementing the test code requires language knowledge. As already mentioned, the participants were **voluntarily selected**. However, we tried to find a way to have both: individuals with experience using the *BDDFramework* and individuals without experience using with *BDDFramework*, **equally distributed** among the test groups: of the **6** participants with experience, half (**3**) were placed in each group, and the remaining **8** individuals (without *BDDFramework* context) were placed **4** in each group. This separation for each of the groups was made **randomly**. We considered that this **intra-group diversity** of individuals in the test samples (despite the **inter-group balance**) could be **enriching to understand several perspectives**: *How is the adaptation of novice users to the developed component, in relation to how they adapt to the BDDFramework? How is the transition from the BDDFramework to the developed component from the point of view of an experienced user?.*

In addition to the 14 people already mentioned, there were also **2 pilots** (not accounted for the results), one for each approach, which served to tune the testing materials, the quasi-experiment estimated duration and ultimate the details for the real tests. The representation of the participants in the quasi-experiment and their distribution among the 2 independent groups can be found in the figure 6.1.

---

<sup>6</sup> We used 14 subjects given the availability of people with *BDDFramework* experience and since we wanted to have a balanced quasi-experiment

<sup>7</sup> the group experimenting with the existing approach

<sup>8</sup> the group experimenting with the new approach

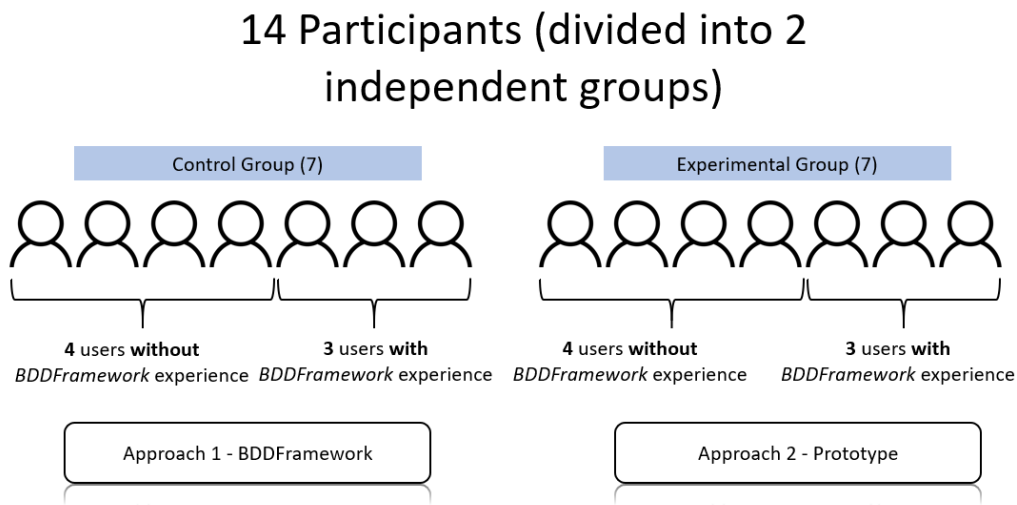


Figure 6.1: Scheme of the division of participants into two groups, experimental and control.

### 6.1.3 Materials

The quasi-experiments were conducted under the **same conditions and experimental material**. The participant did not need to bring anything with them, as all necessary equipment was made available. The quasi-experiments took place in meeting rooms at OutSystems' office in Linda-a-Velha, one participant at a time. The materials used in the quasi-experiments were:

- **One introductory guide** for each approach, briefly describing the subject of the thesis, the problem we had in hands and some basic concepts of Behavior-Driven Development development (in particular the Gherkin syntax needed to describe the scenarios). This first part of the document (which has a total of one and a half pages) is common to both approaches, and the final part of it varies, depending on the approach: in the case of users doing the test with the existing methodology then the script presents a brief description of the *BDDFramework* and in the case of the approach that uses the component that was developed in this dissertation, a brief description of it is presented. The guides are available in Appendix B;
- **One laptop**, with presentation support, Service Studio 11 installed and with Internet connection to be able to deploy the application and view the execution results in the browser;
- **Two questionnaires**: NASA-TLX (annex I[42]) and the System Usability Scale (SUS), in annex II[14];
- **One presentation** containing a practical example of the process using the tool (according to the approach in question) and the problems statement as well as some

useful information about the tests, like some notes on how to publish the module. This, as well as the introductory guide, are the only materials allowed for the participants to consult during the quasi-experiment;

- **Two published OutSystems applications** for the user to visually explore. Those correspond to the applications under test in the problem resolution phase.
- **One block** to take notes on test results (record the times, point out errors and suggestions at the end);
- **One stopwatch** to track activity time.

The last two materials presented were only used by the quasi-experiment host.

#### 6.1.4 Tasks

The requested tasks were the same for both approaches, considering we wanted to compare the frameworks, and as such, the problem should be the same, so the obtained results are not biased.

In total, each participant performed two challenges (**limited in time**). These were designed so that the total duration of the quasi-experiment did not exceed **60 minutes per participant**, including the response time for both questionnaires, the reading time for the brief introduction and also the demonstration of the framework (*BDDFramework* alone in the case of **approach 1** or Prototype in the case of **approach 2**) that preceded the realization of the tasks. As such, the challenges had to be relatively **simple** and straightforward, with **easy-to-interpret** problems so that they do not give participants any interpretation problems. Questions were not allowed while performing the tasks, but participants could ask questions before the start of the activities, if there were any questions when reading the task description. At the same time tasks should test the various features of the developed Prototype and the characteristics identified as being fundamental in a BDD test automation framework, so that we can analyze our component against the *BDDFramework*, **under the same conditions**.

The actual descriptions of the tasks presented to the participants can be found in annex [III](#):

- **Task 1:** We gave the participant an OutSystems' application (*Rectangle Area*), which can be seen in figure [6.2](#). The deployed app was available for the user to examine, but also the OutSystems application module (both could be consulted during the quasi-experiment). The participant was asked to implement **one test scenario** in a test module created for this purpose, that already contained the *BDDFramework* dependencies as well as the *Rectangle Area* project dependencies. This application asks for a length and a height for a rectangle, and by clicking the button to calculate the result, the area of the corresponding rectangle is calculated (length x height). The

test scenario that the participants were asked to implement should test valid length and height values (for instance 20 x 30) and must confirm that the returned result is correct. In this example, it should be 600. The task is considered to be complete when the test project is published, and the scenario passed with a positive result, including, obviously, the implementation of the test logic. This task has a **maximum duration of 15 minutes** and is **focused on the writing and implementation of a BDD scenario from scratch**, on a test project with nothing in it.

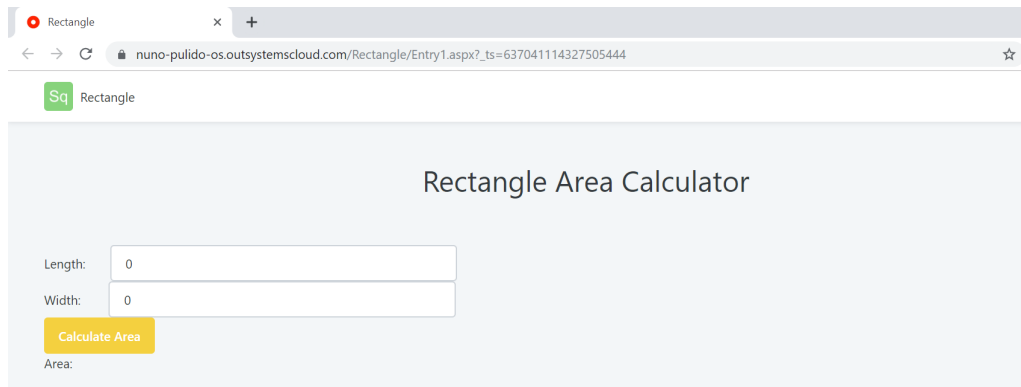


Figure 6.2: Interface of the *Rectangle Area* OutSystems application, created to test Task 1.

- **Task 2:** In this task, we return to the application covered in the demo part of the quasi-experiment, *Triangle Kind*, which can be seen in figure 6.4. This application, such as the one used in Task 1, was previously developed before the quasi-experiment, but unlike the previous one, **a test scenario had already been implemented** (the example implemented in the demonstration before the quasi-experiment). The application was available in the browser and kept open in Service Studio with everything properly imported and participants were asked to implement **a test scenario** in the same module that had been used for the demonstration (and therefore **already had a scenario**). Like in the previous task, the important logic was available in Server Actions to make the developer work easier in the test code implementation phase. The *Triangle Type* application, given the length of the 3 sides of a triangle, calculates its type: Equilateral, Isosceles, or Scalene: A triangle is called *Equilateral* if all sides are the same length; A triangle is called *Isosceles* if 2 sides are the same length, the other being different; A triangle is called *Scalene* if all sides have different lengths; If you enter lengths that do not form a valid triangle then the application returns that it is *Not a Triangle*. To build a triangle, all sides must be smaller than the sum of the others 2 sides. The example in the demo was the creation of a test scenario for the equilateral triangle, and the scenario was described as shown in the figure 6.3. Since this scenario already existed in the system (and was already implemented) the idea was now to take advantage of this implementation for the Scalene triangle test. The task is considered completed when the test project is published, and the scenario passed with a positive result, including

the implementation of the test logic. This task has a **maximum duration of 10 minutes** (the idea was to choose a duration that would made the user consider shortcuts like reuse sentences and to be able to accomplish the task in the allotted time) and is **focused on reusing existing phrases and implementations**.

```
1 Feature Type of Triangle
2
3 Scenario When all edges are valid and have the same length (3) then the triangle should be equilateral
4 Given First edge has length "3"
5 And Second edge has length "3"
6 And Third edge has length "3"
7 When The user clicks the button to calculate the triangle type
8 Then The result should be "Equilateral"
9
```

Figure 6.3: Scenario description to test the Equilateral Triangle demonstrated in the demo phase.

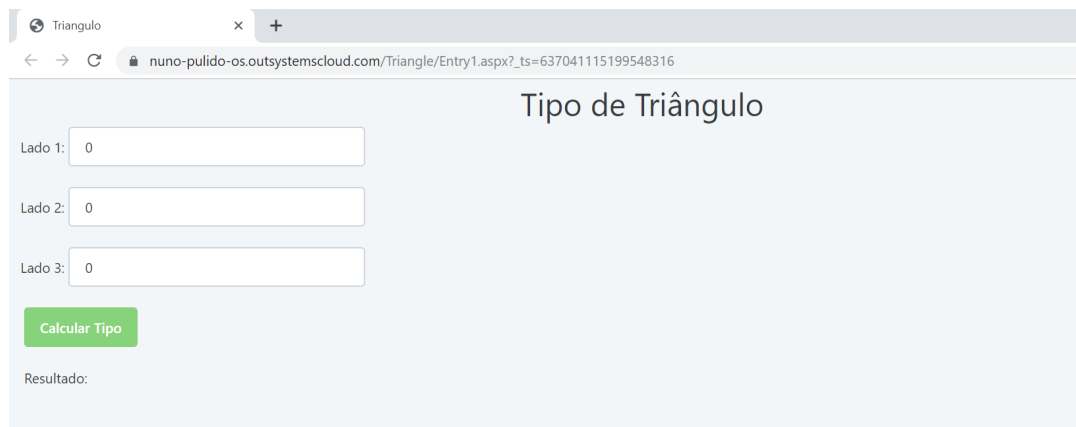


Figure 6.4: Interface of the *Triangle Kind* OutSystems application, created to test Task 2.

### 6.1.5 Hypotheses, Parameters and Variables

For the high level goals presented in section 6.1.1 we define the null (H0) and alternative hypotheses (H1):



- H<sub>0</sub>Creation:** Using the developed component does not influence the creation of new BDD tests.
- H<sub>0</sub>CreationSpeed:** The use of the developed component does not influence the speed of creating new BDD tests.
- H<sub>1</sub>Creation:** Using the developed component influences the creation of new BDD tests.
- H<sub>1</sub>CreationSpeed:** The use of the developed component influences the speed of creating new BDD tests.
- H<sub>0</sub>Reuse:** Using the developed component does not influence step reuse in BDD testing.
- H<sub>0</sub>ReuseSpeed:** The use of the developed component does not influence step reuse speed in BDD tests.
- H<sub>1</sub>Reuse:** Using the developed component influences step reuse in BDD testing.
- H<sub>1</sub>ReuseSpeed:** The use of the developed component influences step reuse speed in BDD tests.
- H<sub>0</sub>Correctness:** The use of the developed component does not influence the correctness of BDD tests.
- H<sub>1</sub>Correctness:** The use of the developed component influences the correctness of BDD tests.
- H<sub>0</sub>Ease:** The use of the developed component does not influence the ease of BDD testing.
- H<sub>1</sub>Ease:** The use of the developed component influences the ease of BDD testing.

Figure 6.5: Hypotheses and sub-hypotheses formulated for the desired high level goals.

The **independent variables** are presented in table 6.1.

| Name                   | Values  |
|------------------------|---|
| <i>Approach</i>        | <i>BDDFramework</i> , <i>Prototype</i>  |
| <i>User Experience</i> | <i>BDDFramework</i> experienced user,<br><i>BDDFramework</i> non-experienced user |
| <i>Task</i>            | creation (1), reuse (2)   |

Table 6.1: Overview of the *independent* variables.

The **dependent variables** for the hypothesis defined are presented in table 6.2.

| Name               | Values                           |
|--------------------|----------------------------------|
| <i>Speed</i>       | time taken to complete each task |
| <i>Ease</i>        | NASA TLX score                   |
| <i>Correctness</i> | Score                            |
| <i>Usability</i>   | SUS score                        |

Table 6.2: Overview of the *dependent* variables.

For the evaluation of the **time taken to perform each task** (speed), we collected the duration of it (*completion time* - *start time*), which allows measuring the efficiency each

framework in the two tasks (creation and reuse).

At the same time a static **correctness** assessment was also made according to the final solution presented by the user, joining the 2 tasks (creation and reuse) in a single score, as both are part of the test process as a whole and the tasks are undifferentiated regarding the correctness of the solutions found. This evaluation was based on the BDD characteristics presented in the course of this dissertation and the recommended practices of structuring of a BDD test in an OutSystems eSpace (section 2.18.1). All the evaluated point were presented to the participants during the quasi-experiment presentations and were applied in the demonstration. The result of this analysis was taken into account by assigning a dichotomous score of 0 or 1 for each of the following aspects in the final published project:

- **Self-descriptive Gherkin steps:** as the name implies a step is considered self-describing if it is self-explanatory of what it is testing (these should be short and summary, indicating concrete test values if any);
- **Correct eSpace organization:** 1 Web Screen for each feature, 1 Web Block for each scenario. The Web Blocks must be inside the Web Screen of the corresponding features;
- **Equal steps must have the same description:** when a new scenario is introduced, the same steps should have the same description (e.g. in the case of the triangle, all tests should use the phrase "Side 1 measures X"when indicating the length of the first edge if it is how it was initially described);
- **Reuse of test actions for equal steps:** in the case of the *BDDFramework* through shortcuts namely copy-paste of actions to reuse their implementation or with the creation of centralized Server Actions (like it is done in the Prototype). In the developed prototype it is dependent on the previous point: if equal new steps have the same description of the existing ones then the server actions implementing it are reused, without intervention required.
- **Assignment of meaningful names to screens/blocks:** for navigation in the Service Studio code purposes (these should not just be given to the descriptions in the scenarios) or when we need to return to the failing scenarios after we check the results report;
- **Use of the available Server Actions exporting the application code:** To truly test an application we must use its implementation logic to run the test instances, not a replication of it or simulating it on the test project side. Hence the need to export the main actions of applications in Server Actions that allow the logic they implement to be used for testing purposes in the test projects;

- **Tests run successfully:** observable from the output obtained when the test was published. This point concerns the 2 tasks since we have different applications for each task.

Note that mostly aspects related to the BDD process and the recommended practices of BDD testing in OutSystems are evaluated, therefore the correction of the implementation logic is not the main evaluation focus considered for the correctness. It is also important to note that the test may pass even if it is not well implemented: the scenarios may not be testing what they are supposed to test or they may not be covering well certain under test functionality (for example not using the project application exported logic but using an implemented logic on the test project side). Even if the test passes in the desired time it may always be the case that the test logic is not well implemented and this happened as we will see in the results.

For the evaluation of the **ease** of the process we use the NASA TLX questionnaire. This measures the perception of cognitive effort spent on the task through 6 metrics, with a weighted final result: Mental Demand, Physical Demand, Temporal Demand, Effort, Performance, Frustration. This questionnaire is presented in the end and takes into account the framework used and it is task independent as we refer to the process as a whole here. We will use the unweighted (raw) version of the test and later we will explain why in the upcoming sections.

Finally, a **usability** analysis is also performed using the System Usability Scale (SUS) test presented to the participants.

### 6.1.6 Quasi-experiment Design

As mentioned in previous sections, the sample of participants was randomly collected with the only requirement being to have an even number of participants, all of them with OutSystems experience and some of them also having *BDDFramework* experience (also in even number). Participants were randomly assigned to tasks: four subjects without *BDDFramework* experience for approach 1 and the other four for approach 2 and three subjects with *BDDFramework* experience for approach 1 and the same number for approach 2 (if a participant performed the *BDDFramework* tasks (creation and reuse), the next participant would be allocated to the Prototype quasi-experiment, so that the number of participants performing each task would be balanced).

We chose a **Between-subjects**<sup>9</sup> study design type of quasi-experiment because we wanted ask the same problems (tasks) for everyone and it is always hard and subjective to find a problem with equivalent difficulty and to minimize the learning effect of the participant. Since we are using similar processes the learning and knowledge factor will always have an impact because our approach also uses the *BDDFramework*. In addition,

---

<sup>9</sup>different people tested each approach, so that each person is only exposed to a single interface.

the implementation code (which is part of the task) will be the same and we use time metrics that need to be as accurate as possible. Even if we tried to do an **Within-subjects**<sup>10</sup> experimental design with the order of the tasks being changed between participants (half of the participants starting with approach 1 and the others with approach 2) as an attempt to balance the effects of learning and memorization, the results may still be affected by the ordering. Also, Between-subjects studies have shorter sessions than within-subject ones [25]. Of course we lose out on the number of people who did each test (with a within subject approach we could have more results, 14 per approach) and we lose some comparative feedback which can be very useful but we thought that everything considered a Between-subjects approach would be better. We tried to reduce the effect of the comparative feedback loss by showing at the end of the quasi-experiment how the other approach was and asking for informal feedback regarding both (for users with *BDDFramework* experience who have taken the approach using the prototype this is not even necessary as they already know the tool and this was one of the reasons we wanted some of our participants experienced people, to enrich the informal feedback). Regarding the observations and metrics that were taken from the quasi-experiments, these were exactly the same for each approach, as were the final questionnaires.

### 6.1.7 Quasi-experiment Procedure

One participant at a time made the quasi-experiment and these were carried out in the OutSystems' R&D office, in isolated meeting rooms booked for 1 hour and a half time-slots per participant. The quasi-experiments were meant to last about 60 minutes but counting on eventual delays or conversations we preferred to reserve the room for a little longer. We tried to make all quasi-experiments in the same room and choose the ones as remote and quiet as possible, however this was not possible due to the large number of advance reservations for those. We scheduled sessions according to the limited availability of each participant so this 14 quasi-experiments evaluation period lasted on for about 2 weeks.

This process is summarized in the scheme shown in figure 6.6.

---

<sup>10</sup>the same person tests all the processes (interfaces).

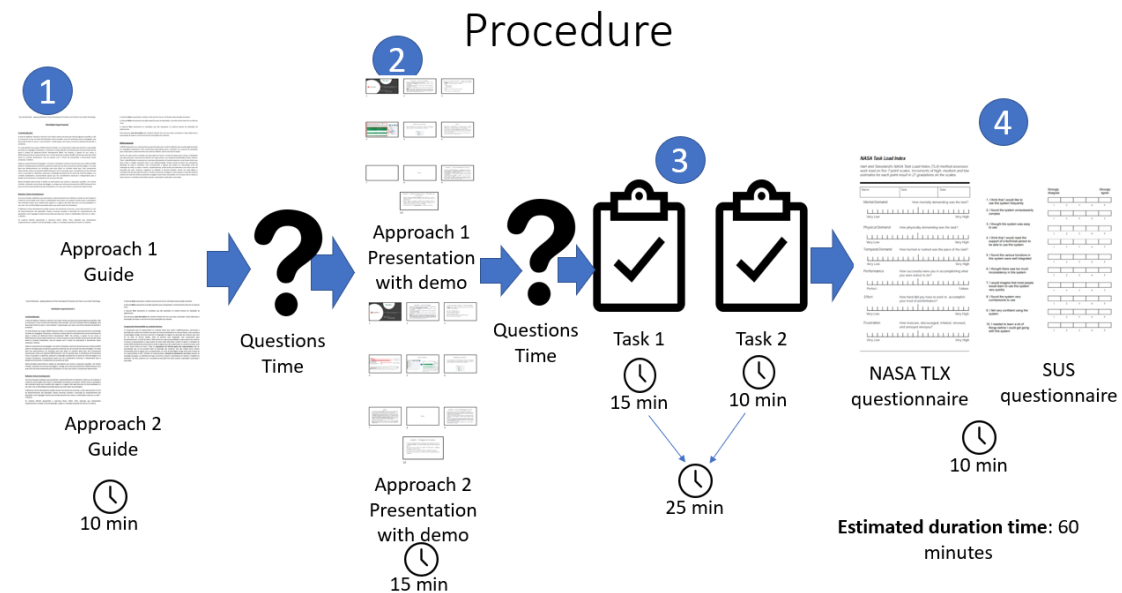


Figure 6.6: Visual representation of the quasi-experiment procedures and estimated times for each phase of the quasi-experiment. The introductory script (1) and the presentation (2) were different for each approach, while tasks and questionnaires were the same. Phases 1 and 4 were done on paper, while phases 2 and 3 were done using a laptop computer. The estimated duration of the tasks was 60 minutes, although in most cases it was less, since participants were able to perform the tasks in time. Following phase 4, there was usually an informal conversation in a more relaxed context (but still important) and outside the quasi-experiment environment, where participants gave their informal feedback and opinions about the frameworks and testing processes presented.

When the participant arrived in the quasi-experiment room, we first gave some thanks for their presence and then we began by giving the participant an introductory guide (1,5 pages) explaining the problem that led to the dissertation and the main objectives of the quasi-experiment. In this script, we also explained what is Behavior-Driven Development, with special emphasis on its purpose and describing in detail the Gherkin syntax, which was indispensable for the accomplishment of the tasks. The final section of this document concerned about the test framework that would be used in the quasi-experiment (*BDDFramework* or *Prototype*) and provided a brief description of its functionality.

At the end of this reading phase, the participant already had a bit more context, and so we explained which approach he would take during the quasi-experiment and we briefly described how the quasi-experiments would be carried by the participants: half of the participants to use the approach 1 (*BDDFramework*) and the other half approach 2 (*Prototype*), to accomplish the 2 BDD tasks requested. We concluded this phase by asking the participant if there were any questions so far and if they had understood well the concepts of BDD and Gherkin.

With no further questions, we would now approach the computer phase and tell the participant that we would now see how the process and assigned framework worked in practice (for the approach he was testing), through examples. To do so, we first presented

a test application, *Triangle Type*, which we described and demonstrated opening its implementation in Service Studio and its interface in the browser for the participant to explore at will. This application returns the type of a triangle given the size of its 3 edges. Then, we presented a test for the application (The Equilateral Triangle) as a Gherkin scenario:

## Cenário de teste exemplo – Triângulo Equilátero

- **Descrição:** Quando todas as arestas são válidas e têm o mesmo comprimento (3) então o triângulo é equilátero
- **Given:**
  - O lado 1 mede "3"
  - O lado 2 mede "3"
  - O lado 3 mede "3"
- **When:**
  - Utilizador clica no botão para calcular o resultado
- **Then:**
  - O resultado deve ser "Triângulo Equilátero"

Figure 6.7: Gherkin scenario given to the user as a test example for the application *Type Of Triangle*.

**Note:** The previous image was placed directly as it was presented to the user, as well as some of the materials that are illustrated throughout this section and as such they are in Portuguese language since it was like that we made the quasi-experiment materials available to users, since some of them might not be very comfortable with the English language especially in some more technical terms and we did not want that to be an obstacle in the comprehension and realization of the test.

Then we presented a snapshot of the referred scenario within Service Studio (a print inside the presentation), indicating the templates used and displaying the features of the *BDDFramework*, like the *AssertSteps* library and block templates. Some important mentions regarding the frameworks on how scenarios should be described and how the structure of the tests should be organized as well as how the reuse could be achieved and the use of parameters (reuse and parameters only in the case of the Prototype approach) were also made.

After this phase, we made a small demo where the user was shown how to test, step-by-step, the exemplified functionality (*Equilateral Triangle*), using directly the Gherkin descriptions presented to save some time. In both demos (*BDDFramework* or Prototype depending on the approach that the participant would validate), we covered the entire testing process, from scenario description to test code implementation (using the *Triangle Type* application code as well) and we published it in the browser as well, where we verified that the test obtained positive result (passed). The project under test was also made available to the participant for consulting, and the important logic duly exported

in Server Actions so they can be available to use in the test project (*TriangleTests*).

After the demo, we finally moved on to the task resolution phase: first, we presented task 1 and its statement. This task had a maximum time of completion of 15 minutes. We asked the user to test a scenario for another OutSystems application (*Rectangle Area*), which was also available for the user to explore, both in the browser and in Service Studio. Also, the user had a test project (*RectangleTests*) ready with all the dependencies (*BDDFramework* and *Rectangle Area*) imported. We created and imported the dependencies for them because this phase is not important for evaluation purposes and we can save some unnecessary time. When the user completed the first task (or the time was up), we moved on to task 2 (reuse), where the user was asked to implement one more scenario in the demo application, *Triangle Type*, in the same test project used in the demo phase, that already had a fully implemented test scenario (Equilateral Triangle). During the realization of the tasks no questions were allowed, as we did not want to influence and bias the test result.

Upon completion of the tasks, the user was asked to answer two questionnaires. First, the NASA TLX and then the System Usability Scale (SUS). The users were asked for some data such as their name and the date the quasi-experiment took place. In the meantime we pointed out in a notes block the results of both the correction of the tests and the time taken for each task.

In the end there was still time for some informal feedback the user wanted to give or recommendations.

## 6.2 Results and Analysis

In this section we will present the results of the quasi-experiments performed. For the analysis and comparison of **speed** between the two tools in the two tasks performed, we will use some descriptive metrics and also the Welch's t-test, which is a robust variant of Student's t-test, to test whether our prototype is an improvement to the existing framework, with the help of some visual schemes so that we can visually analyze the differences. In addition to **speed**, we will also evaluate the **ease** and the **correctness** of the solution achieved with the different tools. This last one by conducting a custom study based on the principles of BDD and BDD testing standards in OutSystems to assess correctness (ease and correctness are task independent). To evaluate the ease of the process we will evaluate the results of the NASA TLX questionnaire and we will also perform the also the Welch's t-test. Finally we will also analyze the results for the **usability** with the results from the SUS test and the Welch's t-test.

Table 6.3 summarizes some **descriptive statistics** for the **speed evaluation**, for the creation and the reuse tasks.

|          | Task            | Tool         | # | Mean   | SD     | Minimum | Maximum | Skew. | Kurt. |
|----------|-----------------|--------------|---|--------|--------|---------|---------|-------|-------|
| Speed(s) | Creation<br>(1) | BDDFramework | 7 | 660,14 | 81,28  | 576,00  | 765,00  | 0,35  | -1,75 |
|          |                 | Prototype    | 7 | 472,86 | 102,36 | 300,00  | 600,00  | -0,49 | -0,10 |
|          | Reuse<br>(2)    | BDDFramework | 7 | 419,00 | 125,14 | 275,00  | 600,00  | -0,29 | -1,59 |
|          |                 | Prototype    | 7 | 169,57 | 53,58  | 90,00   | 241,00  | -0,48 | -0,87 |

Table 6.3: *Speed* descriptive statistics for the creation and the reuse tasks.

These differences are easier to see visually with the aid of the 6.8 and 6.9 boxplots.

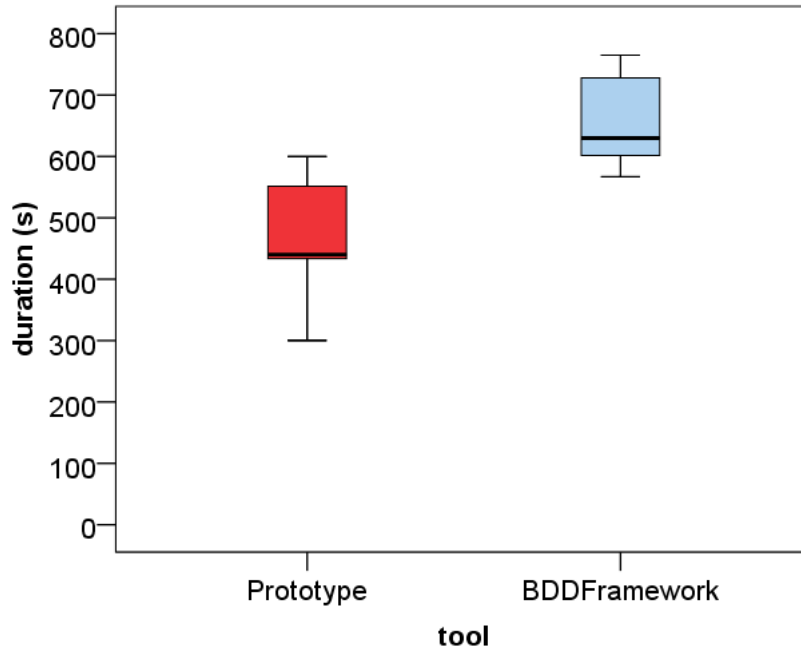


Figure 6.8: Speed for the BDDFramework (blue) and Prototype (red) in task 1.

Concerning the **ease** evaluation, after completing the 2 tasks, participants answered the **NASA TLX** questionnaire, which can be found in annex I. The results for both tools are presented below. We opted to use **unweighted scores** for the workload calculation to reduce the amount of time needed to perform the TLX test and not lengthen the quasi-experiment even more and also because several studies have compared raw NASA TLX scores to weighted NASA TLX scores and have found mixed results. Some say we gain better sensitivity when removing weights, others say the contrary[27]. The questionnaire presents a scale (0 to 100) for 6 measures that ultimately result in a score that evaluates the workload, in this case the tool we are using. The interpretation of the results is made with table 6.4.

Table 6.5 and presents the workload results for both tools.

Table 6.6 summarizes some **descriptive statistics** for the **ease evaluation**, for both tools.

In the graph of figure 6.10 we compare the workload score for each one of the 6 NASA TLX workload metric, individually, and in the boxplot in figure 6.11 we compare the mean workload score with both approaches.



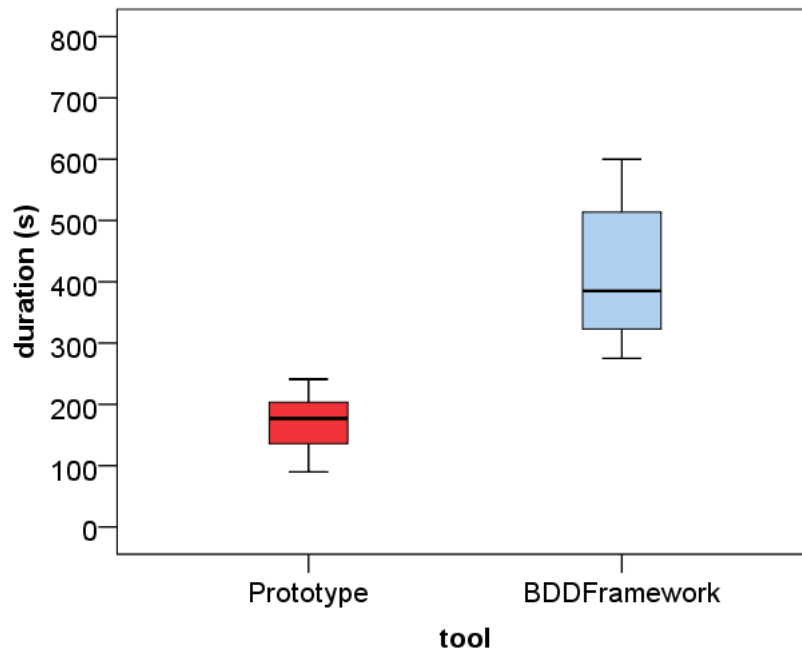


Figure 6.9: Speed for the BDDFramework (blue) and Prototype (red) in task 2.

| Workload      | Value  |
|---------------|--------|
| Low           | 0-9    |
| Medium        | 10-29  |
| Somewhat High | 30-49  |
| High          | 50-79  |
| Very high     | 80-100 |

Table 6.4: NASA TLX score interpretation.

| Category                       | BDDFramework | Prototype    |
|--------------------------------|--------------|--------------|
| Mental Demand                  | 50,00        | 20           |
| Physical Demand                | 5,00         | 5,71         |
| Temporal Demand                | 48,57        | 15           |
| Performance                    | 21,43        | 13,57        |
| Effort                         | 57,14        | 15           |
| Frustration                    | 64,29        | 7,86         |
| <b>Mean Score (Unweighted)</b> | <b>41,07</b> | <b>12,86</b> |

Table 6.5: NASA TLX mean scores for both tools.

| Tool                | Mean  | SD    | Minimum | Maximum | Skew. | Kurt. |
|---------------------|-------|-------|---------|---------|-------|-------|
| <b>BDDFramework</b> | 41,07 | 22,89 | 5,00    | 64,29   | -0,92 | -0,62 |
| <b>Prototype</b>    | 12,86 | 5,23  | 5,71    | 20,00   | -0,22 | -0,75 |

Table 6.6: NASA TLX descriptive statistics.

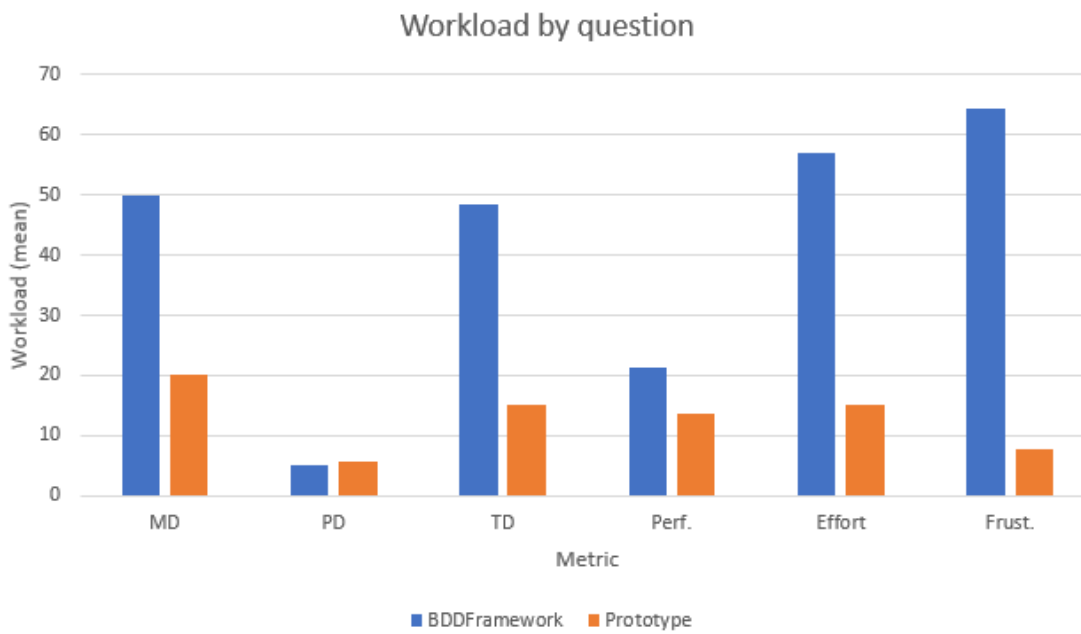


Figure 6.10: Graph comparing the mean score for each of the workload metric in both approaches.

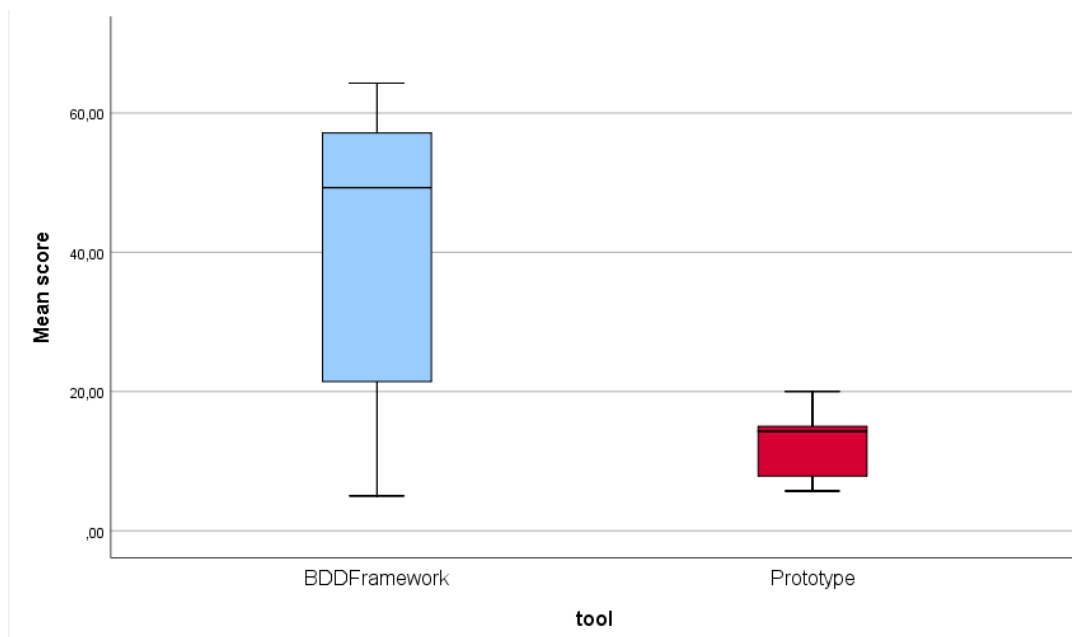


Figure 6.11: Boxplot for the Nasa TLX mean classifications for both tools.

Concerning the **correctness** evaluation, we kept the responses (test eSpaces) of each of the participants (with their consent) for each task and analyzed them, evaluating their BDD tests on a dichotomous scale (yes or no), regarding whether or not each of the correctness requirements was met. These requirements, as mentioned earlier (in section 6.1.5), result from the principles of BDD as well as some recommended practices for BDD testing in OutSystems. All of these were presented to the user during the trial presentation phase and during the demo. Table 6.7 shows the percentage of correct answers for each of the evaluated correctness requirements, for the *BDDFramework* and for the Prototype, respectively.

| Question  | Correct rate (%) |              |
|---|------------------|--------------|
|   | BDDFramework     | Prototype    |
| 1. The participant wrote self-descriptive Gherkin steps.  | 71,43            | 71,43        |
| 2. The participant created a Web Screen for each feature. Each BDDScenario was placed inside a Web Block that must be within the Web Screen of the corresponding feature. | 71,43            | 100          |
| 3. The participant used the same sentences for equal step descriptions.   | 57,14            | 85,71        |
| 4. The participant used mechanisms/shortcuts to reuse the actions with the implementation logic for equal steps.  | 57,14            | 100          |
| 5. The participant assigned meaningful names to test screens/blocks.  | 85,71            | 100          |
| 6. The participant used the exported Server Actions from the tested application.  | 85,71            | 85,71        |
| 7. The participant published the test project for task 1 and tests successfully run.  | 100              | 100          |
| 8. The participant published the test project for task 2 and tests successfully run.  | 100              | 100          |
| <b>Mean</b>   | <b>78,57</b>     | <b>92,86</b> |

Table 6.7: Overview of the correctness results in the BDDFramework and in the Prototype.

Complementing the tasks, at the end of the quasi-experiment we made a System Usability Scale (SUS) test to each one of the participants, regarding the framework used (*BDDFramework* or Prototype), to evaluate **usability**. This test is task-independent since it is focused on the usability of the tools. SUS is a simple, ten-item questionnaire that offers a global view of subjective assessments of usability indicating the degree of agreement or disagreement with each one of the 10 items on a five point scale (1-Strongly disagree, 5-Strongly agree), as we can see in figure 6.8[14]. The SUS questionnaire presented to the participants can be found in Annex II

Each item's score contribution will range from 0 to 4. For items **1,3,5,7**, and **9** the score contribution is the scale position minus one and for items **2,4,6,8** and **10** the contribution

is 5 minus the scale position). We should multiply the sum of the scores by 2.5 to obtain the final SUS score. Odd items have higher values as the most favorable response, while even items have lower values as the most favorable response. The questionnaire is designed in this way to avoid biasing the answers by giving the participant that the highest results are always better, or always worse[14]. By applying these transformations to the score values we get a more uniform scale, and it was after applying these transformations that we drew the following tables and graphs.

| Score  | Rating                               |
|--------|--------------------------------------|
| <25    | Worst imaginable<br>(Not Acceptable) |
| 26-49  | Poor<br>(Not Acceptable)             |
| 50-52  | Ok<br>(Not Acceptable)               |
| 53-73  | Good<br>(Marginal)                   |
| 74-85  | Excellent<br>(Acceptable)            |
| 86-100 | Best imaginable<br>(Acceptable)      |

Table 6.8: Meaning of SUS score.

Tables 6.9 and 6.10 show us the mean classification given by the participants to each SUS question, for the *BDDFramework* and for the Prototype approaches, respectively.

| Item  | Mean answer |
|---|-------------|
| 1. I think that I would like to use this system frequently                                    | 2,00        |
| 2. I found the system unnecessarily complex.  | 1,43        |
| 3. I thought the system was easy to use.  | 2,29        |
| 4. I think that I would need the support of a technical person to be able to use this system. | 2,14        |
| 5. I found the various functions in this system were well integrated.                         | 2,43        |
| 6. I thought there was too much inconsistency in this system.                                 | 3,00        |
| 7. I would imagine that most people would learn to use this system very quickly.              | 2,00        |
| 8. I found the system very cumbersome to use.   | 1,14        |
| 9. I felt very confident using the system.  | 1,86        |
| 10. I needed to learn a lot of things before I could get going with this system.              | 1,71        |

Table 6.9: Mean SUS answer for each question, for the *BDDFramework* testers.

Figure 6.12 shows us a graph comparing the mean classifications for each SUS item, in both quasi-experiment approaches.

Table 6.11 represents the descriptive statistics recorded for the SUS test and figure

| Item  | Mean answer |
|---|-------------|
| 1. I think that I would like to use this system frequently                                    | 3,71        |
| 2. I found the system unnecessarily complex.  | 4           |
| 3. I thought the system was easy to use.  | 3,86        |
| 4. I think that I would need the support of a technical person to be able to use this system. | 3,57        |
| 5. I found the various functions in this system were well integrated.                         | 4           |
| 6. I thought there was too much inconsistency in this system.                                 | 4           |
| 7. I would imagine that most people would learn to use this system very quickly.              | 3,71        |
| 8. I found the system very cumbersome to use.   | 3,71        |
| 9. I felt very confident using the system.  | 3,71        |
| 10. I needed to learn a lot of things before I could get going with this system.              | 3,71        |

Table 6.10: Mean SUS answer for each question, for the Prototype testers.

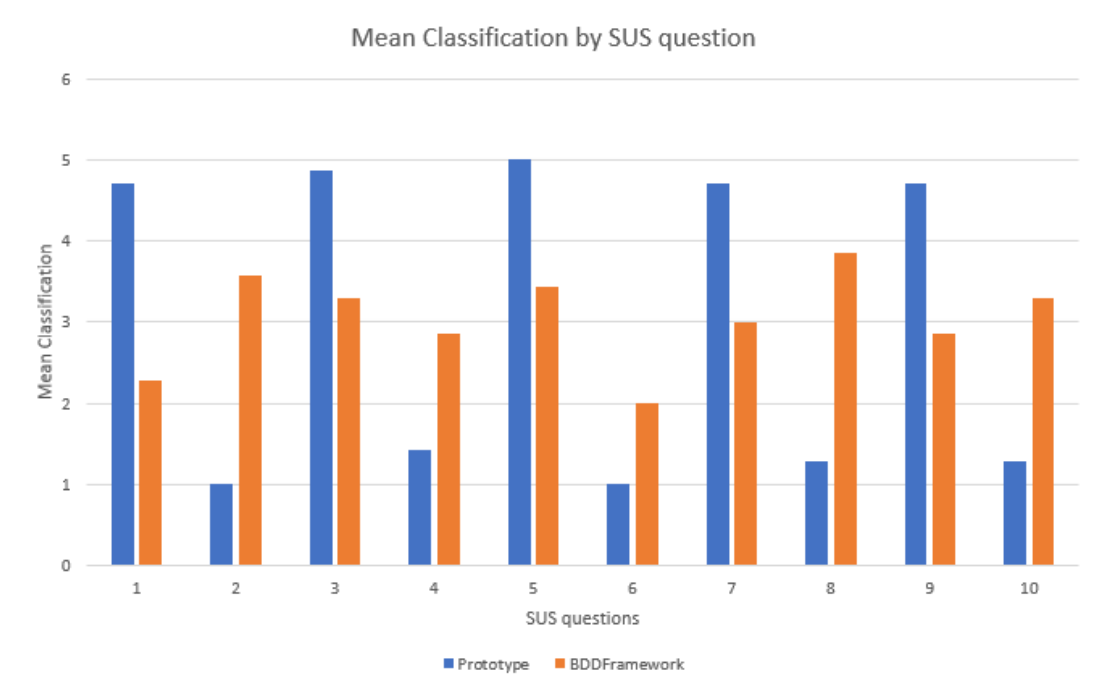


Figure 6.12: Graph representing the mean SUS responses for each item, for both frameworks.

6.13 the boxplot with the distribution of the results.

| Tool                | # | Mean  | SD   | Minimum | Maximum | Skew. | Kurt. |
|---------------------|---|-------|------|---------|---------|-------|-------|
| <b>BDDFramework</b> | 7 | 50,00 | 9,79 | 37,50   | 65,00   | 0,47  | -0,88 |
| <b>Prototype</b>    | 7 | 95,00 | 5,95 | 85,00   | 100,00  | -0,83 | -0,72 |

Table 6.11: SUS descriptive statistics.

In the boxplot in figure 6.13 we can visually compare the mean SUS score of both approaches.

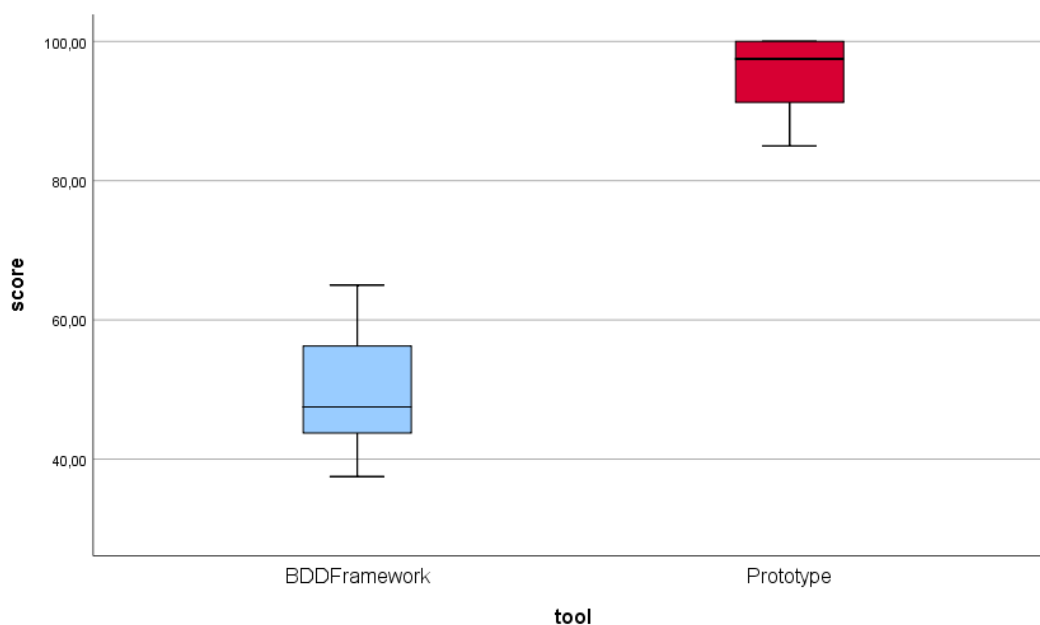


Figure 6.13: Boxplot for the SUS mean score for both tools.

### 6.3 Discussion

Concerning the **speed assessment**. As we can see from the table 6.3, for the creation task, *BDDFramework* users performed the exercise in an average time of **660.14** seconds and Prototype users in an average time of **472.86** seconds, **considerably faster**. For the reuse task, *BDDFramework* users performed the exercise in an average time of **419** seconds and Prototype users in a considerably faster average time of **169.57** seconds. The variance of the results is not very pronounced, being more significant for *BDDFramework* in the reuse task. This was the most unstable and inaccurate combination of factors (Coefficient of variation =  $125.14 / 419.00$ , approximately 0.3), and this may be due to the fact that there may presumably be some variability caused by factors such as participants' experience and above all the fact that *BDDFramework* has no defined reuse mechanisms and as such, to face this challenge each participant opted for different approaches and shortcut strategies (copy and paste of scenarios for example) while others made the process slower.

As can be seen in the boxplots in figures 6.8 and 6.9, there is no inter-quartile range (IQR) interception, which in itself demonstrates that the differences are noticeable between both task 1 and task 2 approaches. In such small sampling there is always a high probability that the results will be affected by factors external to the experience such as the developers individual skill or even the mood of the test taker (may have felt the pressure of being evaluated), but these are factors that we can not easily control.

One of the factors that can affect variance among participants is the **experience factor**. As noted earlier, each sample of 7 users, who tested each of the tools, had 3 experienced users. In figures 6.14 and 6.15 you can see the graphs with the average times for each approach, in the two tasks performed, organized by user experience. The differences between experienced and non-experienced users are not very significant even though they exist (experienced users were slightly faster) and were more noticeable in approaches using *BDDFramework*, especially in the reuse task. This can mean several things like what has been said earlier about the faster ways to reuse that people with extensive user experience already have. Since the tool does not support it directly, nor does it have mechanisms for it, novice users tend to have a harder time doing it quickly. Also, the fact that, probably, the developed prototype is easier to understand and learn on a first contact, because of the automation that exists in the structuring and creation of the tests, and in the prototype the user makes most of the interactions with Gherkin in the feature file, not the development platform, simplifying the process further. We will see this later in this section with the analysis of the NASA TLX questionnaire to evaluate the ease of using the tools.

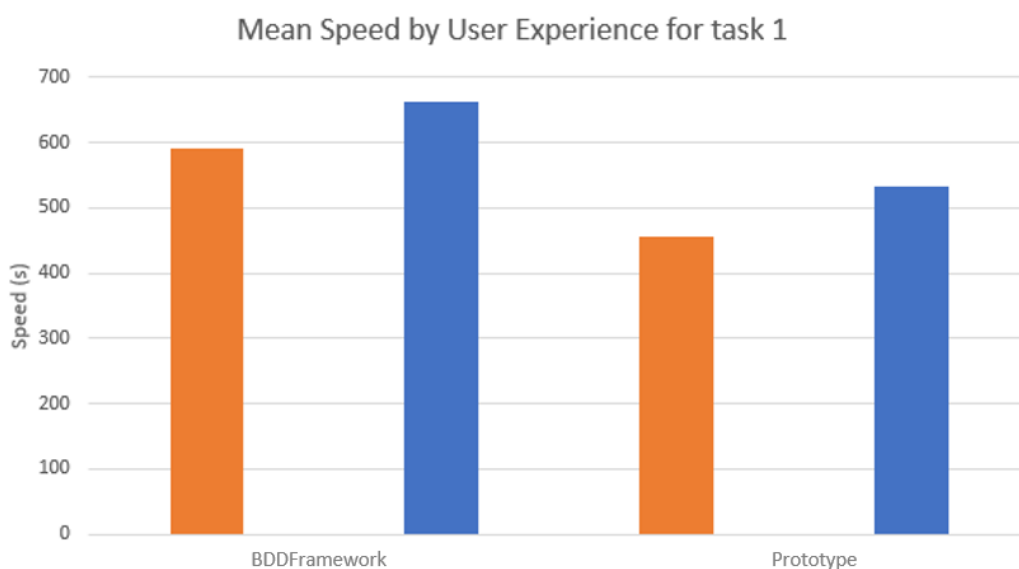


Figure 6.14: Mean speed for task 1, for participants with *BDDFramework* experience (orange) and for participants without *BDDFramework* experience (blue).

Sometimes when participants published the test application at the end of the test to

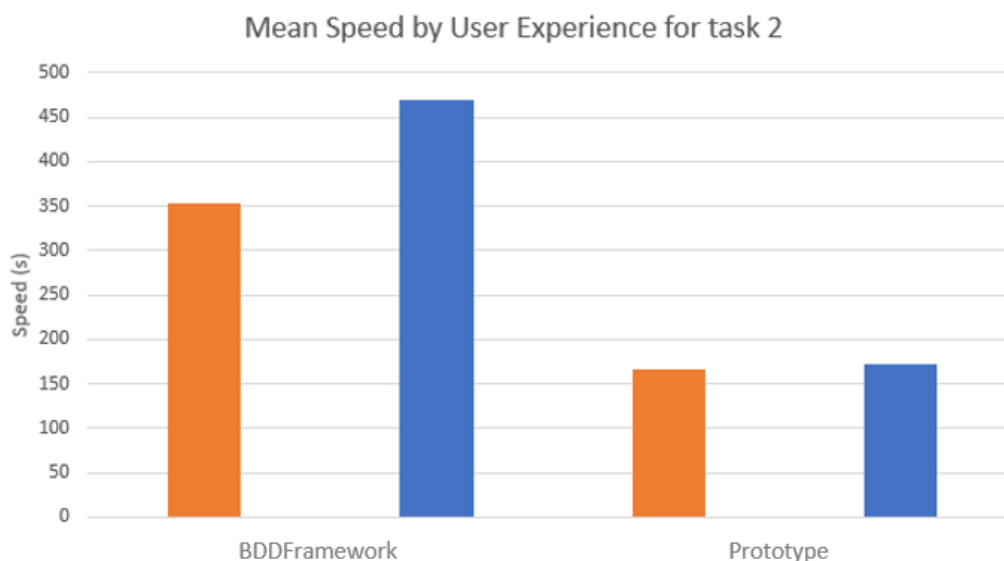


Figure 6.15: Mean speed for task 2, for participants with *BDDFramework* experience (orange) and for participants without *BDDFramework* experience (blue).

see results (time only stopped when the scenario passed with a positive result) and realized that they had not given permission to the Anonymous user role or they had no add Entry Point node in the application they could not see their results and had to publish again when they noticed this. The fact that the Prototype already does this automatically facilitates the process avoiding this kind of error. These and other factors may have slightly influenced the time differences, which are still small if we compare the 15 and 10 minute timeboxes for the first and second tasks, respectively (the **minimum** duration for task 1 using the *BDDFramework* was 567 seconds and the **maximum** duration was 765 seconds, **minimum** duration for task 1 using Prototype was 300 seconds and the **maximum** duration was 600 seconds, **minimum** duration for task 2 using *BDDFramework* was 275 seconds and the **maximum** duration was 600 seconds; for task 2 using the prototype the **minimum** duration was 90 seconds and the **maximum** duration was 241 seconds).

|       | Statistic | df1 | df2   | Sig. |
|-------|-----------|-----|-------|------|
| Welch | 14.37     | 1   | 11.41 | .003 |

Table 6.12: Welch t-test for task 1, concerning the speed variable.

An independent-samples **Welch t-test** was conducted to compare speed in *BDDFramework* and Prototype conditions (table 6.12) in the creation task. There was a **significant difference** in the scores for the Prototype ( $M=472.86$ ,  $SD=102.36$ ) and the *BDDFramework* ( $M=660.14$ ,  $SD=81.28$ ) conditions;  $t(11.41)=14.37$ ,  $p = 0.003$ . These results suggest that Prototype really **does have an effect on speed of completion of task 1** (creation).



Specifically, our results suggest that when users use our Prototype, they can create scenarios **faster** than what they can with the *BDDFramework*.

|       | <b>Statistic</b> | <b>df1</b> | <b>df2</b> | <b>Sig.</b> |
|-------|------------------|------------|------------|-------------|
| Welch | 23.50            | 1          | 8.13       | .001        |

Table 6.13: Welch t-test for task 2, concerning the speed variable.

An independent-samples **Welch t-test** was conducted to compare speed in *BDDFramework* and Prototype conditions (table 6.13) in the reuse task. There was a **significant difference** in the scores for the Prototype (M=169.57, SD=53.58) and the *BDDFramework* (M=419.00, SD=125.14) conditions;  $t(8.13)=23.50$ ,  $p = 0.001$ . These results suggest that Prototype really **does have an effect on speed of completion of task 2** (reuse). Specifically, our results suggest that when users use our Prototype, they can reuse scenarios **faster** than what they can with the *BDDFramework*.

It is expected that these speed difference results could have an even greater significant impact on larger problems, although we cannot state it with statistical certainty given that the problem only calls for the implementation of 2 simple scenarios for different problems. In practice, test projects should have many more scenarios and if this time saving we add on the quasi-experiments is constant then the time saved at the end for bigger projects will be very significant. However, we can not state this for sure (just assume) as we do not have data to support it and it may also be the case that the differences subside as *BDDFramework* has a slightly higher level of complexity and workload, and with time to learn its users can recover some time (in this quasi-experiment some people were completely new to the tool). Although this can be true, we do can not be sure of that as the Prototype has reuse mechanisms which for large projects will at some point make the scenarios automatically reused only by equality of Gherkin steps. Repeating sentences added to the system are already implemented most of the time and in the *BDDFramework* this automated reuse is not supported, there always has to be manual developer work.

Concerning the **ease assessment**. As we can see in tables 6.5 and 6.6, the Prototype mean workload is **12,86** which is considered **Medium** with the classifications from table 6.4. This workload is significantly less than the *BDDFramework* workload which is **41,07** (Somewhat High). If we look at the metrics individually, Prototype has a smaller workload on all of them. The main differences are mainly in Mental Demand, Temporal Demand, Effort and Frustration. The differences in physical demand and performance are not so sharp as neither of the tools is physically demanding. In terms of performance the small gain of the prototype may be due to the participants being able to come up with the most correct solutions but in both approaches all participants were able to accomplish the tasks with high success.

|       | Statistic | df1 | df2  | Sig.  |
|-------|-----------|-----|------|-------|
| Welch | 8,66      | 1   | 5,52 | 0,029 |

Table 6.14: Welch t-test for the NASA-TLX, concerning the ease variable.

An independent-samples **Welch t-test** was conducted to compare the ease in *BDDFramework* and Prototype conditions (table 6.14). There was a **significant difference** in the scores for the Prototype (M=41.07, SD=22.89) and the *BDDFramework* (M=12.86, SD=5.23) conditions;  $t(5.52)=8.66$ ,  $p = 0.029$ . These results suggest that Prototype really **does have an effect on the ease of the framework**. Specifically, our results suggest that when users use our Prototype, they can perform BDD tests **easier** than what they can with the *BDDFramework* alone.

Concerning the **correctness assessment**. As we can see in table 6.7, the correctness is higher in the Prototype (92.86%) than in the *BDDFramework* (78.57%). This may be due to the fact that some of these selected requirements are already automatically met by the prototype thanks to automation, which can lead to the major differences that are mainly found in questions 2 and 4. In question 2 it is natural since the process of creating the blocks and dragging/copying them to the appropriate locations can lead to some errors, especially for less experienced users. The most common mistake in this regard was placing *BDDScenarios* from the *BDDFramework* directly on Web Screens, other than within a reusable Web Block. This is a bad practice which the user would not easily notice he made as it would not cause problems to perform this specific tasks and so it would pass unnoticed. Requirement 4 had the greatest discrepancy in the responses. Again this is a point that is automatically guaranteed in the Prototype but it is directly up to the participant to describe the same steps with the same sentences, which did happen to all participants in Prototype, except for one. This is something that is not achieved directly just by using the Prototype. Sentence description has to be done anyway but it is expected to be more motivating to write in a text editor the user likes (not in the context of the quasi-experiment since they had no choice) than in Service Studio inside placeholders. However, the errors in the scenario description were the same in both cases (requirement 1). In any case, in point 4 we had 100 % correctness in the reuse of actions, although we did not have full correctness in point 3 (in the Prototype). This may be because the participant manually reused Server Action for different steps. However, this could have been achieved directly if the descriptions were the same.

It should also be noted that if we exclude inexperienced users, the correction results would be quite identical in both approaches. It is noteworthy that all participants were able to complete both challenges within the time limit, hence the ratings of questions 7 and 8 are 100 % (the test is only considered completed when it is published and the scenarios pass all), although some were short in time in the end (near the limit). It was

intended that the time limit was comfortable as we wanted to give the participant the opportunity to complete the test and obtain more complete results and to perform the test calmly (even if timed) and to do so without feeling too pressured. However, imposing a limit was necessary to force the participant to pay some attention to that factor and also because we did not want to lengthen the quasi-experiment too much.

Concerning the **usability assessment**. The result obtained for the mean SUS score of the Prototype was **95** which is a very promising result according to table 6.8. The BDDFramework SUS score by itself is **50** which is OK (according to table 6.8), but still **far from the prototype in terms of usability**.

|              | Statistic | df1 | df2  | Sig.  |
|--------------|-----------|-----|------|-------|
| <b>Welch</b> | 108,00    | 1   | 9,90 | 0,000 |

Table 6.15: Welch t-test for SUS mean score.

An independent-samples **Welch t-test** was conducted to compare SUS scores in *BDDFramework* and Prototype conditions (table 6.15). There was a significant difference in the scores for Prototype (M=95.00, SD=5.95) and *BDDFramework* (M=50.00, SD=9.79) conditions;  $t(9.90)=108.00$ ,  $p = 0.000$ . These results suggest that Prototype really **does have an effect on usability**. Specifically, our results suggest that when users use our Prototype, they have a **better usability experience** then what they can with the BDDFramework.

## 6.4 Comparison with other well-know BDD automation tools

After validating our solution in the OutSystems context and verifying that our prototype is an improvement for the BDD process over the BDDFramework alone, it is time now to see what the prototype looks like compared to other well-know BDD test automation tools that exist for other programming languages. To do so, we took the study of Wang and Solís, presented in sections 3.2 and 3.2.1, and decided to insert the developed Prototype into it, with the new evaluation model proposed in section 4.2.3. The comparison is made in table 6.16.

**Notes about the table:** Parameter type inference is not supported in the BDDFramework but this attribute is a direct consequence of the previous one (we can not infer parameter types if we have no parameter detection). Regarding scenario outlines (\*1), both the BDDFramework and the Prototype do not support them directly but the way the OutSystems language is built and the way it works it is possible to perform these kind of tests with relatively ease in the BDDFramework. There is even some documentation teaching how to perform Data-Driven Testing in OutSystems with the BDDFramework.[59] and consequently in the Prototype. Integration with other tools is also partially supported (\*2). Although the BDDFramework and the Prototype both lack more complete

| Feature   | Cucumber | SpecFlow | BDDFramework | Prototype |
|---|----------|----------|--------------|-----------|
| Feature and Scenario description in text files (Outside an IDE) | ✓        | ✓        | ✗            | ✓         |
| Step Definition Generation                                      | ✓        | ✓        | ✗            | ✓         |
| Automated Parameterization of Inputs in the Gherkin Sentences   | ✓        | ✓        | ✗            | ✓         |
| Parameter Type Inference  | ✓        | ✓        | -            | ✓         |
| Scenario Outlines   | ✓        | ✓        | *1           | *1        |
| Step Aggregation  | ✓        | ✓        | ✗            | ✗         |
| Hooks (Setup/Teardown)  | ✓        | ✓        | ✓            | ✓         |
| Step Reuse Detection  | ✓        | ✓        | ✗            | ✓         |
| Automated Execution   | ✓        | ✓        | ✓            | ✓         |
| Integration with Management/Reporting Tools                     | ✓        | ✓        | *2           | *2        |

Figure 6.16: The most used BDD frameworks compared with the Prototype and the *BDDFramework* in the new proposal of evaluation model. Attributes marked with a X mark are missing.

reporting mechanisms about the detailed current state of the tests in a project, we have some ways available (with the support of the REST API *BDDFramework*[58]), to manage some information reporting how many test scenarios failed and passed in the executions (with detailed information about the failing ones). In this particular aspect, the Prototype may even have some advantage in relation to the *BDDFramework*, since it works with text files, which can be **easier to integrate with other tools**, unlike the *BDDFramework*, where scenarios are described within Service Studio.

As we could see, many of the features that we defined as important in a BDD test automation tool that were present in some of the more complete tools are now also available in the prototype we developed which uses the *BDDFramework*. It is now possible to describe scenarios and features in text files and import them directly into the development platform, with automatic generation of test screens and step definitions in OutSystems

#### 6.4. COMPARISON WITH OTHER WELL-KNOW BDD AUTOMATION TOOLS

---

actions, following a standard organization and structure that allows reuse of existing sentences and consequent reuse of the code. Another aspect that was not previously supported is parameterization, which is now possible and which supports inference of data types. One problem we did not address was the lack of integration with other tools to allow more complete report mechanisms about the BDD scenarios of a test project, although the text files that the prototype deals with are more manageable in this aspect. The Prototype lacks as well some Data management mechanisms across scenarios (we have available the BDDFramework hooks) such as step aggregations and outlines, despite the fact that the OutSystems language already provides some mechanisms that make it easier to naturally perform this kind of testing than in other languages[59].



## CONCLUSIONS

In this section, we will first revisit the work that has been done throughout this dissertation. Next we will see the contributions made and finally, present some suggestions for future work.

## 7.1 Overview of the developed work

OutSystems provides a low-code application delivery platform that comprises and simplifies every stage of app development and delivery process. OutSystems enables fast, agile and continuous development, delivery and management of web and mobile applications. One of the fundamental aspects of this low-code language is that it has a **fast development speed** hence the importance of pushing testing activities closer to the moment when development is done. Also, establishing a standardized way for developers and testers to have a conversation over tests and behaviors, having in sight the possibility of including the business people and thus having the full *Three Amigos* involved in the process. Many customers think in requirements first, then development and only at the end in testing. It is vital to approximate these development stages and especially the **testing phase should be done alongside development**. In fact, testing should be a part of the development process and developers themselves should test software and know how to do it properly. BDD can take a huge part in this, bringing the business into development, always with the aim of “building things right”, creating tests that check if the software is correct and above all if it works as the customer wants (acceptance criteria). Discovering errors as early as possible to avoid regressions can have a great impact given the speed of development. The *BDDFramework* is a good starting point. This tool allowed, among other things to establish a test standard for the OutSystems language, making tests easier to understand, even for people who had not designed them. However, to support the BDD

process it still needed some changes, like being accessible to all stakeholders and some automation mechanisms.

We wanted a framework capable of promoting the BDD process, helping to bring the software stages together as well as the people involved in them, promoting team enablement and automation. Therefore, after understanding the principles of BDD and how it could be used to enhance the characteristics of the OutSystems language, we conducted a set of interviews to various people in the OutSystems domain. With these we could verify what was missing in the *BDDFramework* and what could be improved if we wanted to have a BDD process supporting framework as there is in other languages: it was **restricted to technical stakeholders** (developers and testers) and **needed automation mechanisms** at the time of scenario generation (with the *BDDFramework* it was all done by hand from the creation of test screens, to the logic that will implement the steps, including their reuse which goes against the high speed of development) and also, the lack of integration with management tools and test reporting mechanisms. These problems led us to investigate other tools in order to understand how we could reverse the situation.

Among several alternatives that came up, we opted for the development of a component that **uses the *BDDFramework***, without modifying it, and that will use the widgets already provided by it to fill in the scenarios, but allowing to automate the process, from the text files where the scenarios are described. We chose to extend the *BDDFramework* because we wanted to **stay in the OutSystems domain**, taking advantage of the good things the *BDDFramework* offers and using a tool that can continue to be used on its own in various contexts and already known by the community. The developed component currently functions as a **command in Service Studio**. With one click, it automates the process of screen creation and test logic, detecting equal steps to which it assigns the same (centralized) implementation, and also dealing with parameters.

To validate that the objectives were successfully achieved we performed tests with real users (OutSystems developers) to understand if the prototype we implemented served the purposes it had proposed to solve. The tests results suggest that the prototype was successful in improving several aspects, when compared to the *BDDFramework*: **speed, ease, correctness and usability**. Of course, it would be ideal if we had an evaluation that involved all the stages of the process in a larger time frame and with teams using the prototype for their daily testing activities, with all the *Three Amigos* involved in conversations leading to a joint interpretation and decomposition of functionalities, later described in the form of scenarios. However, this was not possible given available resources, but we are still pleased with the results obtained.

We started this dissertation with the following objective:



Realizing the characteristics of Behavior-Driven Development and Low Code technologies, we want to develop a test automation framework in the OutSystems domain that enables the BDD process technologically, having as a starting point the existing *BDDFramework*.

Based on the results we are in position to state that the objective was successfully achieved.

## 7.2 Contributions

This dissertation includes the following contributions:

- A **set of requirements** that serves as a basis for the development of BDD testing automation frameworks;
- A set of interviews conducted with stakeholders from various fields related to OutSystems and testing, which resulted in a **compilation of needs associated with the testing process** and a **detailed analysis of the *BDDFramework***. They also produced various testimonials and different views on testing as well as real case examples of people using BDD in OutSystems with other tools;
- A component that works as a **command in Service Studio** and uses some of the *BDDFramework* features to support the BDD process. This takes advantage of all that *BDDFramework* has to offer but allows the use of **feature files described outside of Service Studio** and automates the process of scenario generation and test logic creation, dealing with parameterization and reuse of scenarios. The developers only have to worry about code implementation when new scenarios are introduced into the system;
- Establishment of a **structured test design standard** in OutSystems, as the component automatically organizes scenarios in the same way and respecting the recommended practices for BDD test design in OutSystems, by organizing widgets and actions within the test projects and through **naming conventions** designed to make the work easier for developers and taking this weight off them;
- **Performance, ease, correctness and usability** studies for the *BDDFramework* and the developed Prototype;

## 7.3 Future Work

Regarding possible future extensions to this work, we highlight the following:

- The component could be executed from the command line or through an **executable app** outside of Service Studio. This would further enable business people to integrate into the process as they could run the tests themselves and view the execution report. Another important aspect is that with this, we would be able to automatically generate updated eSpaces every time feature files are updated, thus streamlining the process of test specification from the business person to the developers. In OutSystems this can be achieved via command line command;
- More integration with test management mechanisms to ensure greater test coverage and more information on the implemented tests. Integration is currently possible through the BDDFramework REST API (also available in the Prototype) which provides some information such as the number of tests that are failing and information about them in execution time but this information may be more complete to allow for better process management by product owners, to have an overall view of the system;
- Addition mechanisms for scenario-level test data management to allow direct integration of outlines, aggregations, among others. Some of these aspects are facilitated by the language and the *BDDFramework* (setup and teardown hooks) but can still be improved to allow easier Data-Driven Testing integration;
- The Prototype could give the user more freedom and allow **more block customization** right from the feature file, without having to do it through Service Studio, such as renaming Web Blocks. This may be possible by adding tags to the text file to allow titles to be added to blocks. These would be optional and the user only used if he wanted;
- There could be more **reciprocal communication between feature files and Service Studio**. As things are now done, power is on the side of text files, but it can be practical from the developer experience point of view, that a scenario, when deleted from Service Studio, is also from the text file with the scenarios. Currently, the way things are done this does not happen and in the next run the deleted scenario will be reinserted, thus forcing the developer to go to the feature file to permanently delete it.

## BIBLIOGRAPHY

- [1] *Acceptance Test Driven Development (ATDD)*. URL: <https://www.agilealliance.org/glossary/atdd/> (visited on 02/07/2019).
- [2] G. Adzic. *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*. Neuri Limited, 2009. ISBN: 9780955683619. URL: <https://books.google.pt/books?id=SZtkPgAACAAJ>.
- [3] A. Alliance. *Exploratory Testing*. visited on 15/06/2019. URL: <https://www.agilealliance.org/glossary/exploratory-testing/>.
- [4] S. W. Ambler. *User Stories: An Agile Introduction*. URL: <http://www.agilemodeling.com/artifacts/userStory.htm>.
- [5] Atlassian. *Jira Software*. URL: <https://www.atlassian.com/software/jira>.
- [6] V. R. Basili and H. D. Rombach. "The TAME project: towards improvement-oriented software environments." In: *IEEE Transactions on Software Engineering* 14.6 (1988), pp. 758–773. DOI: 10.1109/32.6156.
- [7] K. Beck. "Test-Driven Development By Example." In: (2002).
- [8] A. Bertolino and E Marchelli. "A Brief Essay on Software Testing." In: *Software Engineering 1* (2005).
- [9] L. P. Binamungu, S. M. Embury, and N. Konstantinou. "Maintaining behaviour driven development specifications: Challenges and opportunities." In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 175–184. DOI: 10.1109/SANER.2018.8330207.
- [10] L. P. Binamungu, S. M. Embury, and N. Konstantinou. "Behavior-Driven Requirements Traceability via Automated Acceptance Tests." In: *2017 IEEE 25th International Requirements Engineering Conference Workshops* (2017).
- [11] L. P. Binamungu, S. M. Embury, and N. Konstantinou. *Detecting Duplicate Examples in Behaviour Driven Development Specifications*. Tech. rep. 2018.
- [12] S. Braams. *Developing a Software Quality Framework for Low-Code Model Driven Development Platforms Based on Behaviour Driven Development Methodology*. Tech. rep. 2017.

## BIBLIOGRAPHY

---

- [13] Brian. *Best Automation Testing Tools for 2019*. URL: <https://medium.com/@briananderson2209/best-automation-testing-tools-for-2018-top-10-reviews-8a4a19f664d2>.
- [14] J. Brooke et al. "SUS-A quick and dirty usability scale." In: *Usability evaluation in industry* 189.194 (1996).
- [15] R. Bussenot, H. Leblanc, and C. Percebois. "Orchestration of Domain Specific Test Languages with a Behavior Driven Development approach." In: 2018, pp. 431–437. DOI: 10.1109/SYSOSE.2018.8428788.
- [16] R. N. Charette. "Why software fails [software failure]." In: *IEEE Spectrum* 42.9 (2005), pp. 42–49. DOI: 10.1109/MSPEC.2005.1502528.
- [17] *Cucumber*. URL: <https://cucumber.io/>.
- [18] *Cucumber Documentation*. URL: <https://docs.cucumber.io/>.
- [19] *Differences between TDD, ATDD and BDD*. URL: <https://gaboesquivel.com/blog/2014/differences-between-tdd-atdd-and-bdd/>.
- [20] E. Çelik, S. Eren, E. Çini and Keleş. "Software test automation and a sample practice for an enterprise business software." In: *International Conference on Computer Science and Engineering (UBMK)* (2017).
- [21] M. Fowler. *Subcutaneous Testing*. 2011. URL: <https://martinfowler.com/bliki/SubcutaneousTest.html>.
- [22] M. Fowler and H. Vocke. *The Practical Test Pyramid*. 2018. URL: <https://martinfowler.com/articles/practical-test-pyramid.html> (visited on 02/01/2019).
- [23] A. Ghahrai. *Why Selenium and Cucumber Should Not Be Used Together*. 2019. URL: <https://www.testingexcellence.com/selenium-and-cucumber-ui-automation-challenges/>.
- [24] *Gherkin Reference*. URL: <https://docs.cucumber.io/gherkin/reference/>.
- [25] N. N. Group. *Between-Subjects vs. Within-Subjects Study Design*. 2018. URL: <https://www.nngroup.com/articles/between-within-subjects/> (visited on 09/10/2019).
- [26] S. Hardy. *Continuous Delivery*. 2008. URL: <https://continuousdelivery.com/>.
- [27] S. G. Hart. "NASA-task load index (NASA-TLX); 20 years later." In: *Proceedings of the human factors and ergonomics society annual meeting*. Vol. 50. 9. Sage publications Sage CA: Los Angeles, CA. 2006, pp. 904–908.
- [28] D. Homan. "Cost Benets Analysis of Test Automation." In: (1999).
- [29] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. "Reporting Experiments in Software Engineering." In: Jan. 2008, pp. 201–228. DOI: 10.1007/978-1-84800-044-5\_8.
- [30] R. Jeffries and G. Melnik. "TDD–The art of fearless programming." In: *Ieee Software* 24.3 (2007), pp. 24–30.

- 
- [31] L. Karam. *A guide to UI Testing*. URL: <https://apiumhub.com/tech-blog-barcelona/ui-testing/>.
- [32] Kent Beck. *Test Driven Development: By Example*. 2003.
- [33] K. Kudryashov. *The Beginners Guide to BDD*. Tech. rep. 2015. URL: <https://inviqa.com/blog/bdd-guide>.
- [34] T. Linchpin. *A Beginners Guide To The Agile Method & Scrums*. 2019. URL: <https://linchpinseo.com/the-agile-method/> (visited on 07/29/2019).
- [35] M. Ehmer and F. Khan. "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques." In: 3 (2012), pp. 12–15.
- [36] *Manifesto for Agile Software Development*. 2001. URL: <https://agilemanifesto.org/>.
- [37] B. Marick. *Exploration Through Example*. URL: <http://www.exampler.com/old-blog/2003/08/21/>.
- [38] W. Matt and A. Hellesoy. *The Cucumber Book*. Ed. by P. Bookshelf. 2012.
- [39] M. Meyer. "Continuous Integration and Its Tools." In: *IEEE Software* 31.3 (2014), pp. 14–16. ISSN: 0740-7459. DOI: 10.1109/MS.2014.58.
- [40] N. M. Minhas, K. Petersen, N. B. Ali, and K. Wnuk. "Regression Testing Goals - View of Practitioners and Researchers." In: *Asia-Pacific Software Engineering Conference Workshops (APSECW)* (2017).
- [41] C. Nagle. *Test Automation Frameworks*. URL: <http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm>.
- [42] Nasa. *Nasa Tlx - Task Load Index*. URL: <https://humansystems.arc.nasa.gov/groups/TLX/>.
- [43] A. S. Nezhad, J. J. Lukkien, and R. H. Mak. "Behavior-driven Development for Real-time Embedded Systems." In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2018, pp. 59–66. DOI: 10.1109/ETFA.2018.8502653.
- [44] D. North. *JBehave*. URL: <https://jbehave.org/>.
- [45] D. North. *Introducing BDD*. 2006. URL: <https://dannorth.net/introducing-bdd/>.
- [46] D. North. "Agile specifications, BDD and Testing eXchange." In: 2009.
- [47] OutSystems. *Architecture*. URL: <https://www.outsystems.com/evaluation-guide/architecture/>.
- [48] OutSystems. *How does OutSystems support testing and quality assurance?* URL: <https://www.outsystems.com/evaluation-guide/how-does-outsystems-support-testing-and-quality-assurance/?origin=d>.

- [49] OutSystems. *Internal documentation*.
- [50] OutSystems. *Static Entities*. URL: [https://success.outsystems.com/Documentation/11/Developing\\_an\\_Application/Use\\_Data/Data\\_Modeling/Static\\_Entities](https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Data/Data_Modeling/Static_Entities).
- [51] OutSystems. *Table Records Widget*. URL: [https://success.outsystems.com/Documentation/11/Reference/OutSystems\\_Language/Web\\_Interfaces/Designing\\_Screens/Table\\_Records\\_Widget](https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Web_Interfaces/Designing_Screens/Table_Records_Widget).
- [52] OutSystems. *Test Automator Team*. URL: <https://www.outsystems.com/forge/component-details/82/Test+Automator/>.
- [53] OutSystems. *Unit and Regression Testing with OutSystems*. URL: <https://www.outsystems.com/evaluation-guide/unit-and-regression-testing-with-outsystems/>.
- [54] *OutSystems tools and components*. URL: [https://success.outsystems.com/Evaluation/Architecture/1{\\\\_}OutSystems{\\\\_}Platform{\\\\_}tools{\\\\_}and{\\\\_}components](https://success.outsystems.com/Evaluation/Architecture/1{\\_}OutSystems{\\_}Platform{\\_}tools{\\_}and{\\_}components) (visited on 01/07/2019).
- [55] S. Pratik. *How to Overcome UI Automation Testing Challenges?* 2017. URL: <https://www.utest.com/articles/how-to-overcome-ui-automation-testing-challenges?comments=3>.
- [56] J. Proença. *BDDFramework*. URL: <https://www.outsystems.com/forge/1201/>.
- [57] J. Proença. *How to Automate BDD Testing in OutSystems, Part 1: An Introduction to the BDDFramework*. 2019. URL: <https://www.outsystems.com/blog/posts/intro-bddframework-testing/>.
- [58] J. Proença. *How to Automate BDD Testing in OutSystems, Part 2: The BDDFrameworks Test-Execution REST API*. 2019. URL: <https://www.outsystems.com/blog/posts/automate-bddframework-testing/>.
- [59] J. Proença. *How to Automate BDD Testing in OutSystems, Part 3: Data-Driven API Tests With the BDDFramework*. 2019. URL: <https://www.outsystems.com/blog/posts/bddframework-data-testing/>.
- [60] J. S. Rajal and S. Sharma. "A Review on Various Techniques for Regression Testing and Test Case Prioritization." In: 2015.
- [61] P. Rayner. *BDD is a Centered Community Rather than a Bounded Community*. URL: <http://thepaulrayner.com/bdd-is-a-centered-rather-than-a-bounded-community/>.
- [62] M. Rehkopf. *Sprints*. URL: <https://www.atlassian.com/agile/scrum/sprints>.
- [63] S. Rose, M. Wynne, and A. Hellesøy. *The Cucumber for Java Book: Behaviour-driven Development for Testers and Developers*. Pragmatic programmers. Pragmatic Bookshelf, 2015. Chap. 1, pp. 7–8. ISBN: 9781941222294. URL: <https://books.google.pt/books?id=zQ2voQEACAAJ>.

- 
- [64] R.Santos, C. de Magalhaes, J.Correia-Neto, F.Silva, and L.Capretz. “Would You Like to Motivate Software Testers? Ask Them How.” In: *Electrical and Computer Engineering Publications* 114 (2017), pp. 95–104.
- [65] J. R. Rymer and R. Koplowitz. “The Forrester Wave: Low-Code Development Platforms For ADD Professionals, Q1 2019.” In: (2019).
- [66] K. Schwaber. “SCRUM Development Process.” In: *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 1995, pp. 117–134.
- [67] *SeleniumHQ Browser Automation*. visited on 05/06/2019. URL: <https://www.seleniumhq.org/>.
- [68] T. R. Silva. *Definition of a behavior-driven model for requirements specification and testing of interactive*. Tech. rep. Toulouse, France: Université Paul Sabatier, 2016.
- [69] J. F. Smart. *BDD In Action: Behavior-Driven Development for the whole software lifecycle*. 2014.
- [70] C. Solis and X. Wang. “A study of the characteristics of behaviour driven development.” In: *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE. 2011, pp. 383–387.
- [71] I. Sommerville. *Software Engineering*. 9th ed. 2011, chapters 3, 4.
- [72] *SpecFlow*. URL: <https://specflow.org/>.
- [73] I. The Standish Group International. “Chaos Report 2015.” In: (2015). URL: [https://www.standishgroup.com/sample\\_research\\_files/CHAOSReport2015-Final.pdf](https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf).
- [74] A. P. Veiga. *Project Success in Agile Development Projects*. Tech. rep. 2017.
- [75] M. Winteringham. *The difference between ATDD and BDD*. visited on 05/06/2019. URL: <https://www.hindsightsoftware.com/blog/atdd-vs-bdd>.
- [76] Xpand IT. *Xray Test Management for Jira*. URL: <https://marketplace.atlassian.com/apps/1211769/xray-test-management-for-jira?hosting=cloud{\&tab=overview>.







## APPENDIX 1 INTERVIEW SCRIPTS

### BDD Framework (Interview)

This survey is intended to make a small study about the BDD Framework from OutSystems

Before using the BDD Framework, how was the testing process?  
Can you describe it?

Your answer

---

Why did you decide to use this tool?

Your answer

---

Which type of tests did you perform with it?

Your answer

---

Is it useful for the development and testing processes? If so, in  
which way? Give a brief explanation

Your answer

---

Is there any other tool out there you think suits better the  
OutSystem's development process? If so, which one? Give a  
brief explanation

Your answer

---

What do you think could be improved in the Framework? Which  
additional features do you expect?

Your answer

---

SUBMIT

Figure A.1: Script for the interviewees who only had contact with the BDDFramework

## Applying BDD in the development process (Interview)

This survey is intended to make a small study about how the BDD is currently seen in the IT market, in several different contexts

Why did you decide to use BDD and why did you think it would be better to use this approach rather than others?

Your answer

---

Were the objectives achieved? If not, why?

Your answer

---

Do you apply BDD to the entire development process or only to specific parts? Where?

Your answer

---

Could you please describe how your BDD process works? Which steps do you take and how it works.

Your answer

---

Are you using any framework to support the BDD process?

Yes

No

If so, which one?

Your answer

---

Figure A.2: Script for the interviewees who had contact with both the BDD process and the BDDFramework

APPENDIX



APPENDIX 2 EXPERIMENT INTRODUCTORY  
SCRIPTS

## Atividade Experimental

### **Contextualização:**

A área de *Software Testing* no domínio *Low Code* é ainda uma área que levanta algumas questões e não é consensual. O caso concreto da OutSystems não é exceção. Como foi verificado nesta investigação, esta fase ainda é feita um pouco “como dá jeito” a cada equipa, sem haver uma forma standard de abordar o problema.

Foi neste âmbito que surgiu a *BDDFramework* (2016), um componente criado para permitir a automação de testes em linguagem OutSystems, utilizando a sintaxe *Gherkin* à semelhança de outras ferramentas de apoio à prática do *Behaviour-Driven Development* (BDD). No entanto, e apesar do seu nome, a *BDDFramework* não foi desenvolvida com o intuito de apoiar a prática de BDD, até porque esta não é feita ainda no contexto OutSystems, mas sim apenas com o intuito de automatizar e documentar testes utilizando o *Gherkin*.

Dadas as características da linguagem, nós temos indicações e partimos da premissa que a prática de BDD pode ser vantajosa para a empresa e queremos potenciar isso de um ponto de vista tecnológico. Foi nesta base que desenvolvemos um protótipo para esse efeito no contexto desta tese. Este componente desenvolvido utiliza a já existente *BDDFramework* mas foi pensado para, à semelhança de ferramentas como o *Cucumber* e o *SpecFlow*, potenciar a realização da prática de um ponto de vista tecnológico e no contexto OutSystems, acrescentando aquilo que nós considerámos essencial e indispensável para a adoção da ferramenta e complemento do processo de teste.

Nesta atividade experimental é pedido ao participante que resolva 2 pequenas questões, com tempo limitado, utilizando uma de duas abordagens: a antiga, que utiliza exclusivamente a *BDDFramework* tal e qual como ela existe atualmente para OutSystems e a nova, que utiliza o componente desenvolvido.

### **Behavior-Driven Development:**

Um dos principais problemas que assombram o desenvolvimento de software e levam ao seu fracasso é a falta de comunicação entre todos os *stakeholders* envolvidos nos projetos. Muitas vezes os *developers* não entendem aquilo que é pedido pelo negócio e o negócio não sabe descrever as funcionalidades e o seu valor nem as dificuldades associadas àquilo que está a pedir aos *developers*.

O *Behaviour-Driven Development* propõe resolver isto através de conversas, o mais cedo possível no ciclo de desenvolvimento das aplicações. Destas conversas resultam a descrição do comportamento das aplicações numa linguagem textual estruturada perceptível por todos os *stakeholders* (técnicos ou não) – o *Gherkin*.

Os cenários *Gherkin* apresentam a estrutura *Given, When, Then*, cláusulas que representam respetivamente o estado inicial da aplicação, a ação e o resultado esperado da mesma no sistema.

Figure B.1: Introductory guide for Approach 1 (*BDDFramework*), page 1.

---

A cláusula **Given** representa o estado inicial que tem de ser verificado antes da ação acontecer.

A cláusula **When** representa uma ação específica que vai despoletar o acontecimento descrito na cláusula **Then**.

A cláusula **Then** representa os resultados que são esperados no sistema através da realização da ação/evento.

Esta estrutura **auto-descritiva** dos cenários Gherkin faz com que estes constituam a base ideal para a automação de testes e uma forma de documentação dos sistemas.

#### **BDDFramework:**

A *BDDFramework* é um componente que permite descrever cenários *Gherkin* para a automação de testes em linguagem OutSystems. Este componente disponibiliza para o utilizador um conjunto de *templates* para a descrição e preenchimento dos cenários *Gherkin*, dentro do Service Studio.

Dentro de cada cenário é pedida uma descrição do mesmo e ainda há espaço para colocar os *BDDsteps* que vão preencher a estrutura do Gherkin em cada cenário, nos respetivos *placeholders* (*Given*, *When* e *Then*). Cada *BDDstep* é composto por uma descrição textual e é implementado por uma *Screen Action* que deve conter o código necessário para a sua implementação. Existem ainda ao dispor dos utilizadores *BDDsteps* de *setup* e *teardown*, que correspondem a lógicas que podem ser executadas antes da realização do teste, ou após o mesmo, respetivamente. Existe ainda uma biblioteca com vários tipos de asserções (de valor, positivos, negativos ou *default*). É possível também colocar em cada página o resultado final da execução de todos os cenários presentes na página e neste espaço é mostrado quantos cenários do total de cenários presentes na página é que foram executados com sucesso (isto é, todos os steps tiveram o resultado pretendido) quando a aplicação é publicada e executada.

Figure B.2: Introductory guide for Approach 1 (*BDDFramework*), page 2.

Tese de Mestrado - *Applying Behavior Driven Development Practices and Tools to Low-Code Technology*

## Atividade Experimental 2

### Contextualização:

A área de *Software Testing* no domínio *Low Code* é ainda uma área que levanta algumas questões e não é consensual. O caso concreto da OutSystems não é exceção. Como foi verificado nesta investigação, esta fase ainda é feita um pouco “como dá jeito” a cada equipa, sem haver uma forma standard de abordar o problema.

Foi neste âmbito que surgiu a *BDDFramework* (2016), um componente criado para permitir a automação de testes em linguagem OutSystems, utilizando a sintaxe *Gherkin* à semelhança de outras ferramentas de apoio à prática do *Behaviour-Driven Development* (BDD). No entanto, e apesar do seu nome, a *BDDFramework* não foi desenvolvida com o intuito de apoiar a prática de BDD, até porque esta não é feita ainda no contexto OutSystems, mas sim apenas com o intuito de automatizar e documentar testes utilizando o *Gherkin*.

Dadas as características da linguagem, nós temos indicações e partimos da premissa que a prática de BDD pode ser vantajosa para a empresa e queremos potenciar isso de um ponto de vista tecnológico. Foi nesta base que desenvolvemos um protótipo para esse efeito no contexto desta tese. Este componente desenvolvido utiliza a já existente *BDDFramework* mas foi pensado para, à semelhança de ferramentas como o *Cucumber* e o *SpecFlow*, potenciar a realização da prática de um ponto de vista tecnológico e no contexto OutSystems, acrescentando aquilo que nós considerámos essencial e indispensável para a adoção da ferramenta e complemento do processo de teste.

Nesta atividade experimental é pedido ao participante que resolva 2 pequenas questões, com tempo limitado, utilizando uma de duas abordagens: a antiga, que utiliza exclusivamente a *BDDFramework* tal e qual como ela existe atualmente para OutSystems e a nova, que utiliza o componente desenvolvido.

### Behavior-Driven Development:

Um dos principais problemas que assombram o desenvolvimento de software e levam ao seu fracasso é a falta de comunicação entre todos os *stakeholders* envolvidos nos projetos. Muitas vezes os *developers* não entendem aquilo que é pedido pelo negócio e o negócio não sabe descrever as funcionalidades e o seu valor nem as dificuldades associadas àquilo que está a pedir aos *developers*.

O *Behaviour-Driven Development* propõe resolver isto através de conversas, o mais cedo possível no ciclo de desenvolvimento das aplicações. Destas conversas resultam a descrição do comportamento das aplicações numa linguagem textual estruturada perceptível por todos os *stakeholders* (técnicos ou não) – o *Gherkin*.

Os cenários *Gherkin* apresentam a estrutura *Given, When, Then*, cláusulas que representam respetivamente o estado inicial da aplicação, a ação e o resultado esperado da mesma no sistema.

Figure B.3: Introductory guide for Approach 2 (*Prototype*), page 1.

---

A cláusula **Given** representa o estado inicial que tem de ser verificado antes da ação acontecer.

A cláusula **When** representa uma ação específica que vai despoletar o acontecimento descrito na cláusula **Then**.

A cláusula **Then** representa os resultados que são esperados no sistema através da realização da ação/evento.

Esta estrutura **auto-descriptiva** dos cenários Gherkin faz com que estes constituam a base ideal para a automação de testes e uma forma de documentação dos sistemas.

#### **Componente desenvolvido no contexto da tese:**

O componente que foi desenvolvido no contexto desta tese utiliza a *BDDFramework*, permitindo a descrição dos cenários em ficheiros de texto ao invés de diretamente no *Service Studio*, como acontecia na abordagem antiga, de forma a permitir a integração do negócio na descrição dos cenários (sem este ter de estar no *Service Studio*). Além de permitir esta integração, este componente gera automaticamente: os ecrãs de teste (1 *Web Screen* por cada funcionalidade) e insere dentro dos mesmos os blocos correspondentes a cada cenário de teste (cada *Web Block* contém lá dentro o template do cenário BDD devidamente preenchido) e ainda as lógicas que vão implementar os cenários de teste, tudo a partir desse ficheiro de texto. Todas as **steps/frases do Gherkin iguais são reaproveitados** (daí ser aconselhado que se tire proveito disto na descrição dos cenários), pois são criadas *Server Actions* centralizadas para as lógicas que os implementam, ao invés da abordagem antiga onde estes tinham de ser reaproveitados à mão. Também foi implementada a **deteção de parâmetros nas frases** (através da utilização de aspas) e a inferência de tipos, de forma a reduzir e automatizar ao máximo o trabalho do developer. No final, podemos ver o resultado da execução dos testes quando a aplicação é publicada e executada.

Figure B.4: Introductory guide for Approach 2 (*Prototype*), page 2.







## ANNEX 1 NASA TASK LOAD INDEX QUESTIONNAIRE

### NASA Task Load Index

*Hart and Staveland's NASA Task Load Index (TLX) method assesses work load on five 7-point scales. Increments of high, medium and low estimates for each point result in 21 gradations on the scales.*

| Name  | Task | Date |
|---|------|------|
| <p><b>Mental Demand</b>      How mentally demanding was the task?</p> <p>Very Low      Very High</p>                                |      |      |
| <p><b>Physical Demand</b>      How physically demanding was the task?</p> <p>Very Low      Very High</p>                            |      |      |
| <p><b>Temporal Demand</b>      How hurried or rushed was the pace of the task?</p> <p>Very Low      Very High</p>                   |      |      |
| <p><b>Performance</b>      How successful were you in accomplishing what you were asked to do?</p> <p>Perfect      Failure</p>      |      |      |
| <p><b>Effort</b>      How hard did you have to work to accomplish your level of performance?</p> <p>Very Low      Very High</p>     |      |      |
| <p><b>Frustration</b>      How insecure, discouraged, irritated, stressed, and annoyed were you?</p> <p>Very Low      Very High</p> |      |      |

Figure I.1: NASA-TLX questionnaire.





## ANNEX 2 SYSTEM USABILITY SCALE QUESTIONNAIRE

|  | Strongly<br>disagree |   |   |   | Strongly<br>agree |
|--|----------------------|---|---|---|-------------------|
| 1. I think that I would like to use this system frequently                                   | 1                    | 2 | 3 | 4 | 5                 |
| 2. I found the system unnecessarily complex  | 1                    | 2 | 3 | 4 | 5                 |
| 3. I thought the system was easy to use  | 1                    | 2 | 3 | 4 | 5                 |
| 4. I think that I would need the support of a technical person to be able to use this system | 1                    | 2 | 3 | 4 | 5                 |
| 5. I found the various functions in this system were well integrated                         | 1                    | 2 | 3 | 4 | 5                 |
| 6. I thought there was too much inconsistency in this system                                 | 1                    | 2 | 3 | 4 | 5                 |
| 7. I would imagine that most people would learn to use this system very quickly              | 1                    | 2 | 3 | 4 | 5                 |
| 8. I found the system very cumbersome to use   | 1                    | 2 | 3 | 4 | 5                 |
| 9. I felt very confident using the system  | 1                    | 2 | 3 | 4 | 5                 |
| 10. I needed to learn a lot of things before I could get going with this system              | 1                    | 2 | 3 | 4 | 5                 |

Figure II.1: System Usability Scale (SUS) questionnaire.





## ANNEX 3 TASK DESCRIPTIONS

### Desafio 1 – Área do Retângulo (15 min)

- Desenvolver **1 cenário de teste** para a aplicação “Área do retângulo”.
- Esta aplicação pede um **valor para o comprimento** e outro para a altura e quando se clica no botão para calcular a área é retornada a **área do retângulo**: comprimento x altura
- O cenário de teste deve **testar valores válidos**, por exemplo 20x30 e deve confirmar se o valor retornado é o correto, neste caso deveria ser 600 (20x30)
- Este desafio foca-se no **design e criação** de um novo cenário e implementação da sua lógica
- Nota: Foi criado um projeto em branco para a realização dos testes. Este já importa o projeto do Retângulo para podermos utilizar as *Server Actions* disponibilizadas no mesmo durante o teste. A *BDDFramework* encontra-se também importada.

Figure III.1: First task.

## Desafio 2 (Reaproveitamento) - Triângulo (10 minutos)

- Voltando à aplicação **Tipo de Triângulo** é agora pedido que se teste o triângulo **Escaleno**, no mesmo projeto que tinha sido iniciado na demonstração
- Nota: O projeto já importa o Triângulo para podermos utilizar as Server Actions disponibilizadas no mesmo durante este. A *BDDFramework* encontra-se também importada. O cenário que se encontra no projecto não precisa de ser removido
- Este desafio foca-se no **reaproveitamento** de lógicas já existentes.

Figure III.2: Second task.