Automatização de testes de Software para OutSystems: A importância das boas práticas no desenvolvimento e sua influência na automatização de testes

Joana Isabel Pereira Salgueiro

20180188

Orientadores

José Carlos Meireles Monteiro Metrôlho

Fernando Reinaldo Silva Garcia Ribeiro

Trabalho apresentado à Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Desenvolvimento de Software e Sistemas Interativos, realizada sob a orientação científica do Professor Doutor José Carlos Meireles Monteiro Metrôlho e do Professor Doutor Fernando Reinaldo Silva Garcia Ribeiro, do Instituto Politécnico de Castelo Branco.

junho de 2021

Composição do júri

Presidente do júri

Professor Doutor Arlindo Ferreira da Silva

Professor Adjunto da Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco

Vogais

Professor Doutor Luís Cláudio dos Santos Barradas

Professor Adjunto da Escola Superior de Gestão e Tecnologia do Instituto Politécnico de Santarém

Professor Doutor Fernando Sérgio Rodrigues Brito da Mota Barbosa

Professor Adjunto da Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco

Professor Doutor José Carlos Meireles Monteiro Metrôlho

Professor Adjunto da Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco

Agradecimentos

De forma breve quero agradecer a todas as pessoas que me apoiaram durante este processo de escrita e desenvolvimento da tese.

Primeiramente, agradecer aos meus dois orientadores, professores José Carlos Meireles Monteiro Metrôlho e Fernando Reinaldo Silva Garcia Ribeiro pela ajuda e apoio na escrita da tese, pela paciência e sobretudo por não me deixarem desistir nos momentos de maior desmotivação.

Em segundo lugar, agradecer à minha família pelo suporte e apoio que definitivamente me ajudou a não desistir e a manter forte e firme durante este processo.

Por último, mas não menos importante, agradecer à *OutSystems* e a todos os meus colegas de trabalho por estarem sempre disponíveis para esclarecer as minhas dúvidas, bem como pela documentação bastante completa disponível que foi muito útil no desenvolvimento desta tese.

Resumo

As plataformas *low-code* permitem acelerar o desenvolvimento de software através da redução de codificação manual, o que tem permitido desenvolver aplicações mais rapidamente, mas também que profissionais com formações distintas se tornem software *developers*. Isto tem possibilitado recrutar mais profissionais para as áreas das Tecnologias da Informação, requalificando alguns deles de diferentes áreas, mas, ao mesmo tempo, trazendo também para a área de desenvolvimento de software profissionais sem formação sólida nesta área. Embora os testes de software devam ser realizados em todas as aplicações, independentemente da linguagem de programação ou da plataforma usada, o que antes foi referido reforça, ainda mais, a necessidade de testar as aplicações desenvolvidas com plataformas de desenvolvimento *low-code*. Neste trabalho, estudou-se o processo de automatização de testes na plataforma *OutSystems* e o impacto que as boas práticas durante o desenvolvimento têm sobre o processo de automatização de testes. O foco incidiu nos níveis de teste de Componentes, Integração/API e Sistema/*End-to-End*. Os exemplos apresentados mostram que a implementação de boas práticas durante o processo de desenvolvimento pode ter uma influência significativa no processo de automatização de testes. No entanto, é importante avaliar se a carga de trabalho associada à implementação das boas práticas pode prejudicar alguns dos benefícios normalmente associados ao uso de uma plataforma *low-code*. Além disso, é importante considerar que nem todas as ferramentas de automatização de testes têm o mesmo comportamento. Assim, para saber se as boas práticas de desenvolvimento têm impacto na implementação e execução dos testes utilizou-se, como caso de estudo, uma aplicação desenvolvida em *OutSystems* para a qual foram elaborados 3 cenários de teste os quais foram implementados e executados com 3 ferramentas/*frameworks* de teste distintas. Para cada cenário foi analisado o processo de teste em termos de dificuldade, experiência, custo e tipos de testes executados por cada uma das ferramentas. De forma a validar a discussão resultante da execução dos cenários apresentados, foram também auscultados profissionais da área das Tecnologias de Informação, com experiência em desenvolvimento *OutSystems*, com o objetivo de analisar a perceção que estes têm sobre a importância dos testes de software no desenvolvimento *low-code* e sobre a influência das boas práticas de desenvolvimento no processo de automatização de testes.

Palavras-chave

OutSystems

Plataformas *Low-Code*

Qualidade de Software

BDDFramework

Ghost Inspector

Tricentis Tosca

Testes de Software

Automatização de Testes

Abstract

Low-code platforms allows software development by reducing manual coding, which has allowed the development of applications faster, but also for professionals with different backgrounds to become software developers. This has made it possible to recruit more professionals for the IT areas, requalifying some of them from different areas, but, at the same time, also brought professionals without solid training in this area to the area of software development. Although software testing must be performed on all applications, independent of the programming language or platform used, the aforementioned further reinforces the need to test applications developed with low-code platforms. In this work, we studied the test automation process in the OutSystems platform and the impact that best practices during development have on the test automation process. The focus was on Unit, Integration/API and System/End-to-End testing levels. The examples presented show that implementing best practices during the development process can have a significant influence on the test automation process. However, it is important to assess whether the workload associated with implementing best practices can undermine some of the benefits typically associated with using a low-code platform. Also, it is important to consider that not all test automation tools behave the same. Thus, to find out if the development with the best practices have an impact on the implementation and execution of the tests, an application developed in OutSystems was used as a case study for 3 test scenarios, which were implemented and executed with 3 different test frameworks. For each scenario, the testing process was analyzed in terms of difficulty, experience, cost and types of tests performed by each of the tools. To validate the discussion resulting from the execution of the scenarios presented, professionals in the IT area, with experience in OutSystems development, were also consulted their perception on the importance of software testing in low-code development and on the influence of the best practices on the test automation process.

Keywords

OutSystems

Low-code platforms

Software quality

Software testing

BDDFramework

Ghost Inspector

Tricentis Tosca

Test automation

Indíce

Índice de figuras

Lista de tabelas

Lista de abreviaturas, siglas e acrónimos

CSS	*Cascading Style Sheets*
ISTQB	*International Software Testing Qualifications Board*
TDD	*Test-Driven Development*
ATDD	*Acceptance Test-Driven Development*
BDD	*Behavior Driven Development*
GUI	Interface Gráfica do Utilizador
UI	*User Interface*
DDD	*Domain-Driven Design*
4LC	*4 Layer Canvas*
API	*Application Programming Interface*
JS	*Javascript*
XPath	*XML Path Language*
REST	*REpresentational State Transfer*
SOAP	*Simple Object Access Protocol*
DEV	Desenvolvimento
QA	Qualidade
CI	Integração Contínua
E2E	*End-to-End*
DOM	*Document Object Model*
URL	*Uniform Resource Locator*
HTTP	*Hypertext Transfer Protocol*
IP	*Internet Protocol*
JSON	*JavaScript Object Notation*

1. Introdução

As plataformas de desenvolvimento *low-code* são ambientes de desenvolvimento integrado com base visual, que incluem muitas das ferramentas e funcionalidades normalmente usadas para projetar, codificar, testar e gerir as suas aplicações [3]. Estas plataformas têm como objetivo facilitar e acelerar a criação e implementação de aplicações com o mínimo de codificação possível, o que faz com que o produto possa ser entregue num prazo menor. A *OutSystems* [1] é uma das plataformas de desenvolvimento *low-code* normalmente apontada como líder neste mercado. Esta plataforma permite desenvolver rapidamente aplicações *web* e *mobile* e oferece um conjunto de variados componentes que permitem construir ecrãs, fluxos lógicos, conexão com base de dados, consumir e expor serviços *REST/SOAP* de uma forma rápida e menos complicada.

Apesar de todas as facilidades de desenvolvimento com estas plataformas, é necessário que os projetos sejam entregues com o máximo de qualidade para que também os clientes fiquem satisfeitos com a plataforma. No entanto, e apesar de o desenvolvimento *low-code* ser frequentemente associado ao desenvolvimento sem erros, tal não é verdade e é necessário estudar as diversas estratégias e ferramentas de teste que melhor se adequem com estas plataformas. É também importante que as ferramentas de teste usadas se adequem aos processos ágeis, normalmente associados à utilização destas plataformas, e que permitam que os testes possam ser feitos também de uma forma simples e rápida e que não comprometam o tempo de desenvolvimento das aplicações. Esta tem sido aliás uma das questões que os clientes colocam à *OutSystems* no início dos projetos e que está relacionada com as alternativas existentes para a execução de testes em *OutSystems* de uma maneira simples e automatizada.

Foi esta realidade que fez com que neste trabalho se use a plataforma *OutSystems* e para a qual se pretende contribuir na área de testes de software, que como se sabe, é uma das etapas que muito contribui para o desenvolvimento de software de qualidade. No entanto, os testes de software em plataformas *low-code* são também uma das vertentes ainda pouco exploradas.

Embora atualmente a *OutSystems* possua uma ferramenta que permite realizar testes, a *BDDFramework* [2], esta não serve ainda todos os propósitos dos clientes e não oferece ainda uma boa performance. Por isso, importa estudar soluções existentes que ajudem a ultrapassar as lacunas antes referidas, e perceber quais as implicações que as boas práticas de desenvolvimento em *low-code* têm no processo de teste, de forma a utilizar estas ferramentas de teste de forma mais eficiente e a otimizar a sua utilização.

1.1. Objetivos

Neste trabalho pretende-se estudar ferramentas de teste de software que possam ser usadas para testes de software desenvolvido em *OutSystems*. É realizada uma comparação entre as diversas ferramentas de testes funcionais para aplicações *web*. O objetivo é identificar quais as boas práticas que os *developers* devem utilizar durante o desenvolvimento para que os testes possam ser mais facilmente executados de forma automatizada. Para avaliar a performance das ferramentas de teste estudadas, e as implicações da utilização de boas práticas de desenvolvimento em *OutSystems*, no processo de automatização de testes funcionais, são realizados testes numa aplicação desenvolvida em *OutSystems* a qual inclui, no seu desenvolvimento, diferentes componentes. São também realizados inquéritos a profissionais que trabalham com a plataforma *OutSystems* de forma a melhor compreender a utilização de ferramentas de teste e a sua aceitação por estes profissionais.

1.2. Cronograma

Tendo por base os objetivos deste projeto e de forma a concretizar cada um deles com sucesso, foi realizado um plano de trabalho, dividido em 7 fases, que se apresentam de seguida e na Tabela 1:

1. Estudo sobre qualidade de software e automatização de testes em plataformas de desenvolvimento *low-code* e em especial em *OutSystems*;
2. Estudo e análise de ferramentas de teste que permitam a automatização e execução de testes funcionais e UI em aplicações *OutSystems*;
3. Estudo das boas práticas de desenvolvimento em OutSystems e quais as suas implicações nas atividades de teste;
4. Identificação e análise da aplicação *OutSystems* a ser testada;
5. Conceção, implementação e automatização dos testes usando as ferramentas estudadas;
6. Análise dos resultados obtidos com os testes e identificação das ferramentas que apresentam melhor desempenho e das boas práticas a seguir pelos *developers*.
7. Escrita da dissertação.

Tabela 1 - Cronograma

	Nov.	Dez.	Jan.	Fev.	Mar.	Abr.	Mai.	Jun.	Jul.	Ago	Set.	Out.	Nov.	Dez.	Jan.	Fev.	Mar	Abr.	Mai.
1																			
2																			
3																			
4																			
5																			
6																			
7																			

1.3. Organização do documento

Este relatório está organizado em 7 capítulos.

Após este primeiro capítulo, onde é feita uma introdução e enquadramento ao trabalho, segue-se um segundo capítulo no qual são explicados vários conceitos necessários para a compreensão do restante documento como a qualidade de software, testes, tipos e níveis de teste e as suas utilidades, o que é o *low-code* e algumas plataformas *low-code* e, por fim, como são realizados os testes em *OutSystems* atualmente.

No terceiro capítulo, são apresentadas algumas ferramentas de testes que existem e que podem ser usadas para realizar testes em aplicações *web* desenvolvidas em *OutSystems* bem como vantagens e desvantagens de cada uma. Neste caso, as ferramentas são a *BDDFramework*, o *GhostInspector* e o *TricentisTosca*.

No quarto capítulo são apresentadas algumas das boas práticas existentes em *OutSystems* e que devem ser consideradas durante o desenvolvimento de todas as aplicações e são analisadas as suas implicações nas atividades de teste. São também dados exemplos de acordo com a plataforma *OutSystems*. Contudo, as práticas apresentadas podem ser aplicadas a qualquer linguagem de programação.

No quinto capítulo são descritos 3 cenários de teste que serão implementados e executados com as 3 ferramentas de teste selecionadas tendo em conta as boas práticas descritas no quarto capítulo. É também apresentada a aplicação onde serão efetuados os testes, aplicação esta desenvolvida em *OutSystems*. No final do capítulo é apresentada uma secção com uma análise dos resultados obtidos na execução dos testes com as ferramentas selecionadas de modo a serem tiradas algumas conclusões sobre as ferramentas e sobre os testes executados.

No sexto capítulo, é apresentado um questionário feito a pessoas com experiência na área das tecnologias da informação e testes de modo a obter alguns resultados e a compreender o que estas pessoas acham sobre os testes e a sua importância.

Por fim, no sétimo capítulo, são apresentadas as conclusões do trabalho e são identificadas algumas linhas de trabalho futuro.

2. Testes de software e plataformas de desenvolvimento *low-code*

Neste capítulo serão apresentados conceitos e fundamentos sobre qualidade e testes de software. É também apresentada uma breve introdução às plataformas de desenvolvimento *low-code* e são abordadas especificidades do teste de software na plataforma *OutSystems*.

2.1. Conceitos

Antes de abordar o estudo dos testes de software é importante clarificar alguns dos conceitos que vão ser usados. Esta clarificação é relevante pois, existem diferenças quanto à terminologia usada consoante se segue um determinado autor ou determinada bibliografia. Assim, neste documento são usados os conceitos e terminologia adotados pelo ISTQB (*International Software Testing Qualifications Board*) [3], nomeadamente:

- Erro: uma ação humana que produz um resultado incorreto.
- Defeito (ou *bug*): uma imperfeição num componente ou sistema que pode causar falha no desempenho do componente ou do sistema, por exemplo uma definição de dados incorreta. Um defeito, se encontrado durante a execução, pode causar a falha do componente ou do sistema.
- Falha: resultado da execução de um componente ou sistema diferente do resultado esperado.

Durante este documento são também usados os termos de desenvolvedor/*developer* para referir a pessoa que desenvolve programas, sistemas e/ou componentes e os termos testador/*tester* para referir a pessoa que testa os programas, sistemas e/ou componentes desenvolvidos pelos *developers*.

2.2. Qualidade de Software

No geral, a qualidade de software está relacionada com a satisfação dos requisitos [4]. Mas como é que são definidos os requisitos? Na metodologia *Agile*, os requisitos são definidos em *user stories* [5] - descrições curtas e simples de recursos necessários para o desenvolvimento de um projeto, normalmente contadas da perspetiva da pessoa que pretende o recurso, normalmente o cliente – e podem ser classificados como funcionais e não funcionais. No entanto, não é apenas o facto do software cumprir um determinado requisito que se pode dizer que estamos perante um software de boa qualidade. Por exemplo, o João queria fazer uma viagem e decidiu utilizar um site para

reservar o voo e o hotel. O João concluiu a tarefa com sucesso, mas na verdade não ficou satisfeito com a qualidade do *site* que utilizou. Isto pode acontecer devido aos requisitos não funcionais do software, requisitos estes que muitas vezes são achados como não importantes para o software.

A qualidade de software de um produto está normalmente associada a vários fatores. A norma ISO/IEC 25010 [6] define um conjunto de categorias com o objetivo de avaliar a qualidade do software, como é possível ver no diagrama da Figura 1.

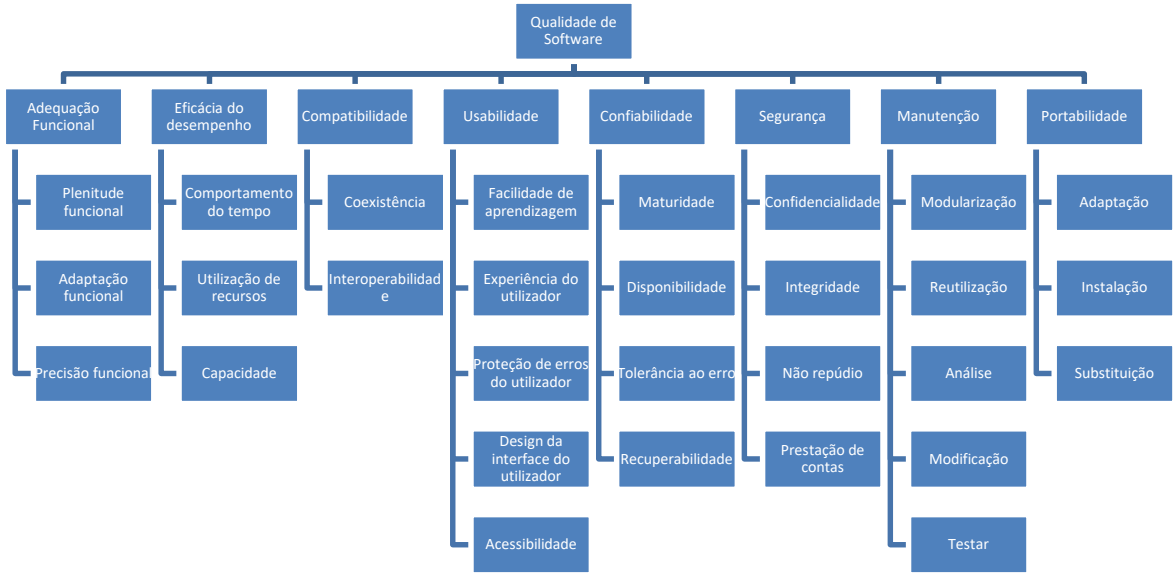

Figura 1 – Modelo proposto pela ISO/IEC 25010.

Este modelo define a avaliação da qualidade de um produto de software e considera 8 categorias que representam características de qualidade que os softwares devem ter de forma a alcançar um nível elevado de qualidade. Uma breve descrição de cada uma das categorias permite melhor conhecer o que é pretendido em cada uma.

Adequação funcional: é avaliado se o produto ou se o sistema fornece as funcionalidades que atendem às necessidades declaradas.

Plenitude funcional: avalia se o conjunto de funcionalidades cobre todas as tarefas e objetivos especificados pelo utilizador.

Adaptação funcional: avalia se as funcionalidades facilitam a realização das tarefas e dos objetivos especificados.

Precisão funcional: avalia se o produto ou o sistema fornece os resultados corretos com o grau de precisão necessário.

Eficácia do desempenho: é avaliado o desempenho relativo à quantidade de recursos, como por exemplo, configuração do software e *hardware* do sistema, utilizados sob condições estabelecidas.

Comportamento do tempo: avalia se os tempos de resposta, processamento e taxas de transferência de um produto ou sistema atendem aos requisitos.

Utilização de recursos: avalia se as quantidades e os tipos de recursos utilizados por um produto ou sistema atendem aos requisitos aquando da execução das suas funções.

Capacidade: avalia se os limites máximos de um parâmetro de produto ou sistema, como por exemplo, número de utilizadores em simultâneo ou tamanho da base de dados, atendem aos requisitos.

Compatibilidade: é avaliado se um produto, sistema ou componente pode trocar informações com outros produtos, sistemas ou componentes e/ou podem executar as funções necessárias, compartilhando o mesmo ambiente de *hardware* ou software.

Coexistência: avalia se um produto ou sistema pode executar suas funções necessárias com eficiência, compartilhando um ambiente e recursos comuns com outros produtos, sem impactos negativos.

Interoperabilidade: avalia se dois ou mais sistemas, produtos ou componentes podem trocar informações e usar as mesmas.

Usabilidade: é avaliado se um produto ou sistema pode ser utilizado por utilizadores específicos para atingir determinadas metas com eficácia, eficiência e satisfação.

Facilidade de aprendizagem: avalia se um produto ou sistema pode ser facilmente usado por utilizadores específicos.

Experiência do utilizador: avalia se os utilizadores reconhecem um produto ou sistema como sendo apropriado para as suas necessidades.

Proteção contra erros do utilizador: avalia se um produto ou sistema protege os utilizadores contra erros.

***Design* da interface do utilizador:** avalia se a interface do produto ou do sistema permite uma interação agradável e satisfatória com o utilizador.

Acessibilidade: avalia se um produto ou sistema pode ser utilizado por pessoas com as mais variadas características e recursos para atingir uma meta especifica num contexto específico.

Confiabilidade: avalia se um sistema, produto ou componente executa funcionalidades especificas sob condições especificas e num determinado período de tempo.

Maturidade: avalia se um sistema, produto ou componente atende às necessidades de confiabilidade sob operação normal.

Disponibilidade: avalia se um sistema, produto ou componente está operacional e acessível quando for necessário para utilização.

Tolerância ao erro: avalia se um sistema, produto ou componente funciona conforme o esperado, apesar da presença de falhas hardware ou software.

Recuperabilidade: avalia se, no caso de uma interrupção ou falha, um produto ou sistema pode recuperar os dados diretamente afetados e restabelecer o estado desejado do sistema.

Segurança: é avaliado se um produto ou sistema protege informações e dados para que os utilizadores, outros produtos ou sistemas tenham o acesso a dados adequado aos seus tipos e níveis de autorização.

Confidencialidade: avalia se um produto ou sistema garante que os dados sejam acessíveis apenas aos utilizadores autorizados a ter acesso.

Integridade: avalia se um sistema, produto ou componente impede o acesso não autorizado, modificação de programas ou dados de computador.

Não repúdio: avalia se ações ou eventos ocorreram, de modo que os eventos ou ações não possam ser repudiados posteriormente.

Prestação de contas: avalia se as ações de uma entidade podem ser localizadas exclusivamente para a entidade.

Manutenção: é avaliado o grau de efetividade e eficiência com as quais um produto ou sistema pode ser modificado.

Modularidade: avalia se um sistema ou programa sofre o mínimo de impactos quando algum componente é alterado.

Reutilização: avalia se existem componentes que podem ser utilizados em mais que um sistema ou produto.

Análise: avalia o impacto de uma ou mais alterações a um produto ou sistema, diagnostica um produto ou sistema no que diz respeito a deficiências, causas de falhas ou defeitos.

Modificação: avalia se um produto ou sistema pode ser modificado sem a introdução de defeitos ou a redução da qualidade do produto existente.

Testar: avalia se os critérios de teste estabelecidos para um sistema, produto ou componente e se os testes podem ser realizados para determinar se esses critérios foram atendidos.

Portabilidade: é avaliado se um sistema, produto ou componente pode ser transferido de um hardware, software ou outro ambiente operacional com sucesso.

Adaptação: avalia se um produto ou sistema pode ser adaptado para hardware, software ou outros ambientes operacionais com sucesso.

Instalação: avalia se um produto ou sistema pode ser instalado e/ou desinstalado com sucesso num ambiente especificado com sucesso.

Substituição: avalia se um produto pode substituir outro produto de software especificado para a mesma finalidade no mesmo ambiente com sucesso.

Os testes de software podem contribuir de forma significativa para a melhoria da qualidade do produto de software nas várias categorias do modelo proposto pela ISO/IEC 25010.

2.3. Testes de Software

Os testes de software [7] são executados para determinar se o software corresponde às especificações que foram acordadas e se este é executado no ambiente pretendido. Os testes são usados para melhorar o software, ou sistema, que está em desenvolvimento. Estes podem ajudar a medir a qualidade do software em termos de defeitos encontrados, tanto relativamente a requisitos e características funcionais como aos requisitos não funcionais (p. ex. desempenho, confiabilidade, usabilidade, segurança, manutenção e portabilidade) [3].

É importante considerar os testes de software nas suas várias abordagens. Deve ser promovida a realização de testes estáticos, automatizados ou manuais, como as revisões de documentação do software ou do código, onde, neste caso, o código fonte não compilado é lido e analisado, normalmente pelos pares. Deve também ser promovida a realização de testes dinâmicos onde o software é executado na tentativa de encontrar defeitos no software e para que sejam feitas as diligências para que estes defeitos sejam corrigidos, melhorando assim a qualidade do produto. Mas é importante também que os testes de software realizados considerem as especificações funcionais, que definem o comportamento que o software deve ter, mas também as especificações não funcionais. O teste das características não funcionais não deve ser descurado pois são estas características que, muitas vezes, levam ao fracasso de um determinado software. Mesmo cumprindo as especificações formais, o software pode falhar por não satisfazer outras restrições como por exemplo, se o código ocupar muita memória será executado lentamente ou se o software não funciona de acordo com o esperado quando executado em alguns sistemas operativos.

2.3.1. Processo de teste de software

Os testes de software não se cingem à execução do teste propriamente dito. "As atividades de teste incluem planear e controlar, escolher as condições de teste, conceber e executar os casos de teste, verificar os resultados, avaliar os critérios de saída, manter informadas as partes envolvidas sobre todo o decorrer do processo de testes e do sistema sob teste, e finalizar ou completar as atividades de encerramento, logo que alguma das fases de teste tenha sido concluída. Além disso, os testes também incluem a revisão de documentos (incluindo o código fonte) e a realização de análise estática" [3].

Embora a parte mais visível dos testes seja a execução dos casos de teste e verificação dos resultados, para que esta fase seja eficaz e eficiente é necessário que seja efetuado um planeamento, a conceção dos casos de teste e só depois a sua execução e avaliação. O ISTQB [3] define um processo de testes que consiste em cinco fases principais:

- **Planeamento e controlo de testes:** nesta fase são definidos os objetivos dos testes e especificadas as atividades de teste, a fim de cumprir os objetivos e missão estabelecidos. O controlo de testes compara o progresso atual com o planeado, e reporta o seu estado, incluindo desvios do plano.
- **Análise e conceção de testes:** a análise e conceção de testes é a fase durante a qual os objetivos gerais dos testes são transformados em condições de teste e em casos de teste.
- **Implementação e execução de testes:** fase em que são especificados os procedimentos e *scripts* de teste através da combinação de casos de teste seguindo uma determinada ordenação e incluindo qualquer outra informação necessária à execução de teste, considera-se que a configuração do ambiente foi efetuada e executam-se os testes.
- **Avaliação de critério de saída e relatórios:** a avaliação do critério de saída é a atividade onde a execução do teste é avaliada face aos objetivos definidos. Esta deve ser efetuada para cada nível de teste.
- **Atividades de fecho da fase de testes:** estas atividades pressupõem a recolha de dados das atividades de teste já encerradas a fim de consolidar experiências, factos e números. As atividades de fecho da fase de testes ocorrem em marcos do projeto, tais como o lançamento do sistema de software, a conclusão de um projeto de testes, um marco alcançado ou uma versão de manutenção concluída.

A existência de um processo de teste bem definido é importante e decisiva nos resultados obtidos.

2.3.2. Classificação dos testes de software

Existem várias classificações associadas aos testes de software. Aqui adota-se a classificação proposta pelo ISTQB [3], a qual classifica os testes de software em níveis de teste e tipos de teste. A classificação em níveis de teste indica se o teste se destina à verificação do sistema ou a parte de um sistema. A classificação em tipos de teste refere-se ao objetivo específico do teste.

Níveis de teste:

- **Testes de componentes:** estes testes são também conhecidos como testes de unidade, módulos ou programas e têm como objetivo procurar por defeitos e verificar o funcionamento de módulos de software, programas, objetos ou classes que possam ser testados separadamente. Podem incluir testes a funcionalidades e características não-funcionais específicas, tais como o comportamento de recursos, testes de robustez ou testes estruturais. Os casos de teste são derivados das especificações dos componentes, da conceção do software ou do modelo de dados. Estes testes contam com o apoio do ambiente de desenvolvimento para obterem a estrutura dos testes unitários ou ferramentas de *debug* por isso, pode-se afirmar que estes tipos de testes envolvem o *developer* que, normalmente, corrige os defeitos logo assim que são detetados sem ser necessária uma grande gestão.
- **Testes de integração:** estes testes têm como objetivos testar as interfaces entre componentes, interações entre diferentes partes de um sistema, tais como, o sistema operativo, sistema de arquivos e *hardware*, e as interfaces entre os sistemas.
  Os testes de integração podem ser executados em diferentes fases:
  1. Após os testes de componentes, caso o objetivo seja testar as interações entre componentes de software.
  2. Após os testes de sistema, caso o objetivo seja testar a interação entre diferentes sistemas ou entre hardware e software.

  Neste nível de teste, os *testers* concentram-se apenas na integração propriamente dita. Por exemplo, se estão a integrar o módulo A com o módulo B, irão estar interessados em testar a comunicação entre os módulos e não a funcionalidade de cada um, como é efetuado durante os testes de componentes.
- **Testes de sistema:** o objetivo destes testes é testar o comportamento de todo o sistema ou produto. O ambiente de teste deve corresponder o máximo possível ao objetivo final ou ao ambiente de produção de modo a minimizar o risco de falhas do ambiente não detetadas em testes anteriores. Os testes de sistema podem basear-se nas especificações de requisitos, processos de negócio, casos de uso, etc.

Estes testes devem investigar os requisitos funcionais e não funcionais do sistema e também a qualidade dos dados. Testes de sistema começam por utilizar a técnica baseada nas especificações (caixa-preta). As técnicas baseadas na estrutura (caixa-branca) podem ser utilizadas para avaliar o rigor dos testes face a um elemento estrutural, como por exemplo a navegação de uma página *web*.

- **Testes de aceitação:** os testes de aceitação são muitas vezes da responsabilidade dos clientes ou utilizadores finais do sistema. O objetivo destes testes é estabelecer a confiança no sistema e, por isso, o seu principal foco não é encontrar defeitos. Estes testes podem avaliar a disponibilidade do sistema para entrega e utilização.

  Os testes de aceitação podem ser executados em várias fases:

  1. Quando o produto é instalado ou integrado.
  2. Durante os testes de componentes caso um teste de aceitação à usabilidade de um componente.
  3. Antes dos testes de sistema caso seja um teste de aceitação a uma nova funcionalidade ou melhoria.

Tipos de teste

- **Testes funcionais:** estes testes baseiam-se em funções e características que são descritas em documentos e na sua interoperabilidade com sistemas específicos. Estes testes são também conhecidos por testes de caixa-preta. Os testes funcionais podem ser executados em todos os níveis de teste, por exemplo, os testes para componentes podem ser baseados em especificações de componentes.
  O ISTQB [3] considera que dentro dos testes funcionais incluem-se também:
  - Testes de segurança: investigam as funções relacionadas com a deteção de ameaças, como vírus.
  - Testes de interoperabilidade: avaliam a capacidade do produto de software de interagir com um ou mais componentes com sistemas outros sistemas.
- **Testes não funcionais:** estes tipos de teste podem ser realizados em todos os níveis de teste. Têm como objetivo medir as características dos sistemas e softwares que podem ser quantificadas, como por exemplo tempos de resposta.

  Os testes não funcionais incluem outros testes, tais como:

  - Testes de desempenho: testes executados para determinar o desempenho de um produto ou software.

  - Testes de carga: tipo de teste de desempenho realizado para avaliar o comportamento de um componente ou sistema com carga crescente.
  - Testes de *stress*: tipo de teste de desempenho realizado para avaliar um sistema ou componente dentro dos limites das suas cargas de trabalho especificadas, como acesso à memória ou servidores.
  - Testes de usabilidade: testes realizados para determinar até que ponto o produto é entendido, fácil de aprender, fácil de utilizar e atraente para os utilizadores.
  - Testes de manutenção: testes utilizados para determinar a capacidade de manutenção de um produto.
  - Testes de fiabilidade: testes utilizados para determinar a confiabilidade de um produto.
  - Testes de portabilidade: testes utilizados para determinar a portabilidade de um produto.
- **Testes estruturais:** estes testes, também conhecidos como testes de caixa-branca, podem ser realizados em todos os níveis de teste. As técnicas estruturais são bem-sucedidas se aplicadas depois das técnicas baseadas nas especificações, isto permite medir o rigor dos testes através da avaliação da cobertura de um tipo de estrutura.
  A cobertura é medida através da execução de vários testes e é expressa em percentagem. Se a cobertura não atingir os 100% têm de ser feitos mais testes para testar os itens que não foram cobertos.
- **Testes de regressão:** após um defeito ser detetado e corrigido o software deve ser testado novamente para confirmar que o defeito foi removido com sucesso e para verificar se, em consequência das alterações efetuadas, não foram introduzidos novos defeitos nas partes não alteradas. Desta forma, podemos afirmar que os testes de regressão são a repetição de testes a um programa/software já testado que sofreu uma modificação devido a um defeito encontrado. Os testes devem ser repetíveis para que possam ser utilizados como testes de confirmação e como suporte aos testes de regressão.
  Os testes de regressão podem ser realizados em todos os níveis de teste, bem como testes funcionais, não funcionais e estruturais. São executados muitas vezes e por isso os testes de regressão devem ser automatizados.

2.4. Plataformas de desenvolvimento *low-code*

Nos últimos anos, tem-se assistido a uma tendência em que as empresas trocam as abordagens tradicionais de desenvolvimento de software por abordagens *agile* [8] e *DevOps* [9]. Esta tendência tem levado também muitas empresas e *developers* a adotar,

cada vez mais, plataformas de desenvolvimento *low-code*. Esta tendência é referida no relatório *The State of Application Development* [10] que num estudo apresentado refere que 41% das empresas entrevistadas disseram que sua organização já está a usar uma plataforma de desenvolvimento *low-code* e, outras 10% disseram que estavam a preparar-se para usar uma. Esse interesse crescente também é corroborado pelo *Low-Code Development Platform Market* [11]. Neste estudo é referido que o tamanho do mercado global de plataformas de desenvolvimento de *low-code* deve crescer de US$13,2 bilhões em 2020 para US$45,5 bilhões em 2025, a uma taxa de crescimento anual de 28,1% durante o período de previsão.

As plataformas de desenvolvimento *low-code* são ambientes de desenvolvimento integrado com base visual, que incluem muitas das ferramentas e funcionalidades que os *developers* e as equipas de desenvolvimento de software usam separadamente para projetar, codificar, testar e gerir as suas aplicações [12]. Estas plataformas são ferramentas de desenvolvimento que têm como objetivo facilitar e acelerar a criação e implementação de aplicações. O *low-code* permite que as aplicações sejam desenvolvidas com o mínimo de codificação e programação possível, o que faz com que o produto possa ser entregue num prazo comparativamente muito menor. Com essas plataformas, os *developers* ainda podem precisar fazer alguma codificação, para operações ou funcionalidades específicas, mas uma parte significativa do trabalho pode ser feita por meio de uma interface de arrastar e soltar [13].

Existem várias vantagens associadas ao uso de plataformas de desenvolvimento *low-code*, nomeadamente [14] o suporte para várias plataformas, facilidade integração de recursos já desenvolvidos, facilidade de adaptação mesmo por pessoas sem conhecimentos específicos de linguagens ou sem experiência em plataformas *low-code*, desenvolvimento de aplicações para múltiplas plataformas simultaneamente. Outros dos fatores de sucesso destas plataformas são identificados em [15] sendo alguns os seguintes: incremento de agilidade; aumento de produtividade; diminuição de custos; melhoria na experiência de utilizador; mudança mais fácil; transformação mais rápida; entre outros.

Atualmente existem várias plataformas *low-code* disponíveis no mercado. Algumas das mais conhecidas, que fazem parte de uma lista mais completa disponível em [16] são:

- OutSystems (www.outsystems.com);
- Creatio (www.creatio.com);
- Mendix (www.mendix.com);
- Appian (www.appian.com);
- PowerApps (www.powerapps.microsoft.com).

As plataformas *low-code* possuem características próprias que divergem entre elas em número e/ou funcionalidade. A *OutSystems* possui funcionalidades de *Drag & Drop,* tem uma curva de aprendizagem baixa, é flexível e é fácil de usar, permite criar protótipos rapidamente, simplifica o ciclo de *DevOps* e facilita a reutilização de

recursos. A *Mendix* permite gerir projetos de forma ágil, possui ferramentas de modelação visual e componentes reutilizáveis. A *Creatio* apresenta um *design* e desenvolvimento com *Drag & Drop* e possui também ferramentas de modelação visual. A *Appian* possui funcionalidades também de *Drag & Drop*, bem como serviços nativos de Inteligência Artificial, integração com outras plataformas através de *Google Cloud*, *Amazon AWS* e *Microsoft Azure*. A *PowerApps* já disponibiliza vários elementos e modelos de páginas pré-construídos de modo a facilitar o desenvolvimento das aplicações. Existem vários outros estudos onde se comparam outras características, como por exemplo os que estão disponíveis em [17][18][19].

A Tabela 2 apresenta uma comparação das plataformas *low-code* mencionadas anteriormente no âmbito da modelação visual e UI, casos de uso, integrações, desenvolvimento, preços e período experimental.

Tabela 2 - Comparação de plataformas *low-code* em termos de modelação visual, casos de uso, integrações, desenvolvimento, preços e período experimental (Fonte: [16]).

	outsystems	Creatio	mendix	Appian	PowerApps
Modelação Visual e UI	Ambiente de desenvolvimento visual, extensível com código customizado. *Design* de arrastar e soltar.	Modelação visual. *Design* e desenvolvimento de arrastar e soltar.	Ferramentas de modelação visual. Interfaces de arrastar e soltar. Componentes reutilizáveis.	Modelação visual. Interfaces de arrastar e soltar.	*Design* de aplicação orientado ao modelo. Modelos e elementos de UI pré-construídos
Casos de Uso	Permite um modelo *offline* complexo Independente de outras tecnologias	Automatização, mecanismo robusto para gerir processos de negócios. Reutilização de módulos. Ferramentas sem código	Metodologia ágil de projetos. UI responsivo.	Reutilização de elementos. Suporte para escalabilidade.	Automatização do fluxo de trabalho. Suporte para colaboração.
Integrações	Com qualquer sistema ou base de dados	Com dados, sistemas e *web services*	APIs públicas SDK	Com dados, sistemas e *web services*	Integração com serviços baseados na *cloud*
Desenvolvimento	*On-premise*, SaaS	*On-premise*, SaaS	*On-premise*, SaaS	*On-premise*, SaaS	*On-premise*, SaaS

Preços	4,000$ (pacote básico) 10,000$ (pacote *standard*)	25$/mês por utilizador	1,917$/mês (pacote básico) 7,825$/mês (pacote empresarial)	90$/mês por utilizador	10$/mês por utilizador para uma aplicação 40$/mês por utilizador para n aplicações
Periodo experimental	Disponível Tempo ilimitado	Disponível Tempo ilimitado	Disponível Limitado a 50 utilizadores	Disponível durante 14 dias	Disponível durante 30 dias

Uma caracterização de várias plataformas *low-code* é apresentada na Figura 2. A caracterização foi elaborada pela G2 [20], que é uma plataforma que ajuda empresas a encontrar o melhor software para o seu tipo de negócio tendo em conta opiniões e avaliações de utilizadores desses mesmos softwares, e considera a satisfação dos utilizadores e a presença das plataformas no mercado.

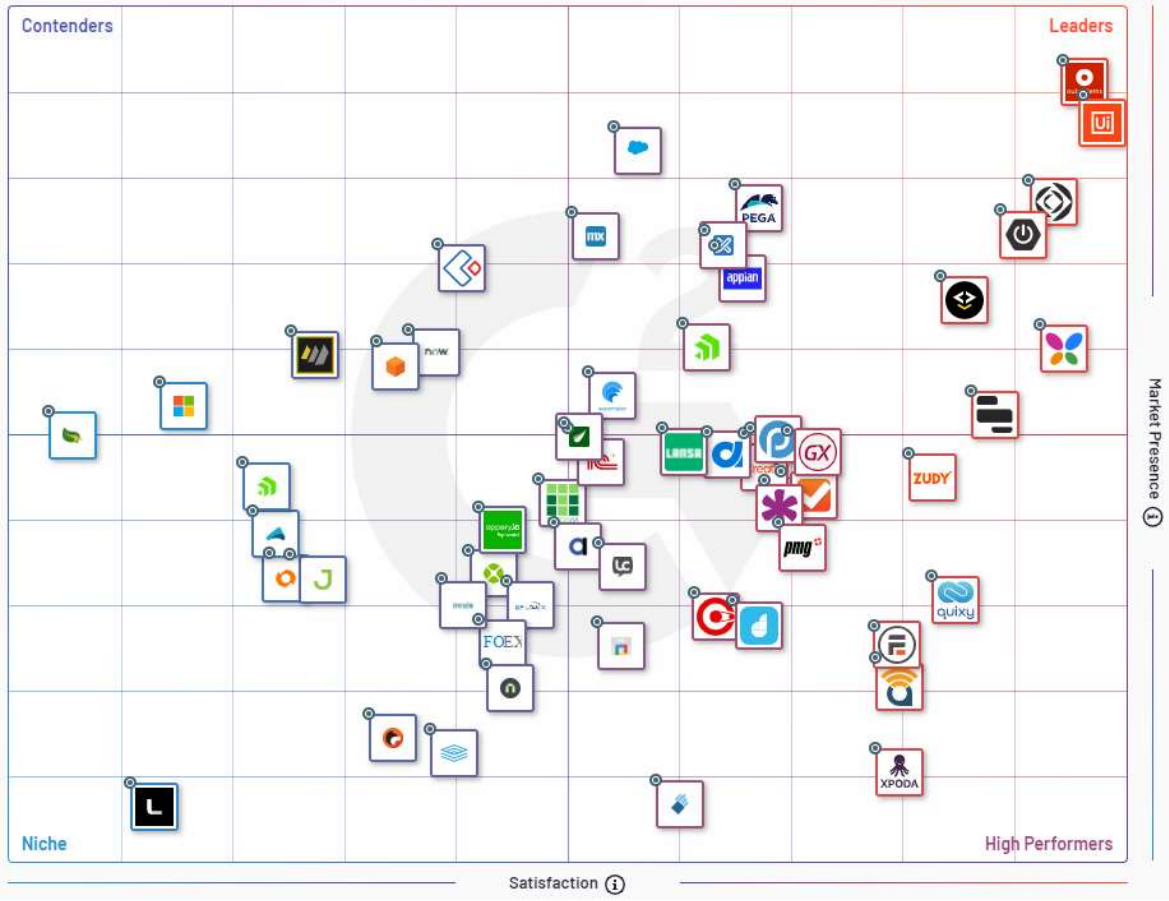

Figura 2 – Plataformas *low-code* considerando a satisfação dos utilizadores e presença no mercado (Fonte: [20]).

Existem outros estudos que classificam as diversas plataformas de diferentes formas. A plataforma *OutSystems*, usada neste trabalho, é uma das mais bem classificadas em vários desses estudos, como por exemplo naquele que é designado como um quadrante mágico da Gartner [21] (Figura 3), que considera os aspetos de habilidade para executar *vs* abrangência da visão, e no qual esta plataforma é uma das líder segundo essa classificação.

Figura 3 – Quadrante mágico para plataformas *low-code* (Fonte: [21]).

Independentemente da caracterização considerada, e da entidade que a elabora, existe um conjunto de plataformas *low-code* que parecem destacar-se, e são frequentemente identificadas como líderes, nomeadamente: *OutSystems*, *Mendix*, *Appian* e *Salesforce*.

2.5. A plataforma Outsystems

A plataforma *low-code OutSystems*, sendo uma das líderes, oferece uma vasta gama de funcionalidades a um vasto universo de *developers*. Os *developers* que trabalham com *OutSystems* conseguem experienciar um desenvolvimento de aplicações, na *cloud* ou *on-premisse*, e ter acesso a painéis de desenvolvimento em tempo real, o que lhes permite obter *feedback* das aplicações desenvolvidas. A *OutSystems* permite que os seus *developers* entreguem aplicações escaláveis e com segurança as quais podem ser integradas em várias plataformas.

A forma como são desenvolvidas as aplicações/softwares pode mudar com a *OutSystems* pois, os *developers* podem implementar e contruir aplicações de forma rápida e correta utilizando ferramentas de alta produtividade. Aplicações desenvolvidas em *OutSystems* podem ser entregues em semanas ou até mesmo dias porque cada aspeto desta plataforma foi projetado para ajudar os *developers* nas fases mais críticas e lentas da gestão do ciclo de vida de uma aplicação. A infraestrutura e o facto de ter várias ferramentas integradas garante que as aplicações mais robustas sejam também seguras e resilientes.

Desde o início que a plataforma *OutSystems* foi projetada para as mudanças serem geridas de forma fácil uma vez que há muitas aplicações que são desenvolvidas que devem estar prontas para evoluírem conforme os negócios e as tecnologias.

Existem algumas funcionalidades e características da plataforma *OutSystems* que são valorizadas pelos clientes, tais como[22][23]:

- **Risco mínimo com retorno do investimento mais elevado:** a *OutSystems* e as plataformas *low-code* no geral são plataformas personalizáveis e que fornecem vários recursos integrados de *cross-platform*, segurança e integração de dados. Desta forma, os *developers* e empresas podem prestar mais atenção a outras atividades importantes do negócio em vez de dedicar tanto tempo a identificar ou resolver problemas técnicos.
- **Contruir uma vez e implementar em todos os sítios:** as aplicações contruídas em *OutSystems* podem ser facilmente instalados em ambientes móveis (*smartphones*) ou de *desktop*. É possível também criar aplicações para vários dispositivos.
- **Funcionalidades inovadoras:** neste tipo de plataformas não é preciso codificar manualmente uma aplicação o que permite que os *developers* possam personalizar aplicações facilmente e criar funcionalidades inovadoras para atender às necessidades do negócio.
- **Manutenção do ciclo de vida completo da aplicação:** Mesmo após um lançamento bem-sucedido de uma aplicação, o *developer* precisa de atualizar as suas aplicações com novos recursos e correções de erros de forma periódica. Com *OutSystems* é fácil fazer atualizações complexas num curto espaço de tempo.
- **Mais satisfação e interesse da parte dos clientes:** o facto das aplicações serem criadas e entregues de forma rápida, funcional e em poucos dias faz com que haja mais satisfação e interesse por parte dos clientes. Isto também se deve ao facto de existirem ciclos de interação dinâmicos e curtos.
- **Gestão de múltiplas aplicações:** a plataforma *OutSystems* utiliza a mesma infraestrutura para gerir várias aplicações simultaneamente, por exemplo, um cliente que pretenda uma aplicação *web* e outra *mobile* consegue utilizar a mesma infraestrutura para o desenvolvimento de ambas.

- **Eliminação dos requisitos das habilidades de técnicos informáticos:** para desenvolver em *OutSystems* não é necessária tanta experiência ou conhecimento de linguagens específicas para criar aplicações. Isto permite que pessoas que tenham interesse neste tipo de desenvolvimento possam criar os seus próprios protótipos e não terem de esperar por técnicos informáticos.
- **Desenvolvimento visual *full-stack*:** o facto de os utilizadores poderem "escrever" código de uma forma muito fácil e simples como arrastar e soltar ecrãs, processos de negócio, lógica e modelos de dados permite a criação de aplicações *full-stack* e *cross-platform*.
- ***Feedback* direto na aplicação:** os utilizadores podem partilhar *feedback* por voz ou por escrito na aplicação de modo a simplificar o processo de gestão de mudança. Esta funcionalidade torna as aplicações melhores e mais rápidas.
- **Mobile fácil:** aplicações para *smartphones* podem ser criadas facilmente com um *layout* e *design* excelente, sincronização de dados *offline* e acesso a dispositivos nativos.
- **Implementação com um clique:** é possível implementar e atualizar aplicações apenas com um clique. A *OutSystems* tem também uma funcionalidade, chamada *TrueChange* que verifica automaticamente as dependências e trata de todos os processos de implementação.
- **Reestruturação automática:** a *OutSystems* analisa todos os modelos e imediatamente reestrutura as dependências. Por exemplo, ao modificar uma tabela do modelo de dados todas as *queries* em que é utilizada essa tabela serão atualizadas automaticamente.
- **Arquitetura em escala:** é possível criar ou alterar serviços e aplicações reutilizáveis de forma rápida e em escala.

De facto, a plataforma *low-code OutSystems* é uma das líderes deste mercado e é das que oferece mais funcionalidades. No entanto, e apesar de o desenvolvimento *low-code* ser frequentemente associado ao desenvolvimento sem erros, tal não é verdade e é necessário estudar as diversas estratégias e ferramentas que melhor se adequem com estas plataformas. Foi esta realidade que fez com que neste trabalho se use esta plataforma e para a qual se pretende contribuir na área de testes de software, que como se sabe, é uma das etapas que muito contribui para o desenvolvimento de software de qualidade. No entanto, os testes de software são também uma das vertentes muito pouco explorada e referida nos estudos comparativos.

2.6. Automatização de Testes em Metodologias Ágeis

As metodologias ágeis são comumente usadas em conjunto com plataformas de desenvolvimento de *low-code*. Nessas metodologias, existem três metodologias de teste que são frequentemente usadas para automatização de testes: *Test-Driven Development* (TDD) [24], *Acceptance Test-Driven Development* (ATDD) [25] e *Behavior Driven Development* (BDD) [26].

TDD [24] é uma metodologia usada para desenvolver código guiado por casos de teste automatizados. Esta metodologia é também conhecida como teste primeiro, pois os casos de teste são escritos antes do código. Os casos de teste escritos são principalmente testes de unidade embora também sejam usados nos níveis de integração ou sistema. Uma limitação do TDD é que se o *developer* entender mal o que o software deve fazer, os testes de unidade também incluirão os mesmos mal-entendidos, dando resultados satisfatórios, mesmo que o software não esteja de acordo com os requisitos especificados [27].

A ATDD [25] é também baseada no conceito de teste primeiro. Nesta metodologia os critérios de aceitação e os casos de teste são escritos no início do processo de desenvolvimento, na fase de confirmação do processo de desenvolvimento de uma *user story*. O ATDD é uma abordagem colaborativa que permite que cada parte interessada entenda como o componente de software se deve comportar e o que os diversos *stakeholders*, *tester*, *developer* e outros, precisam fazer para garantir o seu comportamento. Nesta metodologia são os testes de aceitação que verificam a *user story* implementada [27].

Na BDD [26] assume-se que os comportamentos do software são mais fáceis de serem entendidos pelas partes interessadas quando estas participam na criação. Os testes são então baseados nos comportamentos esperados, e o *developer* executa esses testes enquanto desenvolve o código. Esta metodologia promove uma automatização de teste imediata ainda mais do que na metodologia ATDD [27]. Na BDD, os testes de aceitação são frequentemente formulados usando uma linguagem estruturada, conhecida como linguagem *Gherkin*. Neste caso, os requisitos são definidos usando o formato '*Given – When –Then'* e são a base para os casos de teste de aceitação servindo também como base para a automatização de teste [28].

2.7. Testes Automatizados vs Testes Manuais

Testar o software manualmente é muitas vezes difícil e pouco prático. Os testes manuais podem exigir muitos recursos, são geralmente lentos e são vulneráveis a resultados imprecisos. Existem mesmo muitas situações em que as abordagens de testes manuais podem não ser as mais eficazes, principalmente nos casos em que pode ser útil repetir os testes já efetuados como os testes de regressão. Em muitas situações os testes automatizados representam uma alternativa que deve ser considerada. No

entanto, para que sejam efetivamente válidas, estas abordagens podem implicar algumas alterações significativas no processo de desenvolvimento que, em alguns casos, podem limitar alguns dos benefícios normalmente associados às plataformas de desenvolvimento *low-code*. A automatização, não sendo bem adotada, pode implicar um aumento significativo no tempo de desenvolvimento ou aumento da necessidade de codificação. É também importante considerar que no BDD [26] nem todos os cenários de teste necessitam de ser automatizados pois alguns podem ser difíceis de automatizar ou podem ser mais adequados para abordagens manuais. Assim, é importante avaliar as situações e as implicações em que a automatização dos testes seja benéfica e utilizar as ferramentas de automatização de teste mais adequadas.

2.8. Testes de software na plataforma *OutSystems*

As aplicações criadas em *OutSystems* beneficiam da continua validação de integridade que verifica o impacto de todas as alterações nas camadas das aplicações (modelo de dados, lógica de negócios ou apresentação) para assegurar que tudo está integrado no período da implementação.

Os recursos de autocorreção permitem corrigir automaticamente os problemas ou informar os *developers* sobre as correções que estes devem executar. Quando uma alteração é feita, a *OutSystems* verifica todas as dependências existentes, automatiza os *scripts* de atualização da base de dados e analisa o impacto que essa alteração terá nas aplicações em execução.

A um nível geral, a *OutSystems* faz uma análise de impacto para várias aplicações ao criar planos de implementação no *LifeTime*[1], avaliando o impacto de mover novas versões das aplicações selecionadas para o ambiente de destino antes de ser executada a implementação. Como resultado deste processo, o número de *bugs* introduzidos é, geralmente, menor quando comparado às tecnologias de desenvolvimento tradicionais, o que se traduz em menos ciclos de teste e correção de problemas, reduzindo o esforço associado ao desenvolvimento e entrega. No entanto, não é excluída a necessidade de testes durante o ciclo de vida da aplicação.

[1] LifeTime é uma aplicação que permite gerir os vários ambientes, aplicações, utilizadores do IT e segurança, dando assim cobertura a todo o ciclo de vida da aplicação desde o desenvolvimento até à implementação.

Assim, ao longo do ciclo de vida de uma aplicação *OutSystems* existem vários momentos em que as atividades de teste devem ser realizadas como é possível ver na Figura 4.

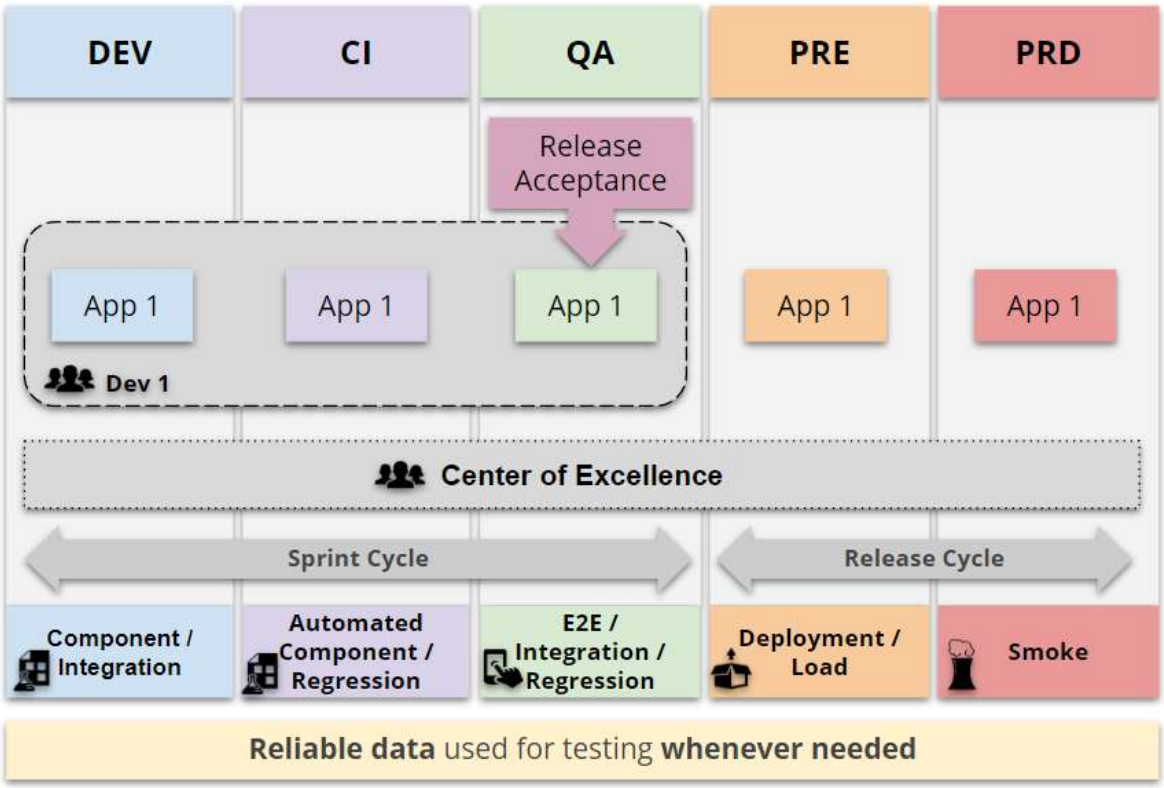

Figura 4 - Atividade típicas de testes de uma aplicação OutSystems [Fonte: [39]].

Os diversos testes são realizados em momentos chave do desenvolvimento do software e cada teste tem objetivos específicos.

Testes de Componentes: testa o comportamento de unidades de código (componentes). Em *OutSystems*, as unidades de código correspondem aos elementos chamados *Actions* onde é implementada a lógica de negócio. Estas ações são entregues pelos *developers*, no ambiente de desenvolvimento (DEV), como parte das atividades desenvolvidas no *sprint*. Como é possível ver na Figura 4 os testes de componentes devem ser executados no ambiente de DEV bem como no ambiente de integração contínua (CI), de forma automatizada, como parte do fluxo de integração. Os testes de componentes podem seguir duas abordagens diferentes:

- Solidariedade: incorporar algumas integrações com componentes *OutSystems* externos.
- Solitária: isolar a comunicação com outros componentes *OutSystems.*

Testes de Integração/API: testes que validam a integração com sistemas externos. Estes testes devem ser executados no ambiente de DEV pelos *developers*, manualmente ou de forma automatizada, ou no ambiente de qualidade (QA) no caso de existirem

integrações que não estejam disponíveis de forma confiável no ambiente de desenvolvimento.

Testes de Sistema ou *End-to-End* (E2E): testes que incluem funcionalidades completas da perspetiva do utilizador final ou do sistema. Geralmente, estes testes são executados através de uma interface *web* ou móvel (testes de interface). Nem todos os testes E2E são testes de UI (*user-interface*). Por exemplo, testar uma API exposta por uma aplicação *OutSystems* pode ser considerado um teste E2E. Este tipo de testes pode ser automatizado pelos *developers* no caso de não serem testes de UI ou podem ser automatizados pela equipa de qualidade no caso de testes de UI, por isso estes testes devem ser executados no ambiente de QA.

Testes de Regressão: testes que têm como objetivo garantir se as funcionalidades executadas de forma correta anteriormente continuam a ser bem executadas em versões mais recentes das aplicações. À medida que são adicionadas novas funcionalidades, a necessidade de ter testes de regressão automatizados também aumenta uma vez que esta é a única maneira de acelerar a validação de novas mudanças nas aplicações. A entrega contínua das funcionalidades depende muito desta capacidade pois os *developers* só podem entregar novas funcionalidades, ou mudanças em funcionalidades, quando estas estiverem prontas para produção. Estes testes devem ser executados no ambiente de CI e também no ambiente de QA.

Testes de Performance: é um teste não funcional que visa testar o comportamento da aplicação ao nível do desempenho. Os testes de carga são o caso mais comum e verificam o comportamento da aplicação em relação ao número de utilizadores. Contudo, para aplicações com uma quantidade de dados muito elevada pode fazer mais sentido testar o seu desempenho com muitos dados a serem manipulados. Deve haver um ambiente específico para executar este tipo de testes.

Testes de Aceitação/Funcionais: estes testes são feitos pela empresa/cliente antes da aprovação de um novo conjunto de alterações ou antes da aplicação entrar para produção. A maioria dos casos abordados são recursos críticos para o negócio, e normalmente, são realizados de forma manual no ambiente de QA mas também podem ser realizados no ambiente de DEV pelos *developers* antes das funcionalidades serem passadas para o ambiente seguinte.

Testes de Segurança: testa a segurança de um determinado sistema explorando vulnerabilidades de segurança comuns em tempo de execução. Estes testes devem ser realizados o mais cedo possível no ciclo de vida de entrega da aplicação e são executados, normalmente, por equipas de segurança internas ou via serviços externos. Estes testes podem ser executados em qualquer ambiente, contudo, se forem executados em DEV a equipa de desenvolvimento pode corrigir rapidamente os problemas e podem também aprender e perceber quais as vulnerabilidades que podem evitar durante o desenvolvimento da aplicação.

Testar uma aplicação requer sempre um investimento por parte da equipa de entrega. Como tal, o tipo de atividades de teste e a quantidade de esforço a serem

investidos irão depender em grande parte da complexidade ou do impacto comercial da aplicação que está a ser entregue. Como regra geral, quanto maior o risco associado a uma funcionalidade mais testes devem ser feitos a essa funcionalidade.

Os testes podem ser executados manualmente utilizando uma abordagem exploratória, seguindo um *script* de teste ou podem ser automatizados com uma infinidade de ferramentas, algumas das quais serão descritas no capítulo seguinte.

Por fim, uma das boas práticas recomendadas é utilizar dados confiáveis sempre que possível de modo a dar suporte a diferentes atividades de teste, permitir cenários de teste realistas e identificar problemas relacionados com os dados no início do ciclo de entrega antes da aplicação ir para produção. Para conseguir isto, as equipas de entrega devem entender as opções que têm para realizar todos os testes o mais rápido possível.

Considerando o exposto nas secções anteriores, este trabalho foca-se na utilização de ferramentas de teste para automatização de Testes Funcionais, ao nível dos Testes de Unidade/Componentes e Testes de Sistema, em aplicações *OutSystems*, e na importância e implicações que a utilização de boas práticas durante o desenvolvimento das aplicações *OutSystems* tem na automatização das diversas atividades de teste.

3. Ferramentas de Testes

Neste capítulo são analisadas ferramentas de teste de software que podem ser usadas para automatizar testes unitários/componentes e de sistema de aplicações *OutSystems*. Embora existam diversas ferramentas, optou-se por considerar neste estudo as seguintes 3 ferramentas: a *BDDFramework* [2], o *Ghost Inspector* [29] e o *Tricentis Tosca* [30]. A *BDDFramework* é uma ferramenta desenvolvida em *OutSystems,* sendo atualmente a única, com estas características, que permite a execução de testes unitários/componentes em aplicações *OutSystems*. A *Ghost Inspector* e a *Tricentis Tosca* são ferramentas pagas, mas facultam um período gratuito. A *Ghost Inspector* é uma ferramenta de automatização de testes *web*, especializada em testes de interface do utilizador. A *Tricentis Tosca* permite não só automatizar testes de interface do utilizador como também permite testes de API o que é bastante útil para testes de serviços e lógica de negócios, esta ferramenta é apropriada para as metodologias *Agile* e *DevOps*.

A *BDDFramework* foi uma das ferramentas escolhidas porque atualmente é a única ferramenta que permite a execução de testes unitários/componentes em aplicações *OutSystems* uma vez que foi desenvolvida mesmo com esse propósito.

O *GhostInpector* foi escolhido por ser uma ferramenta que permite criar, gerir e executar testes de UI baseados em *Selenium* e por oferecer uma interface simples e de fácil aprendizagem.

A *Tricentis Tosca* é uma ferramenta bastante completa uma vez que disponibiliza a possibilidade de desenvolver vários tipos de testes, neste caso, testes de interface do utilizador e também testes de serviços e de lógica de negócios. É por ter todo este potencial que esta ferramenta foi escolhida para análise.

3.1. BDDFramework

A *BDDFramework* [2] é uma ferramenta de automatização de testes desenvolvida em *OutSystems* na qual os testes são especificados utilizando a sintaxe *Gherkin* [31]. *Gherkin* é uma linguagem que serve para estruturar e descrever os comportamentos esperados de uma aplicação. O principal objetivo das ferramentas BDD é oferecer suporte ao *Behavior-Driven Development* (BDD) [32], onde todos os participantes técnicos (por exemplo, *developers*) e não técnicos (por exemplo, analistas de negócio), de um projeto de software, colaboram para definir o seu comportamento. As organizações que não seguem os processos do BDD também podem tirar proveito das ferramentas BDD. Durante todas as fases do desenvolvimento de software os diversos intervenientes no projeto analisam os vários resultados de testes executados diariamente. Um desenvolvimento eficiente e a prática da qualidade exige que todos os envolvidos no projeto entendam os resultados dos testes. Muitos profissionais,

associados ao desenvolvimento e à qualidade do software, consideram a sintaxe *Gherkin* muito apelativa [31], uma vez que ela fornece um padrão que todos devem seguir para configurar determinados níveis de automatização de testes.

Em *OutSystems*, o componente da *forge BDDFramework*[2] pode ser utilizado em aplicações *web*, na componente servidor de aplicações *mobile* e também em *web services*, como REST, SOAP e outras APIs. Esta ferramenta pode ser utilizada com diferentes objetivos dependendo do que é necessário para o contexto do utilizador.

Antes de proceder à execução de um teste, o cenário desse teste deve ser escrito na sintaxe de *Gherkin*, ou seja, o cenário é divido nas seguintes etapas: Dado, Quando e Então. Por exemplo, considerando que estamos perante uma aplicação de compras *online,* o utilizador adiciona um produto ao seu carrinho e verifica que o carrinho foi atualizado corretamente com o produto adicionado e o respetivo preço. Este cenário na sintaxe de *Gherkin* seria representado na seguinte forma:

Cenário: Adicionar um produto ao carrinho

Dado:

Que eu tenho um carrinho

E existe um produto chamado com o nome *xpto*

Quando:

Eu adiciono o produto ao carrinho

Então:

A operação deve ser bem-sucedida

E o carrinho deve ser atualizado corretamente

Cenário: descreve o cenário específico que ilustra uma regra de negócio (neste exemplo, adicionar um produto ao carrinho).

Dado: descreve o contexto inicial do cenário - as pré-condições necessárias antes de realizar a ação/evento que está a ser testado (neste caso, as pré-condições são ter um carrinho de compras virtual e um produto específico a ser adicionado).

Quando: descreve a ação/evento específico - em muitos cenários, deve haver apenas uma dessas etapas (por exemplo, adicionar o produto ao carrinho). Se for necessário adicionar mais do que uma etapa aqui, deve dividir-se o cenário em duas ou mais etapas.

Então: descreve os resultados esperados da realização da ação/evento no sistema. Essas etapas geralmente contêm várias asserções que verificam tudo o que queremos verificar como resultado deste teste.

Nesta sintaxe, o cenário deve ficar claro para quem o lê, sejam eles técnicos ou não técnicos. Essa clareza é uma característica altamente valorizada que o *BDDFramework*

mantém durante todo o processo de *design*, implementação e execução de cenários de teste.

Após a execução do teste, o resultado é apresentado numa página que é composta por todas as etapas do cenário e os respetivos passos de cada etapa. São sinalizados a verde os passos executados com sucesso, a vermelho os passos que falharam e a cinzento os passos que não foram executados, visto que quando falha um dos passos do teste os seguintes não são executados. Por fim, é apresentado se o cenário foi executado, ou não, com sucesso.

Funcionalidades

Nesta *framework*, os cenários de teste são definidos em ecrãs através de *web blocks* que correspondem a uma parte do ecrã e que podem ter a sua própria lógica. A lógica correspondente aos passos de cada teste (dado, quando, então) será dividida por ações nos respetivos ecrãs de teste.

A estrutura de teste da BDD inclui quatro *web blocks* que os *developers* devem utilizar para criar os testes:

- **BDDScenario:** cada cenário é representado por um *BDDScenario*;
- **BDDStep:** cada grupo de etapas (dado, quando e então) é representado por um *BDDStep*.
- **FinalResult:** retorna estatísticas sobre todos os cenários executados no ecrã (contagem de testes bem-sucedidos, contagem de testes com falha e assim por diante). Deve ser sempre incluída no final.
- **SetupOrTeardownStep:** é um tipo especial de etapa que pode ser incluída em cenários, antes ou depois da lógica do teste, para realizar operações de configuração/limpeza de dados que estão fora da estrutura do cenário de uma perspetiva funcional/comercial como por exemplo, excluir os dados de teste que foram criados durante o teste de modo a não criar dados desnecessários na aplicação.

É necessário esclarecer que para executar e montar os testes com a *BDDFramework* é preciso ter conhecimento e saber desenvolver com *OutSystems*.

3.2. Ghost Inspector

O *Ghost Inpector* [29] é uma ferramenta de automatização de testes de software e manutenção de *sites*, que visa verificar problemas num *site* ou numa aplicação. Esta ferramenta realiza operações num *browser*, da mesma forma que um utilizador. Esta ferramenta permite criar, manter e automatizar testes de interface do utilizador para verificar funcionalidades específicas. Esses testes automatizados são executados de

forma contínua a partir do serviço *cloud* e o utilizador que criou o teste é notificado caso este seja executado com erros.

Apesar de ser um serviço pago, esta ferramenta disponibiliza um período de experimentação grátis, onde os utilizadores podem usufruir de todas as funcionalidades do serviço. Existem três planos de compra do serviço e as principais diferenças entre eles são, para além do preço, o número de testes que podem ser executados por mês e o número de elementos que podem aceder à suite ou coleção de testes, visto que os mesmos testes podem ser executados por diferentes elementos de uma equipa.

Funcionalidades

O *Ghost Inspector* possui várias funcionalidades tornando-o completo e fácil de utilizar. Algumas delas são: gravar testes no *browser*, criar e editar testes com o editor de testes, monitorizar as várias funcionalidades dos *sites* ou aplicações, capturar e rever regressões nos testes, customizar as definições dos testes, receber notificações, sendo também possível integrar o *Ghost Inspector* com outras aplicações ou serviços [29].

O *Ghost Inspector* dispõe uma extensão para os *browsers* Chrome e Firefox, que permite gravar operações e funcionalidades do site diretamente no *browser*. A extensão funciona registando as ações que o utilizador executa no *browser*, como clicar em botões ou preencher formulários. Estas ações são depois transformadas em seletores de *JavaScript* para que possam ser executadas, futuramente, na mesma sequência. Para além do registo das ações, a extensão também pode gravar "Asserções". Asserções são confirmações de algo verdadeiro, por exemplo, um elemento existir na página. Estas são úteis para verificar se o teste está a funcionar como é suposto. Durante o processo de gravação os utilizadores podem mudar de ações de gravação para ações de asserção. Neste caso, a extensão irá destacar os elementos, à medida que o rato vai passando sobre eles, e irá utilizar lógica interna para decidir se o utilizador pretende afirmar que o elemento existe ou se deve verificar se o elemento contém o texto suposto.

Caso os utilizadores prefiram criar os seus testes manualmente, podem fazê-lo com o editor visual que o *Ghost Inspector* oferece. Em vez de codificarem os testes estes podem ser construídos de forma rápida e fácil utilizando a interface intuitiva que a ferramenta disponibiliza. Os testes podem ser executados imediatamente e editados posteriormente.

Esta ferramenta suporta um grande número de operações em cada etapa dos testes, desde cliques simples, *uploads* de arquivos ou *JavaScript* personalizado. As etapas podem conter variáveis que podem ser modularizadas e reutilizadas, o que faz com que o teste seja mais fácil de gerir no que toca a etapas comuns em vários testes. Suporta

CSS e XPath e permite importar testes no formato *Selenium* [33]. Algumas das operações, organizadas por categoria, que podem ser executadas nos testes são:

Operações: esta categoria abrange as operações padrão que um utilizador executará numa página *web*.

- Clique: executa um clique do rato num elemento específico. O padrão é um clique com o botão esquerdo do rato, mas estão disponíveis opções para clicar com o botão direito ou até mesmo clicar duas vezes.
- Atribuição: atribui um valor a um elemento.
- *Keypress*: executa uma ação de pressão numa tecla, como por exemplo dar *Enter* num elemento.
- Passar o rato por cima: move o cursor do rato sobre um elemento ativando o evento *JavaScript mouseover*.
- *Drag and Drop*: arrasta um determinado elemento e solta-o em cima de outro. Esta operação ainda se encontra numa versão beta.
- Pausa: pausar por um período específico, definido pelo utilizador. O tempo é especificado em milissegundos. Normalmente, esta operação é utilizada para que uma condição seja garantidamente executada.
- Executar *JavaScript*: executa código *JavaScript* feito pelo utilizador no *browser*.
- Capturar ecrã: pode ser utilizada para capturar a janela atual ou um determinado elemento.
- Sair do teste: terminar o teste mais cedo, obtendo um estado de aprovação ou reprovação no momento atual.

Navegação: esta categoria inclui duas operações simples para navegar para uma página ou atualizar.

- Ir para URL: navega para o URL especificado pelo utilizador.
- Atualizar: atualiza a página atual no *browser*.

Asserções: permitem confirmar que uma determinada condição é atendida na página. Apesar de não serem obrigatórias, as asserções são bastante recomendadas. Usar pelo menos uma asserção no final do teste é geralmente uma boa ideia, pois ajuda a confirmar se o teste foi executado conforme esperado.

- O elemento está presente: este passo é executado com sucesso caso o elemento especificado esteja presente no DOM[2]. Caso contrário o passo falha.
- O elemento não está presente: este passo é executado com sucesso caso o elemento especificado não esteja presente no DOM. Caso contrário o passo falha.

[2] *Document Object Model* (DOM) é uma interface de programação para documentos HTML e XML. Representa uma página onde possa ser alterado o conteúdo, a estrutura e os estilos do documento[38].

- O elemento está visível: este passo é executado com sucesso caso o elemento especificado esteja visível na página, isto é, se o elemento não possuir o atributo *display* definido como *none* ou atributo *visibility* definido como *hidden*.
- O elemento não está visível: este passo é executado com sucesso caso o elemento especificado não esteja visível na página, como foi descrito anteriormente.
- O valor do elemento não é igual: caso o valor/texto do elemento especificado seja igual ao valor introduzido pelo utilizador a etapa é executada com sucesso. Caso contrário a etapa falhará.
- O valor do elemento contém: caso o valor/texto do elemento especificado contenha o valor introduzido pelo utilizador a etapa é executada com sucesso. Caso contrário a etapa falhará.
- O valor do elemento não contém: caso o valor/texto do elemento especificado não contenha o valor introduzido pelo utilizador a etapa é executada com sucesso. Caso contrário a etapa falhará.
- *JavaScript* retorna verdadeiro: esta asserção permite executar *JavaScript* feito pelo utilizador e retornar um valor usando uma instrução de retorno. Caso esse valor de retorno seja verdadeiro este passo irá passar, caso contrário falhará.
- Guardar variáveis: as ações desta categoria permitem a criação de variáveis dentro dos testes.
- Definir variável: cria uma variável com um valor específico.
- Extrair de um elemento: cria uma variável e define-a com valor extraído do elemento especificado.
- Extrair de *JavaScript*: cria uma variável e define-a com o valor retornado pelo código *JavaScript* feito pelo utilizador.

Importar: permite importar passos de outros testes para o teste atual. Isto permite modularizar e reutilizar passos criados anteriormente. Por exemplo, se o utilizador quiser executar testes num *site* cuja autenticação é obrigatória, o *login* terá de ser sempre executado no início de todos os testes logo, será aconselhado fazer um teste para executar o *login* e chamar esse teste dentro de todos os outros.

Com o *Ghost Inspector* os utilizadores podem verificar continuamente se existem problemas nas funcionalidades do site criando testes e executando-os de acordo com o agendamento feito pelo próprio utilizador (por exemplo, a cada hora) ou quando implementa alguma alteração através da interface visual. A execução de um teste é o equivalente à navegação típica de um utilizador através da página *web*. O utilizador conseguirá ver exatamente quais as etapas que foram aprovadas e quais falharam. O *Ghost Inspector* fornece um vídeo completo da execução do teste, os valores de saída presentes na consola do *browser*, capturas de ecrã e até mesmo uma comparação entre a última execução e a atual.

Definições de testes

São várias as definições que se podem personalizar para cada teste presente na conta do utilizador. Em seguida, são enumeradas as várias secções bem como o que pode ser customizado em cada uma delas.

- **Detalhes dos testes:** o utilizador pode alterar informações básicas sobre o teste, tal como o nome ou descrição. Pode também alterar a *suite* (conjunto de casos de teste) à qual o teste pertence.
- **Agendamento dos testes:** nesta secção é definido o agendamento da suite que pode conter vários testes ou pode até ser feito o agendamento de cada teste. O padrão definido é que os testes devem utilizar o agendamento definido para a suite e caso seja definido um horário específico para os testes, então é esse que deve ser utilizado.
- **Definições do *browser*:** as configurações de acesso do *browser* permitem definir as condições iniciais do *browser* para o teste como por exemplo o URL inicial, a versão do *browser* e as credenciais de autenticação HTTP.
- **Tempo entre etapas:** os utilizadores podem ajustar os atrasos de tempo entre as várias etapas de um teste. Podem ser definidos tempos de pausa entre cada etapa, o que permite que os eventos de *JavaScript* sejam concluídos quando as ações são executadas. Para garantir que os elementos necessários para o teste estão carregados, pode ser definido um tempo de espera para cada elemento. Para finalizar o teste é retirada uma captura ao ecrã como já foi referido anteriormente, contudo para garantir que a comparação entre a imagem atual e a do último teste é realizada corretamente, pode também ser definido um período de espera.
- **Opções de exibição:** nesta secção podem ser feitas alterações no tamanho do ecrã utilizado no teste. Isto irá afetar o vídeo e a captura final do ecrã do teste. O utilizador pode também especificar um elemento, através de um seletor, desta forma apenas o elemento será capturado em vez de todo o ecrã. Podem também ser excluídos elementos para que não apareçam na imagem.
- **Geolocalização:** o *Ghost Inspector* oferece a capacidade de executar os testes em regiões específicas de todo o mundo e irá utilizar um endereço IP dessas mesmas regiões.
- ***Links* externos:** esta ferramenta permite associar *links* externos aos testes. Por exemplo, associar um teste que falhou a um *link* de um *ticket* no *Jira*[3]. Qualquer URL permanente pode ser adicionado nas configurações de teste e poderão ser vistos por todos os utilizadores.

[3] Ferramenta de gestão de trabalho para todos os tipos de casos de uso, desde gestão de requisitos e casos de teste até o desenvolvimento ágil de *software*.

3.3. Tricentis Tosca

A *Tricentis Tosca* [30] permite a criação de testes de software automatizados. Esta ferramenta permite:

- Testes de GUI (*Interface Gráfica do Utilizador*);
- Testes de interfaces de programação de aplicações (APIs);
- Testes de aplicações móveis;
- Testes de carga;
- Identificação e gestão dos dados de teste;
- Registo e simulação de serviços;
- Manutenção da qualidade dos dados em todas as etapas do ciclo de testes de *Data Werehouse*.

Para além da automatização de testes, esta ferramenta oferece uma avaliação de risco em tempo real focado no negócio. Isto inclui:

- Ajudar toda a equipa a expor rapidamente os defeitos críticos encontrados no software;
- Integração de testes diretamente no ambiente de integração contínua;
- Garantir a disponibilidade de dados de teste apropriados a qualquer momento;
- Simular o comportamento de sistemas dependentes, caso estes estejam indisponíveis durante a execução dos testes.

Fluxo de teste

Na *Tricentis Tosca* as etapas básicas do ciclo de vida de um teste são as seguintes:

- Identificar os critérios que o sistema em teste deve atender e criar uma estrutura de requisitos clara que reflita esses critérios;
- Projetar uma estrutura de teste lógica para ver o que é necessário para cobrir os seus requisitos. O utilizador tem disponível a funcionalidade de *design* de casos de teste para criar combinações possíveis de casos de teste;
- Criar módulos que contenham as informações técnicas essenciais que a *Tricentis Tosca* precisa para orientar o sistema em teste;
- Criar casos de teste concretos a partir dos módulos. Os casos de teste são uma série de etapas que verificam as especificações dos testes;
- Configurar os testes. Os utilizadores podem gerir de formal central os parâmetros de configuração dos testes para toda a equipa;
- Preparar e executar os testes;
- Após a execução dos testes, a *Tricentis Tosca* mapeia os resultados dos testes com os requisitos definidos anteriormente. Isto oferece uma visão geral do estado do teste e também do estado do sistema em teste.

Podem ainda ser adicionadas as seguintes etapas:

- Gerir os dados de teste;
- Planear os testes com a funcionalidade *Test Planning*;
- Criar e vincular problemas;
- Imprimir relatório do estado do projeto/sistema em teste.

Existe uma empresa chamada *Valori*, que se diz a número 1 em testes de QA e acredita que "num mundo em constante inovação, que depende de dados e tecnologia confiáveis, testes e *feedback* de especialistas são condições cruciais para a sobrevivência e crescimento futuro de uma empresa" [34], que utiliza a ferramenta *Tricentis Tosca* na execução dos testes e que, por sua vez, possui uma parceria com a *OutSystems* desde março de 2021. É por esta razão, que na secção de mercado do *site* da *Tricentis Tosca* é possível ver que existe um pacote chamado "*Valori Accelerator for OutSystems*", este pacote está disponível apenas para a *Valori* e pessoas autorizadas à sua utilização através de projetos *Outsystems* que tenham requisitado este serviço, pelo que não foi possível aceder ao seu conteúdo. Contudo, este pacote contém recursos que ajudam os *testers OutSystems* a acelerar e otimizar a automatização de testes.

O pacote fornece blocos de teste reutilizáveis para lidar com objetos *OutSystems* genéricos, ou seja, objetos (elementos usados em ecrãs, por exemplo, *inputs*) que são comuns em vários testes feitos em diferentes aplicações. Estes blocos de teste reutilizáveis contêm módulos genéricos com identificadores dinâmicos. Os *testers* podem então usar facilmente as nomenclaturas convencionais usadas em *OutSystems*, para identificar os objetos que desejam utilizar nos casos de teste [35].

A *Tricentis* criou uma biblioteca de *links* dinâmicos juntamente com a *Valori* que reúne os nomes padrão dos objetos da plataforma *OutSystems* usados nos sistemas em teste. Isso acelera muito a criação dos casos de teste porque a identificação dos objetos é feita durante a verificação inicial e, na maioria dos casos, não requer ação extra do utilizador para criar nomes lógicos para os objetos [35].

Em suma, este pacote permite identificar objetos e reutilizar a mesma lógica em todos os casos de teste. O facto de reutilizar os mesmos módulos predefinidos dinamicamente permite que não seja necessário que o *tester* verifique os objetos mais comuns, o que diminui bastante o tempo de realização do caso de teste. Este pacote também foi desenvolvido aplicando as melhores práticas descritas na próxima secção [35].

4. Melhores Práticas de desenvolvimento e a sua Influência nos Testes

Todas as linguagens de programação devem seguir as melhores práticas de desenvolvimento e arquitetura. Deste modo, o *developer* irá conseguir garantir que o código/módulo/aplicação desenvolvido possa ser reutilizado por outros serviços, possa ser sustentável e escalável.

Independentemente da tecnologia utilizada, os *developers* devem construir as suas aplicações de acordo com as melhores práticas, não invalidando o desenvolvimento ou a descoberta de novas. Programar de acordo com as melhores práticas faz com que o código possa ser testado mais facilmente pela equipa de qualidade e, deste modo, possam validar as funcionalidades implementadas e testadas.

Neste capítulo serão apresentadas algumas das melhores práticas existentes em *OutSystems* e que devem ser consideradas durante o desenvolvimento de todas as aplicações. Apesar de serem dados exemplos de acordo com a plataforma, estas práticas podem ser aplicadas a qualquer linguagem de programação.

4.1. Isolamento do domínio

Ao desenvolver em *OutSystems* deve-se ter em conta que diferentes conceitos e lógica de negócio devem corresponder a diferentes módulos. Cada módulo deve conter ações que definam a lógica e o conceito de negócio. Essa organização resultará na divisão da lógica de acordo com a regra de negócio correspondente ao módulo. De modo a promover o isolamento do código e, consequentemente dos testes, são recomendados os princípios de *Domain-Driven Design* (DDD) [16]. O DDD diz respeito ao desenvolvimento de sistemas complexos baseados em domínios independentes. Em *OutSystems*, e tendo em consideração as melhores práticas, cada domínio é representado por uma equipa do *LifeTime*. Cada uma das equipas terá associadas o conjunto de aplicações correspondente ao seu domínio de negócio que será independente dos outros domínios, e consequentemente, das outras equipas. Isto irá ter impacto na forma como serão executados os testes na aplicação. De seguida, serão apresentados alguns casos com mais detalhe.

4.1.1. Validações ao nível do UI

Na imagem seguinte, Figura 5, o ecrã do módulo de UI contém um formulário cuja finalidade é o registo de um utilizador que deve ser maior de idade para se registar na aplicação. A ação associada ao botão que submete o registo do utilizador contém todas as validações necessárias para guardar a informação e após estas validações é chamada a ação presente no módulo core que guarda o registo na base de dados. Posto isto, é possível ver que todas as validações acontecem no ecrã e a ação do *core* serve praticamente como um *wrapper* para a ação que cria o registo na base de dados.

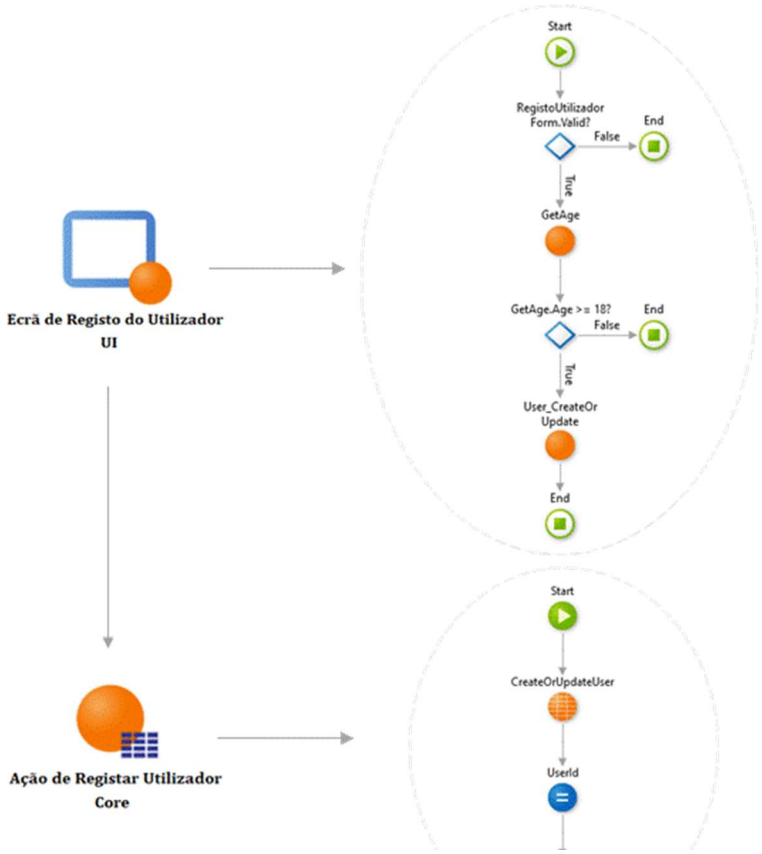

Figura 5 - Esquema do módulo de UI e Core sem aplicação das melhores práticas.

Se a ação do *core* for testada poderão ser submetidos quaisquer valores que esta irá ser sempre executada com sucesso, ou seja, mesmo que seja inserido um utilizador com menos de 18 anos, o teste será executado com sucesso. Isto faz com que qualquer teste que utilize esta ação durante a sua execução seja irrelevante pois, as validações necessárias não são feitas nesta ação, mas sim do lado do ecrã, no UI. Para resolver esta lacuna, todas as validações de lógica de negócio deverão ser movidas para a ação core de modo a garantir que as entradas fornecidas à ação estão de acordo com as regras de negócio daquele módulo.

As ações presentes nos módulos *core* devem ser testáveis e devem ter em conta todas as validações e verificações das regras de negócio de forma independente. Ao nível do UI também devem ser realizadas validações, mas estas devem ser mais focadas na interação do utilizador. Por exemplo, se o formulário é válido, ou não, e não tanto nas regras de negócio, como é o caso do facto de o utilizador ter de ser maior de idade para se registar na aplicação. A aplicação das melhores práticas nas ações mencionadas anteriormente pode ser vista na Figura 6.

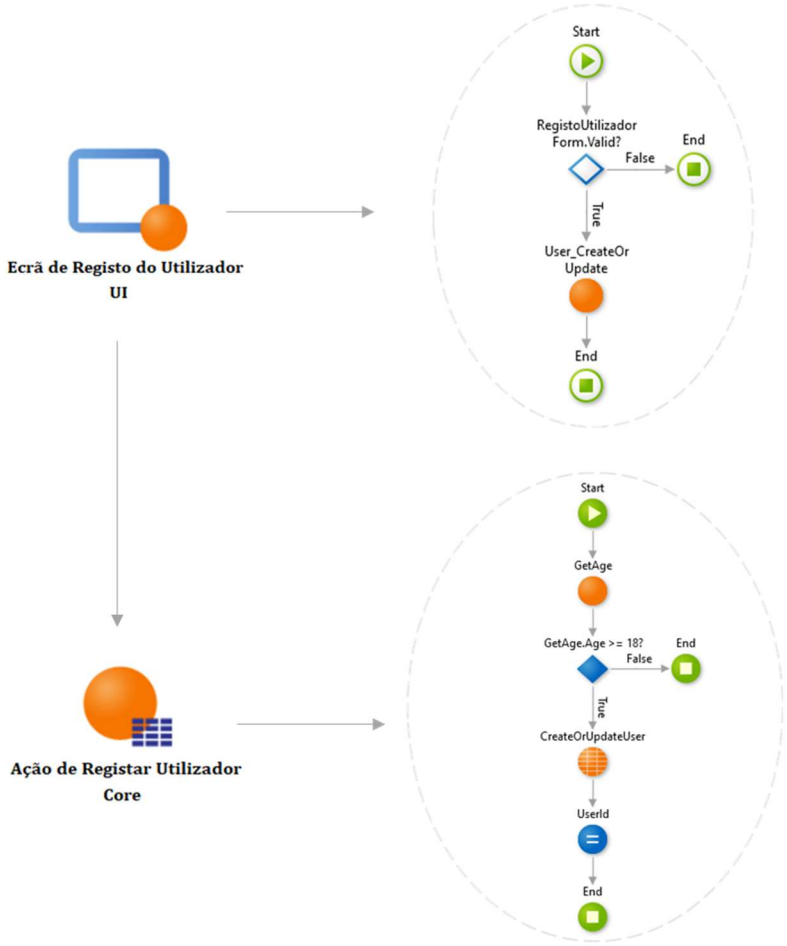

Figura 6 - Esquema do módulo de UI e Core com aplicação das melhores práticas.

4.1.2. Lógica de negócio no nível do UI

A ação do ecrã presente na Figura 7 é utilizada para adicionar produtos na lista de compras de um utilizador. Para ser possível adicionar os produtos é necessário que o utilizador tenha lista de compras e que os produtos existam no sistema.

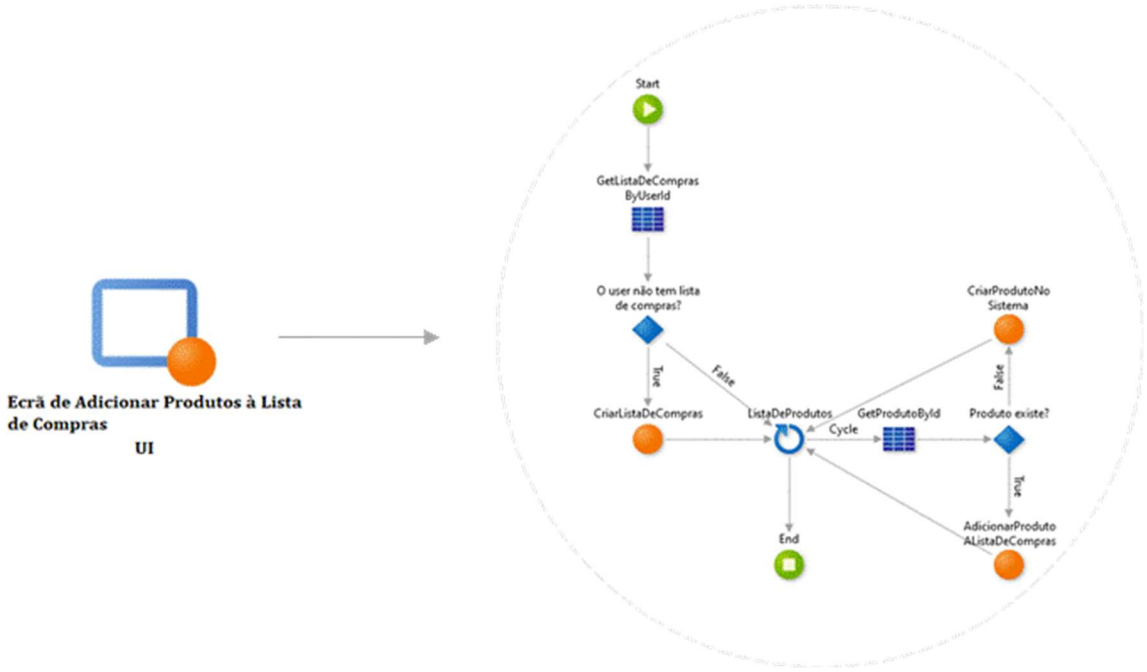

Figura 7 - Fluxo da ação do ecrã sem aplicação das boas práticas.

Para que esta lógica possa ser testada de forma completa, e como um todo, poderá acontecer um dos seguintes casos:

1. O teste terá de ser executado ao nível do UI, o que o tornará a execução potencialmente mais cara, especialmente porque os testes de UI geralmente são mais difíceis de manter e mais lentos de executar;
2. A lógica da ação teria de ser replicada na lógica da ação correspondente ao teste, o que obviamente não é a melhor opção porque desta forma a ação de teste teria de ser atualizada sempre que a ação do ecrã fosse alterada.

A melhor forma de lidar com este problema é encapsular a lógica que precisa de ser testada numa ação pública ao nível do módulo *Core*, que será usada na ação do ecrã. Assim, a lógica de negócio pode ser testada sem depender da implementação de testes de UI, que são muito mais caros de produzir e manter. Estas alterações podem ser visualizadas na Figura 8.

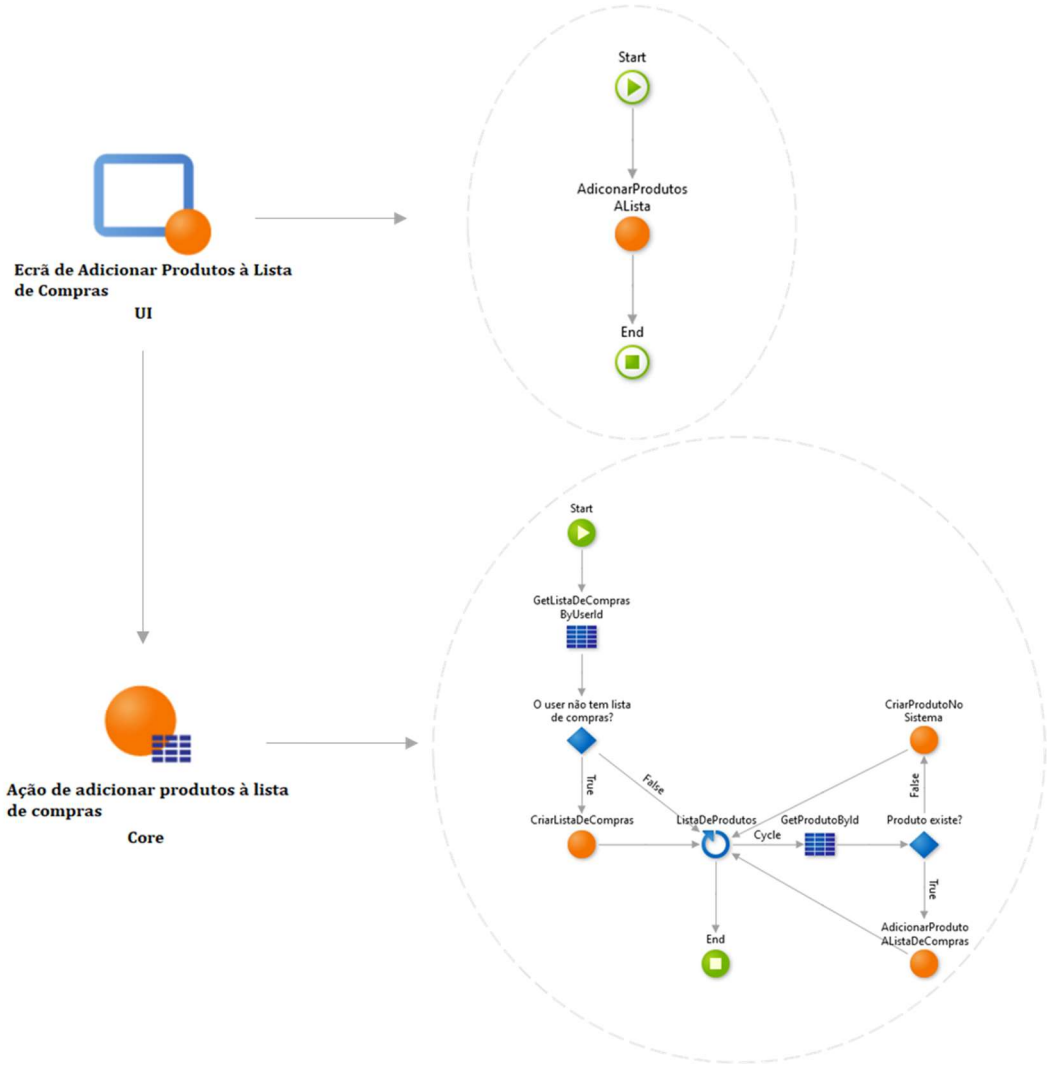

Figura 8 - Esquema do módulo de UI e Core com aplicação das melhores práticas.

4.1.3. Referência entre domínios incompatíveis

O *Domain-Driven Design* recomenda a utilização de ações de serviço (módulo *core*) e entidades públicas para dependências entre domínios ou módulos. Em *OutSystems* todas as entidades têm ações de *Create*, *Read*, *Update and Delete* (CRUD) (Figura 9). Estas ações podem ser utilizadas diretamente por outros módulos caso tenham o atributo "*Expose Read Only: No*". O facto de expor estas ações pode afetar a configuração dos dados de teste.

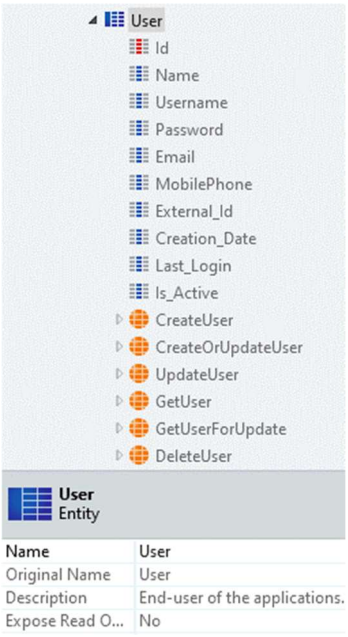

Figura 9 - Ações de CRUD e opção de *Expose Read Only*.

Existem algumas ações que podem ser feitas de modo a contornar este problema. Se as entidades entre módulos forem referenciadas diretamente, ou seja, se forem utilizadas diretamente as ações de CRUD, significa que não há possibilidade de simular dados de outro domínio. Como resultado, é aceitável que as atividades de configuração de teste no módulo local criem dados que sejam necessários diretamente no domínio externo.

No entanto, se essas entidades forem referenciadas apenas através de APIs REST/SOAP ou ações de serviço, isso possibilita a utilização a simulação de serviços para obter dados do domínio externo, desta forma será provida a independência e a confiabilidade do teste. Sendo assim, num cenário deste género não é aceitável que as atividades de configuração de teste no domínio local criem dados necessários à execução do teste no domínio externo. Deverá ser simulado um serviço de modo a serem obtidos os dados.

4.2. Isolamento de APIs

Sempre que se precisar de consumir um serviço REST/SOAP no módulo atual, deve ser feito através de um módulo *wrapper*. Este módulo deve ter um conjunto de ações públicas que utilizam essa API. Estas ações devem ser públicas e desta forma outros módulos que necessitem de utilizar essa API deverão fazê-lo através das ações públicas presentes no módulo *wrapper*.

As ações de serviço estão disponíveis desde a versão 11 da plataforma *OutSystems* e estas são essencialmente chamadas REST que estão disponíveis apenas para outras aplicações *OutSystems* que estejam em execução no mesmo ambiente.

4.2.1. Consumo da mesma API em vários módulos

Num ambiente *OutSystems* é normal a existência de vários módulos que consomem a mesma API. Se a mesma API estiver a ser consumida em vários módulos quer dizer que na presença de um *scope* de execução de testes, em que seja necessário simular serviços para isolar a aplicação em teste do sistema externo, onde a API é exposta, teriam de ser alterados todos os módulos que consumissem essa API específica. Isto seria bastante prejudicial para a manutenção pois sempre que a API fosse modificada teriam de ser alterados todos os módulos que a consumissem.

A melhor solução para este problema é isolar o consumo da API num módulo *wrapper* que expõe os métodos da API através de ações públicas. Os módulos que necessitarem de aceder à API devem fazê-lo através do *wrapper*. Posteriormente, no *scope* da execução de teste, quando é necessário apontar para a simulação de um serviço, isso deve ser feito apenas através do módulo *wrapper*. O *wrapper* deve também possuir lógica para retornar os *outputs* esperados para o teste de modo a poder ser utilizado em ações de lógica de negócio.

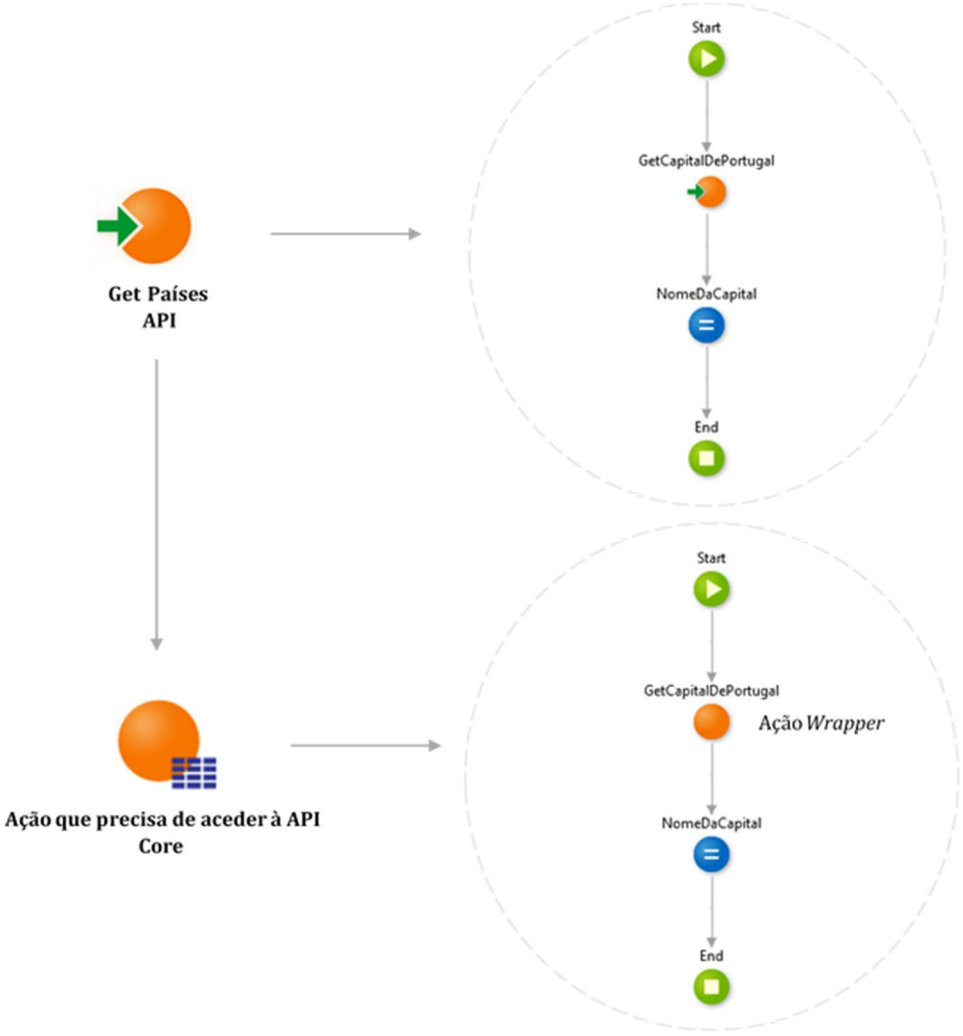

Figura 10 - Esquema do módulo de UI e *Core* com aplicação das melhores práticas.

4.3. Simulação de Web UI

4.3.1. Nomenclatura dos *widgets*

Para executar um teste de UI automatizado em aplicações *web* deve ser simulada a interação do utilizador através dos ecrãs da aplicação para cumprir os passos ou a funcionalidade a executar pelo utilizador. Esta simulação é feita através de um *script* que possui as várias etapas que um utilizador precisa de realizar nos elementos do ecrã. Por exemplo: Clique no *link* para "Iniciar Sessão"; Preencha os campos de texto correspondentes ao nome de utilizador e à palavra chave; Clique no botão "Iniciar Sessão".

Para que o teste seja executado corretamente o *script* de teste precisa de identificar todos os elementos com os quais o utilizador interage. A maneira mais simples de identificar um elemento num ecrã é através do seu identificador (ID). Isto implica que

todos os elementos presentes no ecrã tenham de ser devidamente identificados no *Service Studio* da plataforma *OutSystems*.

Deste modo, durante o desenvolvimento de um ecrã, o *developer* deve garantir que todos os elementos que possuam interação com o utilizador, tais como, *inputs*, botões e *links*, tenham a propriedade "*name*" atribuída com um valor significativo, ou seja, um nome que tenha a ver com a funcionalidade atribuída. Por exemplo um campo de texto correspondente ao nome de utilizador deve ter como nome "Input_NomeDeUtilizador". Isto vai facilitar a identificação do elemento no código html do ecrã uma vez que o ID do elemento irá sempre terminar com o valor dado à propriedade "*name*" do elemento no *Service Studio*.

Eventualmente, pode ser que um teste de UI específico exija elementos adicionais no ecrã, corretamente identificados. No entanto, ao identificar pelo menos os elementos que possuem interação com o utilizador, o *developer* irá minimizar o número de vezes que o trabalho precisa de ser feito. Este esforço faz com que seja mais rápida a implementação um teste de UI.

O facto de os *developers* identificarem os elementos com um nome permite também que a plataforma detete colisões de nomes dentro do mesmo ecrã. Este é um benefício adicional. A plataforma não permite que dois elementos diferentes tenham a mesma propriedade "*name*", portanto, ao segundo elemento com o mesmo nome a plataforma adicionará o sufixo "2" e assim por diante.

Ainda assim, como a *OutSystems* permite construir um ecrã com vários blocos, se for utilizado o mesmo nome em dois blocos diferentes para o mesmo ecrã, esta colisão de nomes não será automaticamente tratada pelo *Service Studio*. Tendo isto em consideração, é recomendado que, ao identificar um elemento num bloco, os *developers* incluam o nome do bloco como um prefixo para o nome do elemento, por exemplo, num bloco chamado "EditarPerfil" o *input* correspondente ao nome do utilizador poderá chamar-se EditarPerfil_InputNomeDoUtilizador. Isto garante que os nomes sejam exclusivos quando um ecrã for composto por vários blocos diferentes. Contudo, pode existir um cenário em que haja necessidade de ter várias instâncias do mesmo bloco num ecrã. Nestes cenários, os *developers* devem envolver cada bloco num *container* (div) com um ID específico e, em seguida, usar o ID da div para identificar cada instância do bloco de uma maneira exclusiva para este ecrã.

As ferramentas de teste geralmente usam seletores *XPath* ou CSS para identificar cada elemento do ecrã. Ao seguir a abordagem mencionada acima para identificação de um elemento do ecrã, esses elementos terão todos identificadores, mas a plataforma *OutSystems* irá gerar um ID "composto" que terminará sempre com o nome real dado ao elemento dentro do *Service Studio*. Na Figura 11 é possível ver os identificadores gerados pelo html após os elementos serem identificados no *Service Studio*.

Figura 11 - Identificadores compostos após ser dado nome aos elementos.

De seguida, é apresentado um exemplo simples de como o utilizador pode escrever um seletor que permita identificar os elementos que terminam com um determinado nome exclusivo. O nome representa os seletores CSS e *XPath* para um *input* com o nome "Input_NomeDoUtilizador" no *Service Studio*.

CSS Selector

```
input[id$="Input_NomeDoUtilizador"]
```

XPath Selector (1.0)

```
//input[substring(@id, string-length(@id) - string-
length('Input_NomeDoUtilizador') + 1) =
'Input_NomeDoUtilizador']
```

XPath Selector (2.0) não costuma ser suportado por ferramentas de teste

```
//input[ends-with(@id, 'Input_NomeDoUtilizador')]
```

4.3.2. Alternativa da "*Extended Property*"

Como alternativa ao uso da propriedade "name" para identificar os elementos, uma *extended property* chamada "os-test-id" pode ser utilizada como identificador. Para o mesmo *input* usado no exemplo anterior, ao adicionar a *extended property* a "os-test-id" com o valor "Input_NomeDoUtilizador" no *Service Studio*, Figura 12, a plataforma adicionará um atributo com o par nome/valor mencionado.

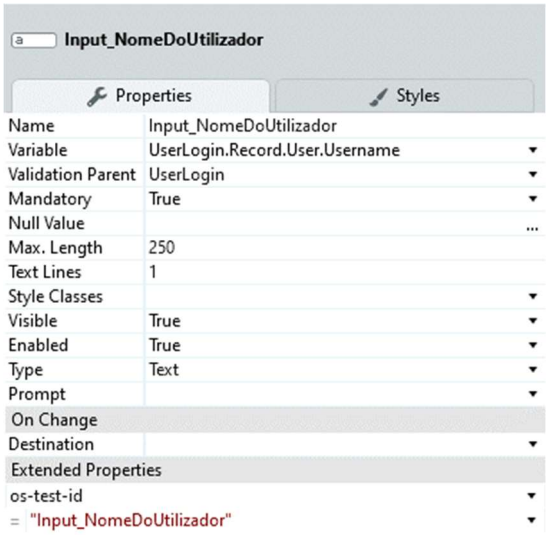

Figura 12 - *Extended Property* num elemento no *Service Studio*.

Para identificar os elementos no ecrã devem ser utilizados os seguintes seletores:

CSS Selector

```
input[os-test-id='Input_NomeDoUtilizador']
```

XPath Selector

```
//input[@os-test-id='Input_NomeDoUtilizador']
```

Esta abordagem permite o controlo completo dos elementos no ecrã, mas apresenta duas desvantagens principais que são:

1. Os *developers* têm de começar a adicionar essas *extended properties* nos elementos no *Service Studio* o que apresenta mais esforço de desenvolvimento. Dar um nome aos elementos pode parecer mais natural;
2. A plataforma *OutSystems* não fornece nenhuma deteção de colisão para as *extended properties*, portanto, cabe ao *developer* garantir isso, o que pode ser especialmente complicado em ecrãs mais complexos.

4.3.3. Mapear os testes para o modelo de 4 camadas (*4 Layer Canvas*)

O *4 Layer Canvas* (4LC) [36] é uma ferramenta de arquitetura *OutSystems* para simplificar o *design* de arquitetura de um projeto.

4LC promove a abstração correta de (micro)serviços reutilizáveis e o isolamento correto de módulos funcionais distintos. É útil nos casos em que o *developer* está a trabalhar e a manter várias aplicações que têm módulos comuns.

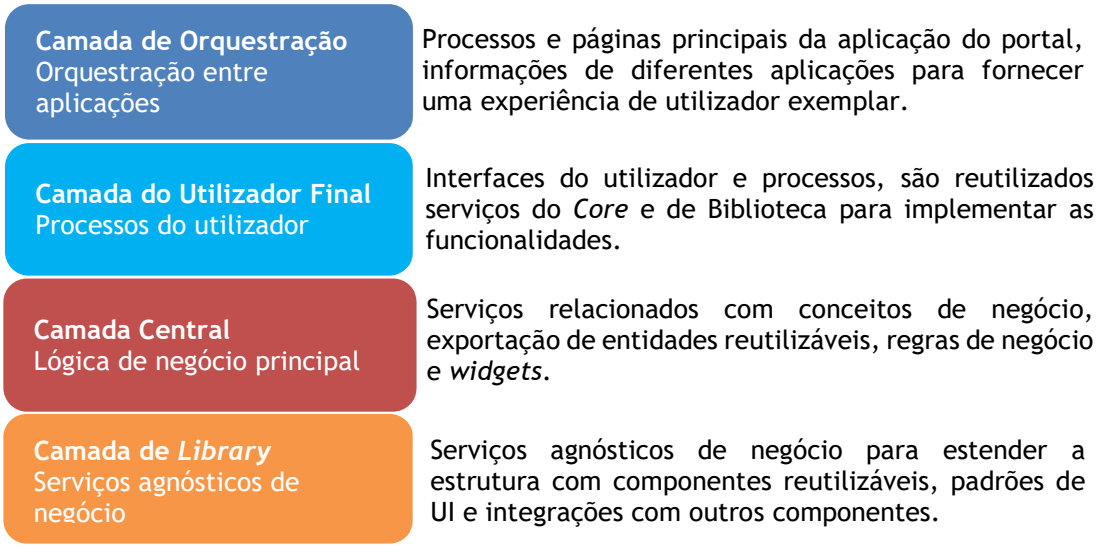

Figura 13 - Modelo de 4 camadas (4 *Layer Canvas*).

Em cada camada existem vários módulos que possuem a implementação de diferentes funcionalidades. Deste modo, para cada camada devem existir testes funcionais, que façam sentido, tendo em conta o nível e funcionalidade desenvolvida.

Camada de orquestração: Testes de UI automatizados

Os módulos de orquestração geralmente implementam ecrãs *cross-application*, ou seja, ecrãs que podem ser utilizados por várias aplicações, e que são mais orientados a informações e não tanto a formulários. Estes módulos fornecem *links* para aceder aos ecrãs da aplicação criados na camada do utilizador final. Desta forma, os testes implementados nas funcionalidades desta camada deverão ser testes de UI que reproduzem a navegação do utilizador que pode ser ou não entre aplicações.

Exemplo de teste:

1. Aceder ao *browser*;
2. Aceder à aplicação “eCommerce”;
3. Verificar que está na página de *login*;
4. Preencher os campos correspondentes ao nome do utilizador e à palavra-chave;
5. Clicar no botão “Iniciar Sessão”.

Camada do utilizador final: Testes de UI *end-to-end* automatizados

Os módulos que completam a camada de utilizador final possuem ecrãs específicos apenas para uma aplicação com um domínio de negócio muito bem definido. Estes tendem a ser mais orientados para informação e formulários e, também se concentram na lógica de negócio da aplicação. Como tal, os testes implementados nesta camada são testes de UI que validam a lógica de negócio implementada.

Exemplo de teste:

1. Fazer *login* na aplicação "eCommerce";
2. Ir para o ecrã "Lista de Produtos";
3. Selecionar o primeiro produto da lista;
4. Adicionar o produto selecionado ao carrinho;
5. Verificar que o carrinho de compras foi atualizado com o produto adicionado.

Camada central: Os módulos *core* implementam conceitos e lógica de negócio. Como as referências horizontais são permitidas na camada central, é estabelecida uma hierarquia entre os módulos centrais. Consequentemente, dependendo de quão alto ou baixo um módulo *core* está na hierarquia, é possível implementar diferentes tipos de testes para ele. Quanto mais baixo o nível, mais próximo o teste estará de um teste de unidade, enquanto os níveis mais altos promovem testes *end-to-end* que podem ter dependências de conceitos de vários módulos e até mesmo integrações com sistemas externos.

- **Teste automatizado *end-to-end* (sem UI)**

  Neste tipo de testes é validada a lógica de negócio chamando diretamente o código *OutSystems* desenvolvido, sem passar pelos ecrãs da aplicação e declarar *outputs* esperados. Na verdade, este tipo de teste pode testar vários componentes da aplicação. Existem duas abordagens que podem ser exploradas:

  1. Chamar explicitamente, numa sequência lógica, as ações que possuem a lógica a ser testada de modo a simular a navegação do utilizador no UI, para verificar um determinado resultado;
  2. Chamar uma única ação que encapsule a orquestração de vários conceitos de diferentes módulos para obter uma validação completa da interação do utilizador.

- **Teste de integração automatizado**

  Os testes de API são utilizados para validar a exatidão das integrações expostas, de acordo com as especificações. Estes testes podem ser feitos a APIs que são expostas de sistemas externos e consumidas dentro da plataforma *OutSystems* ou a APIs que são expostas por um módulo de uma aplicação *OutSystems* para serem consumidas externamente por outras aplicações ou sistemas. Estes testes são geralmente definidos por um conjunto de *inputs* para a API e serão obtidos *outputs* que devem ser analisados de modo a analisar a veracidade dos testes.

- **Teste automatizado de unidade/componente**
  Os testes de unidade/componente validam a lógica de negócio nos módulos de código de nível inferior. Estes módulos são independentes e não têm, praticamente, dependências para outros módulos principais. Por esse motivo, estes módulos normalmente contêm a lógica de negócio específica para um conjunto de conceitos definidos, sem contexto de quaisquer outros conceitos ou lógica de nível superior.

Camada de *Library*:

- **Teste de unidade/componente (não comercial) automatizado**

Os módulos de biblioteca são, por definição, agnósticos quanto à lógica de negócio. Desta forma, eles não contêm lógica de negócio. Mas, uma vez que estes módulos fornecem componentes altamente reutilizáveis por outras aplicações *OutSystems*, pode ser útil testar as funcionalidades fornecidas pelos mesmos. Estes módulos podem conter, por exemplo, sistemas de auditoria onde cada ação executada por um utilizador em qualquer aplicação é registada numa entidade de auditoria.

5. Implementação e execução de testes em *OutSystems*

Para demonstrar e compreender como a utilização das boas práticas, durante o desenvolvimento em *Outsystems*, tem influência no processo de automatização de testes, e para perceber as suas implicações em diferentes ferramentas de teste, neste capítulo descreve-se um conjunto de cenários de teste para uma aplicação desenvolvida em *OutSystems*, os quais são implementados e executados com as 3 ferramentas de teste selecionadas.

5.1. A aplicação a testar: "*The Wine Club*"

A aplicação escolhida para exemplificar a implementação e execução dos testes chama-se "*The Wine Club*" [37] e é uma aplicação que implementa uma loja de venda de vinhos *online* que foi desenvolvida em *OutSystems*. Nesta secção são descritas algumas das suas funcionalidades principais.

Funcionalidades principais

Um utilizador do "*The Wine Club*" pode ver a lista de vinhos disponível no site bem como filtrar a mesma pelos tipos de vinhos. Na Figura 14 é possível visualizar a página principal do clube onde está a lista de vinhos que inclui imagem e o preço do produto. No início da página também está presente uma área de filtros onde o utilizador pode filtrar a lista pelos tipos de vinhos ou até pesquisar por vinhos específicos.

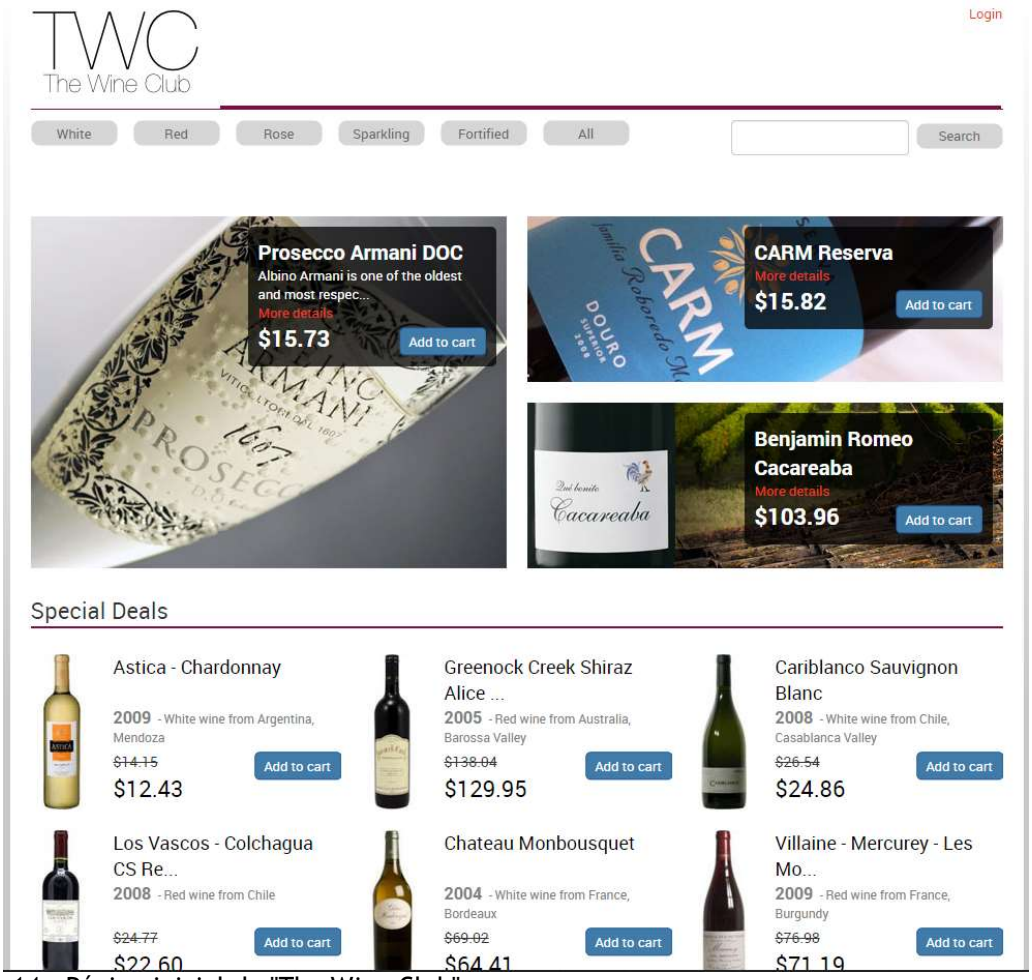

Figura 14 - Página inicial do "The Wine Club".

O utilizador pode também ver o produto com mais detalhe e conhecer também opiniões de outros clientes, caso aceda à página de detalhe do produto. Para isso, basta clicar no produto e irá aceder à página que é possível ver na Figura 15.

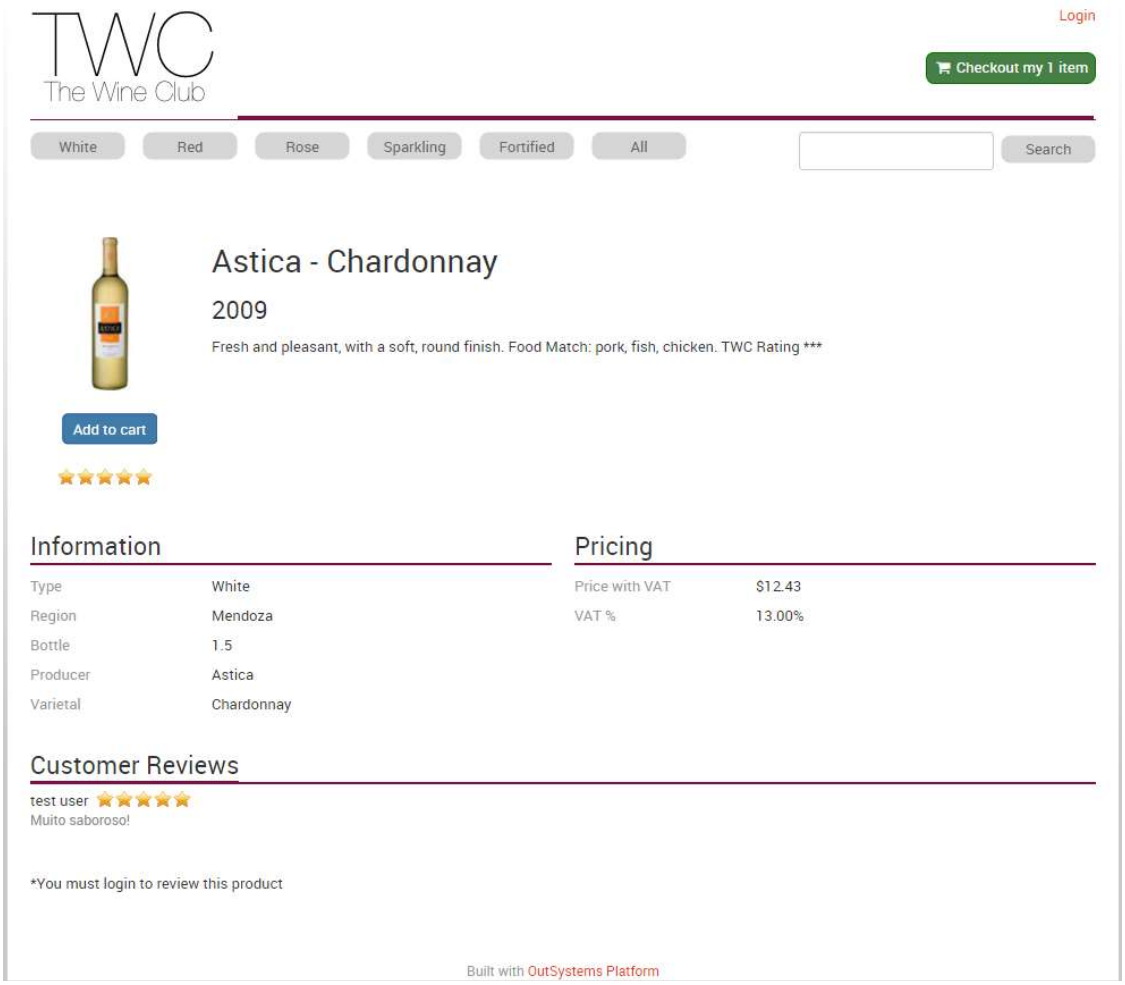

Figura 15 - Página de detalhe do produto.

Para proceder à compra dos vinhos o utilizador apenas necessita de adicionar os produtos ao carrinho de compras, carregando no botão "*Add to cart*". Na Figura 16 é possível ver a página correspondente ao carrinho de compras do "*The Wine Club*", podendo-se visualizar todos os produtos adicionados ao carrinho de compras, bem como o valor da compra e o valor referente ao envio dos produtos.

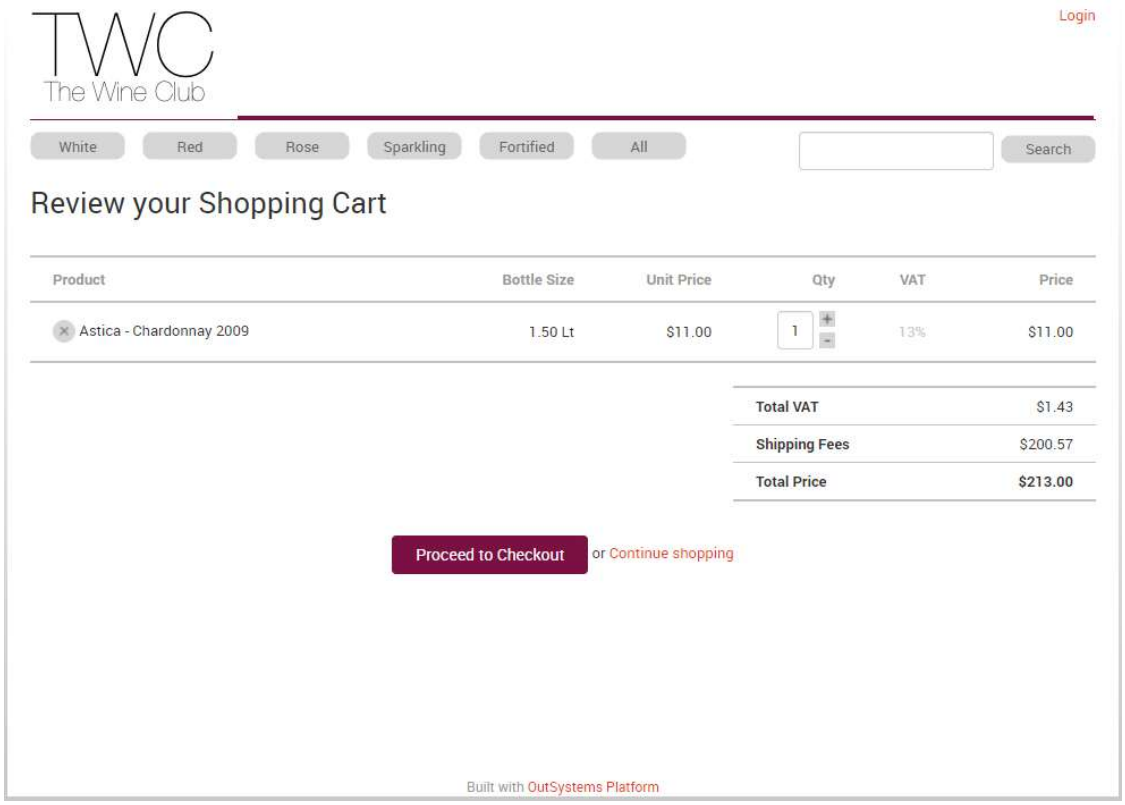

Figura 16 - Página do carrinho de compras do "The Wine Club".

Para iniciar o processo da compra o utilizador deve clicar no botão "*Proceed to Checkout*" e irá ser direcionado para uma página onde poderá fazer *login*, caso já tenha conta na aplicação, ou então deverá registar-se para realizar a sua primeira compra na aplicação. Caso não possua conta, o utilizador apenas tem de fornecer informação como o nome, email e palavra-passe, o que torna este processo bastante rápido e simples.

Após o registo ou *login*, o utilizador será direcionado para a página onde pode ter uma visualização total da sua encomenda, preço total e o endereço para qual esta vai ser enviada como é possível ver na Figura 17.

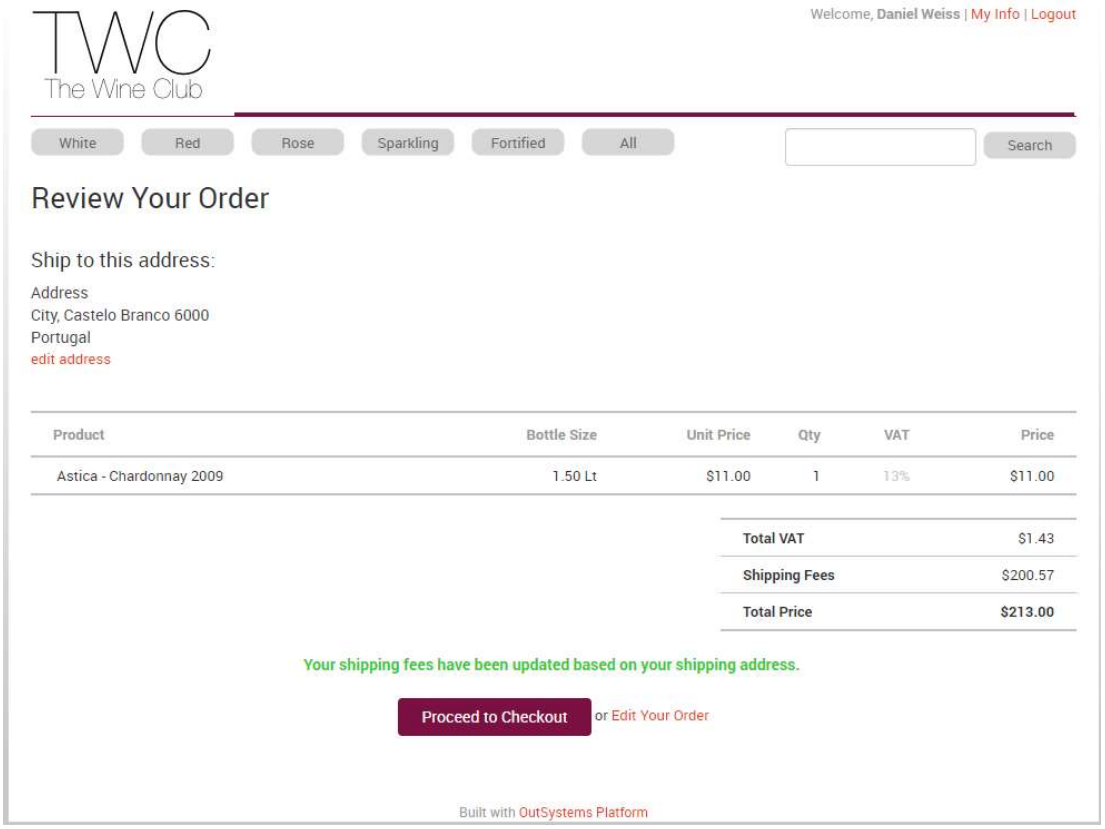

Figura 17 - Visão geral da encomenda.

De modo a finalizar o processo o utilizador apenas terá de preencher um formulário com os seus dados de pagamento.

5.2. Descrição dos cenários de teste

De seguida serão apresentados três cenários de teste que serão implementados com as ferramentas de teste apresentadas no capítulo 3. Com os cenários de teste apresentados pretende-se exemplificar como a utilização das boas práticas no desenvolvimento em *OutSystems*, apresentadas no capítulo 4, têm influência no processo de teste do software.

Os cenários de teste são escritos no estilo BDD de forma a simplificar a sua escrita e leitura.

1º Cenário - Fazer *login* na aplicação

O primeiro cenário corresponde ao início de sessão na aplicação "*The Wine Club*". O utilizador deve aceder à página principal da aplicação, Figura 14, e clicar em "*Login*" de modo a ser redirecionado para o ecrã de início de sessão onde poderá preencher os dados com as suas credenciais.

O cenário de teste pode ser escrito da seguinte forma:

Dado:

Que o utilizador tem conta no "*The Wine Club*"

Quando:

E clica em "*Login*"

E preenche o email e a palavra chave

E clica em "*Sign in*"

Então:

A operação deve ser bem-sucedida

E o utilizador deve ser redirecionado para a página principal da aplicação

2º Cenário - Adicionar um produto ao carrinho

O segundo cenário de teste corresponde a um teste à funcionalidade de adicionar um produto ao carrinho. Neste cenário pretende-se adicionar o produto "*Astica - Chardonnay*" ao carrinho, por parte de um utilizador com sessão iniciada na aplicação. Para simular este cenário o utilizador deve estar na página que contém a lista de produtos, Figura 14.

Dado:

Que o utilizador tem um carrinho

E existe um produto com o nome "*Astica - Chardonnay*"

Quando:

O utilizador adiciona o produto ao carrinho

Então:

A operação deve ser bem-sucedida

E o carrinho deve ser atualizado corretamente

3º Cenário - Validar que Lisboa é a capital de Portugal (API)

O terceiro cenário corresponde a um cenário independente da aplicação "*The Wine Club*", uma vez que o objetivo é exemplificar o teste a uma API e a aplicação "*The Wine*

Club" não usa API externas. Para este cenário o utilizador apenas tem de introduzir o país Portugal de modo a obter a sua capital.

Dado:

Que existe um país chamado "Portugal"

Quando:

O utilizador solicita os dados sobre Portugal

Então:

A cidade de Lisboa deve ser dada como a sua capital

5.3. Implementação e execução dos testes com diferentes ferramentas

Nesta secção será apresentada a execução dos cenários de teste definidos anteriormente nas ferramentas escolhidas.

5.3.1. Usando a *BDDFramework*

Para começar a executar os testes, os *developers* devem começar por adicionar na aplicação um módulo de testes. Neste caso, foi adicionado o módulo *eCommerceTests* como é possível ver na Figura 18. É neste módulo que serão criados os ecrãs para cada cenário de teste. Adicionar um módulo especifico para conter toda a lógica de testes segue uma boa prática mencionada no capítulo anterior, secção 4, uma vez que, desta forma, a lógica correspondente aos testes fica totalmente isolada da lógica implementada para o desenvolvimento da aplicação.

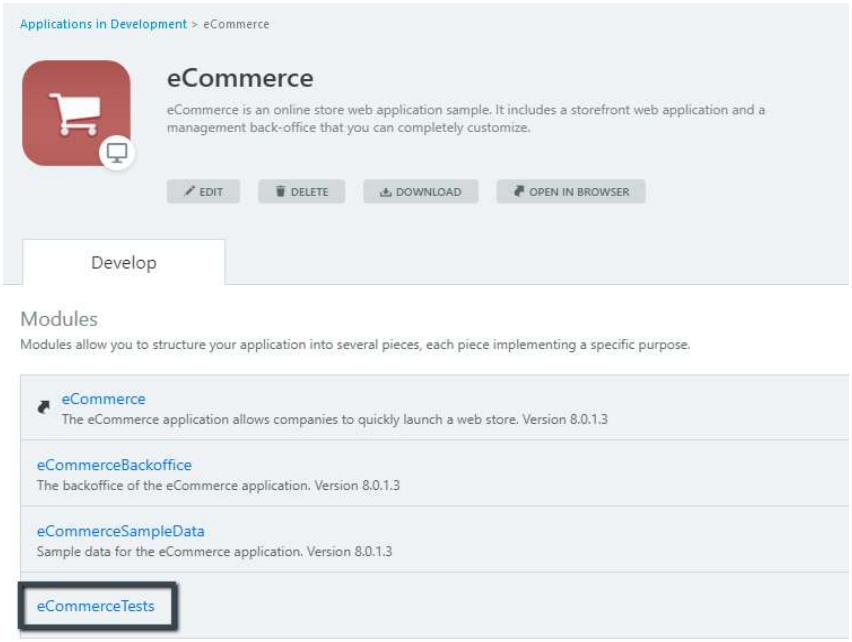

Figura 18 - Visão dos módulos da aplicação a testar.

1º CENÁRIO

Nesta secção descreve-se a implementação e execução do caso de teste correspondente à funcionalidade "Fazer *login* na aplicação".

Implementação do teste

Para o primeiro cenário o *developer* deve começar por criar o ecrã que diz respeito a este cenário, neste caso "*Login*". Este ecrã será então composto pelos blocos da *BDDFramework*, ou seja, o BDD cenário que dentro dele terá um bloco de *setup*, três blocos de *steps*, um bloco de *teardown* e, por fim, um de final *result*. Na Figura 19 é possível ver a estrutura do ecrã no *Service Studio*.

Figura 19 - Ecrã de teste do 1ºcenário "*Login*" no *OutSystems Service Studio*.

Para fazer *login* na aplicação é necessário existir um utilizador criado. Para isso, o bloco de *setup* terá associada uma ação que contém a lógica que permite criar a conta de um utilizador, conforme ilustrado na Figura 20. Assim, o *developer* apenas terá de utilizar ações que foram previamente implementadas no módulo da aplicação. Por exemplo, como neste caso é necessário criar um utilizador pode-se utilizar a ação do ecrã de registo de um utilizador "*RegisterUser*". Para além disso, serão guardados em variáveis valores que possam ser úteis nos próximos passos, como os identificadores do utilizador e do cliente e a palavra-chave.

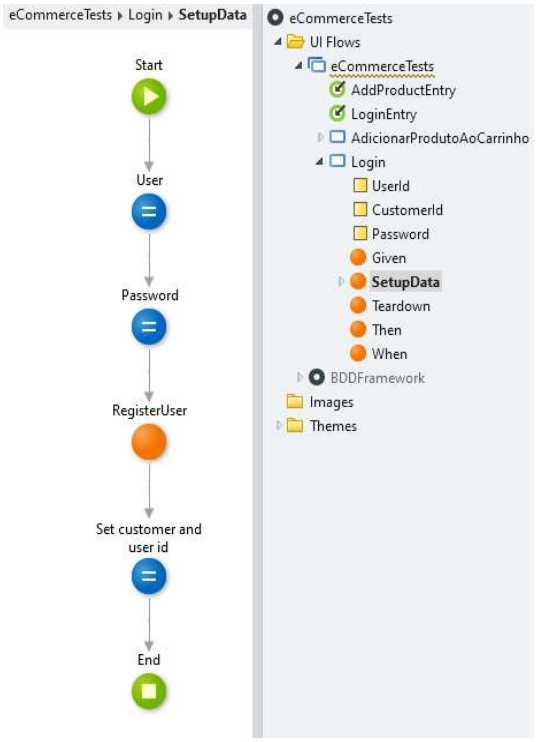

Figura 20 - Lógica de *setup* dos dados para o primeiro cenário.

Uma vez preparados os dados necessários ao caso de teste, pode-se então prosseguir para a primeira etapa "Dado: que o utilizador tem conta no '*The Wine Club*'". Esta etapa é implementada num bloco "*BDDStep*" que terá associado uma ação que contém a lógica para verificar se existe um utilizador na aplicação. Neste caso, a variável ao nível do ecrã que irá servir para guardar o identificador do utilizador criado no passo de *setup* será usada para verificar se este utilizador está criado na aplicação.

A Figura 21 mostra a lógica associada à primeira etapa do cenário, associada ao primeiro "*BDDStep*". Nesta ação é feita uma chamada à base de dados, mais precisamente à tabela *Customers*, onde é feita uma pesquisa pelo identificador do utilizador criado anteriormente. Caso o utilizador exista utilizam-se as ações de verificação que a *BDDFramework* oferece. Neste caso, é verificado se é devolvido um

registo da tabela e se o identificador devolvido é igual ao que foi criado anteriormente e que está guardado na variável local. Caso contrário a verificação irá falhar.

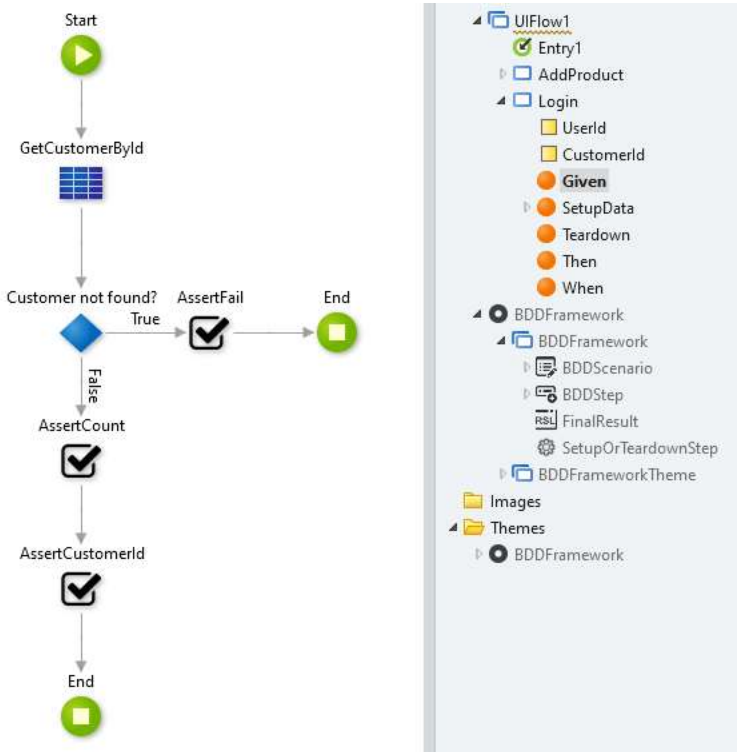

Figura 21 - Lógica da primeira etapa do cenário.

A segunda etapa é constituída por três passos:

"Quando: E clica em '*Login*';

E preenche o email e palavra-chave;

E clica '*Sign in*'".

Neste caso, e visto que a BDD*Framework* é utilizada para testes de componentes, ou seja, testes focados apenas na lógica de negócio, não é possível simular os cliques nem verificar se o email ou palavra-chave foram preenchidos, visto que não há acesso ao ecrã onde está o formulário de *login*. Assim, apenas é pesquisado o utilizador através do identificador do utilizador criado anteriormente, e a informação retornada, nomeadamente o ***username*** e a **palavra-chave**, será utilizada na chamada à ação de *login* que permite a simulação do *login* do utilizador na aplicação. Caso o sistema lance algum tipo de exceção nesta ação quer dizer que o *login* não foi executado com sucesso e o teste falhará.

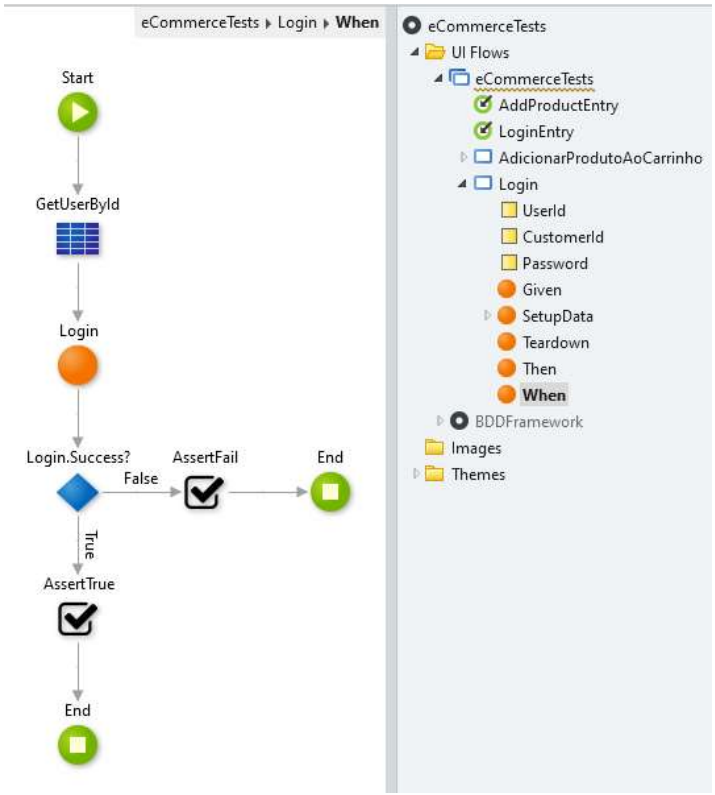

Figura 22 - Lógica da segunda etapa do cenário.

A última etapa do cenário "Então: A operação deve ser bem-sucedida; E o utilizador deve ser redirecionado para a página principal da aplicação" corresponde a um redireccionamento para uma página, após o *login*. Visto que não há acesso aos ecrãs da aplicação, e que esse não é o objetivo dos testes executados através desta *framework*, nesta ação não é executada nenhuma lógica.

Por último, existe um passo chamado "*Teardown*", Figura 23, que, neste caso, é utilizado para eliminar os dados criados para a realização do teste, de modo a não ser criado muito "lixo" na base de dados da aplicação. Neste caso, será então apagado o utilizador criado, assim como o *customer* associado. Será também feito o *logout* da aplicação, visto que foi iniciada sessão na aplicação anteriormente.

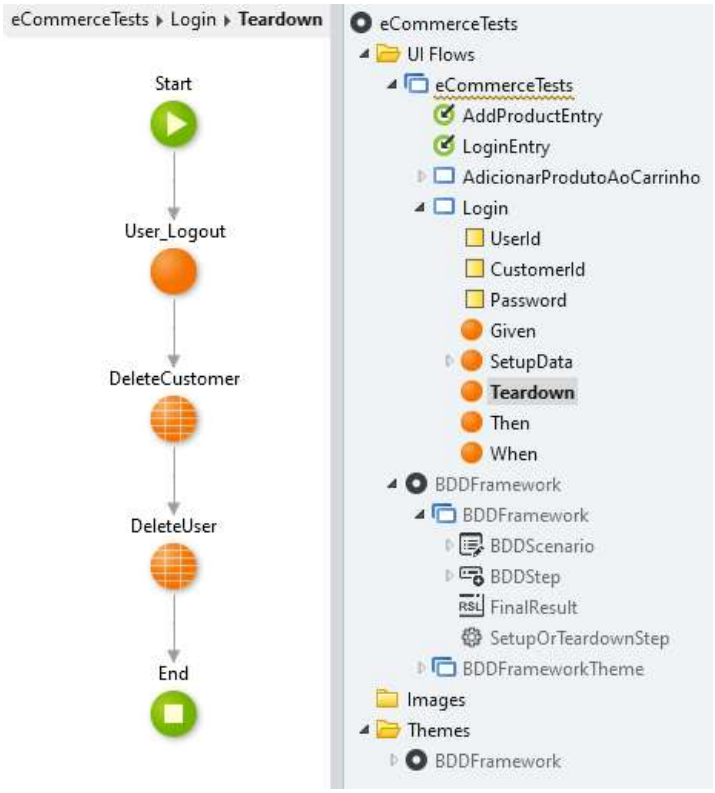

Figura 23 - Lógica de *teardown* do primeiro cenário.

Execução do teste

Ao abrir no *browser* o ecrã de teste correspondente ao *Login*, será visualizado todo o cenário de teste, dividido por passos onde é possível verificar se cada passo e se o teste em geral foi executado com sucesso.

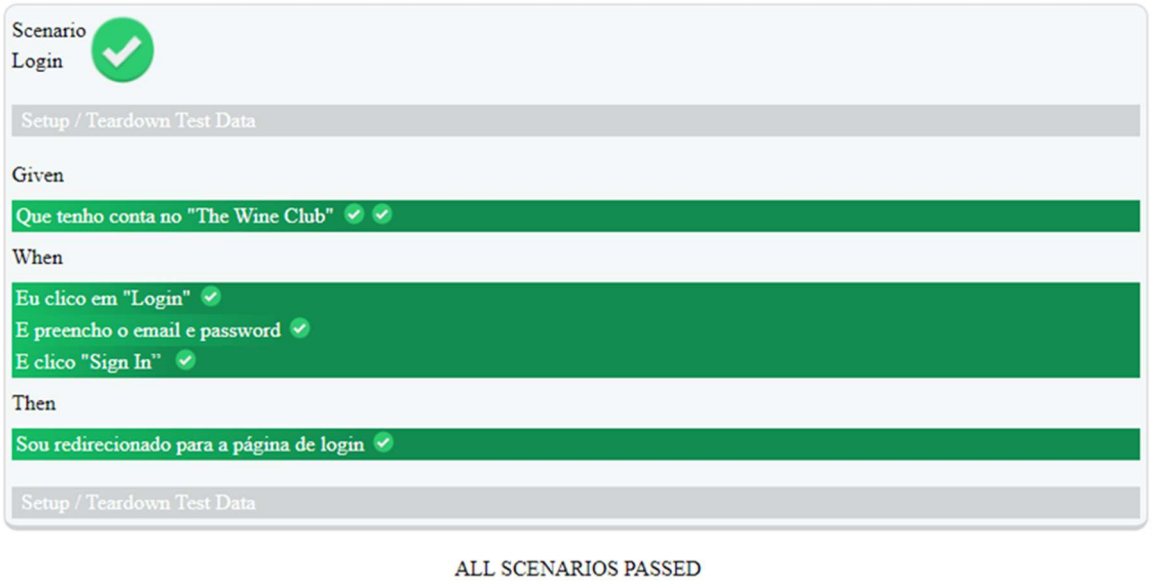

Figura 24 - Primeiro cenário executado com sucesso.

A Figura 24, mostra o resultado do teste, em que todos os passos do teste foram executados com sucesso. Já a Figura 25, mostra o resultado da execução do caso de teste, mas por alguma razão as credenciais do utilizador foram introduzidas de forma incorreta algumas vezes, mostrando assim a mensagem de erro que causou da falha do passo *"Too many failed login attempts. Please try again in 60 minutes."*. Como esta mensagem não corresponde ao resultado esperado, então esse passo do teste falhará e todos os passos seguintes são ignorados.

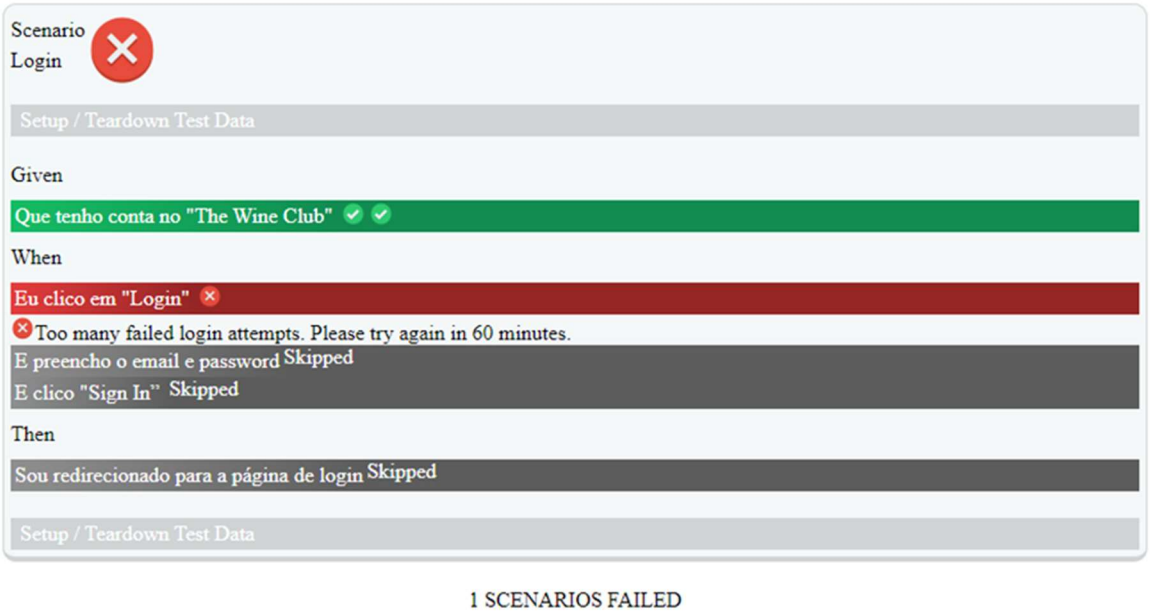

Figura 25 - Falha na execução do primeiro cenário.

2º CENÁRIO

O 2º cenário de teste corresponde ao cenário "Adicionar um produto ao carrinho". A sua implementação e execução são descritas a seguir.

Implementação do teste

Da mesma forma que foi feito para o primeiro cenário, o *developer* deve começar por criar o ecrã que diz respeito a este cenário, neste caso "AdicionarProdutoAoCarrinho". O ecrã será composto pelos blocos da *BDDFramework*, ou seja, o BDD cenário que dentro dele terá um bloco de *setup*, três blocos de *steps*, um bloco de *teardown* e por fim um de final *result*. Na Figura 26 é possível ver a estrutura do ecrã no *Service Studio*.

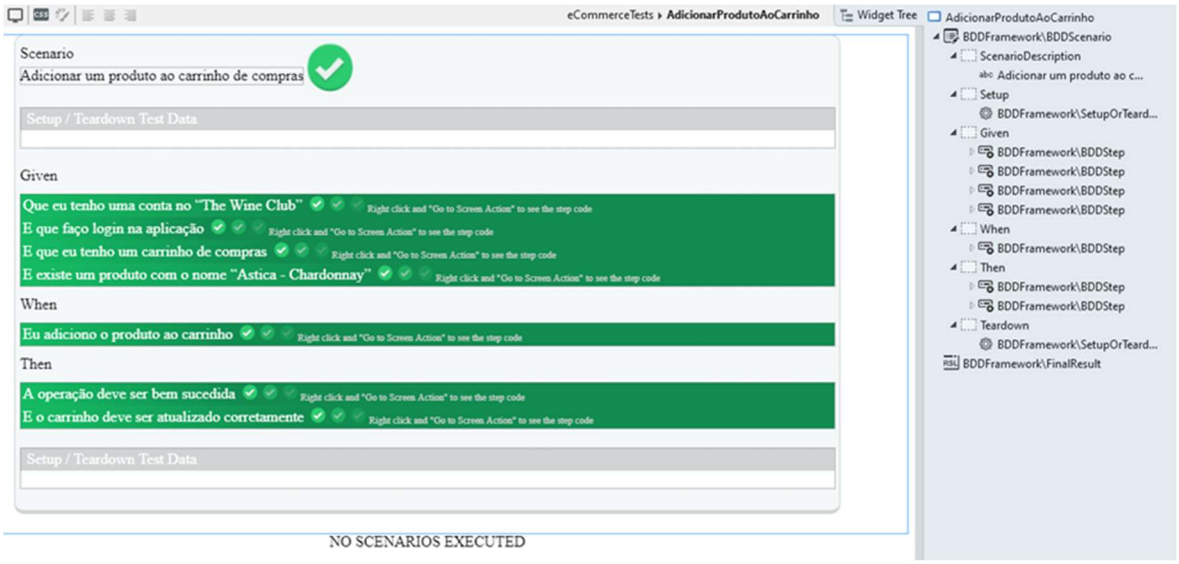

Figura 26 - Estrutura do ecrã de teste do segundo cenário no *OutSystems Service Studio*.

Para a execução deste cenário é necessário um utilizador que esteja registado. Deste modo, na ação de preparação dos dados, Figura 27, será incluída a lógica para o registo do utilizador, de forma semelhante ao que foi feito no cenário anterior. Será também necessário criar um carrinho de compras, para adicionar o produto. Para isso, é apenas necessário chamar a ação desenvolvida anteriormente para a aplicação, "*Cart_CreateNew*", que contém a lógica de criar um carrinho de compras. Nesta ação são guardados também alguns valores em variáveis para que possam ser utilizados posteriormente em pesquisas de base de dados e também, para no final ser possível eliminar os dados criados.

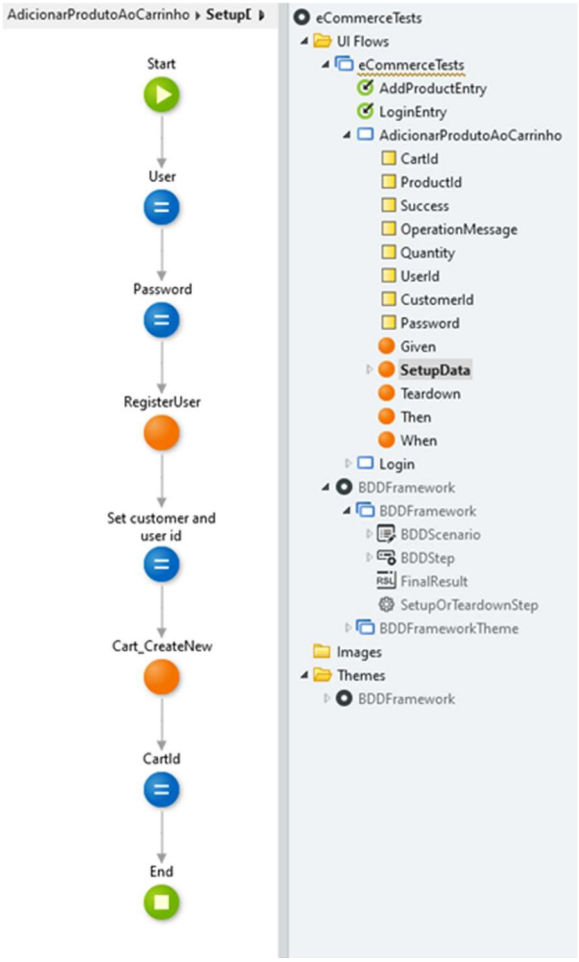

Figura 27 - Lógica de *setup* de dados do segundo cenário.

Uma vez preparados os dados necessários à execução do teste, será necessário implementar o primeiro passo da primeira etapa "Dado que tenho conta no '*The Wine Club*'". Esta etapa corresponde a um bloco "*BDDStep*" que terá associada uma ação que contém a lógica para verificar se existe um utilizador na aplicação, como foi igualmente executado para o primeiro cenário, como foi possível ver na Figura 28. Neste caso, será realizada uma pesquisa na base de dados pelo identificador do utilizador criado na ação de *setup* de dados que foi guardado na variável "*UserId*". Caso o utilizador seja retornado, o passo será executado com sucesso se não, irá falhar.

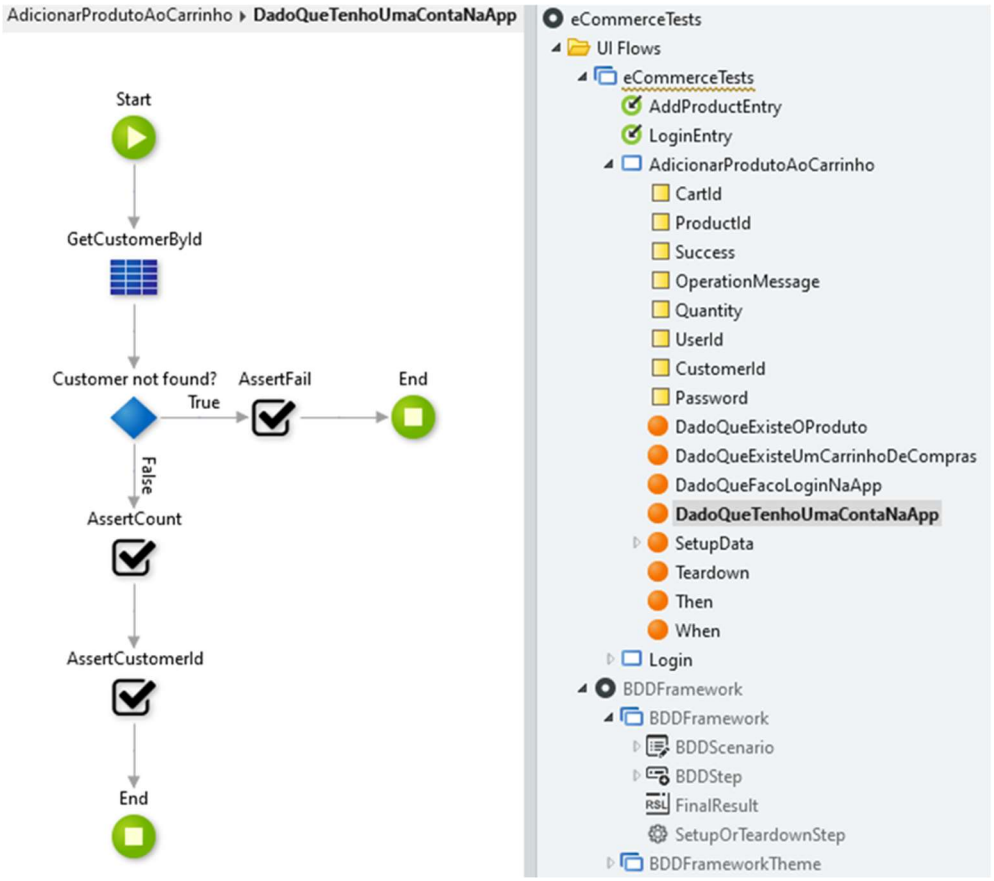

Figura 28 - Lógica do primeiro passo da primeira etapa do segundo cenário.

O segundo passo da primeira etapa “E faço *login* na aplicação”, terá associada uma ação que contém a lógica necessária para que o utilizador criado no passo anterior fará *login* na aplicação, como é possível ver na Figura 29. Caso o *login* seja executado com sucesso o passo será também executado corretamente se não, irá falhar.

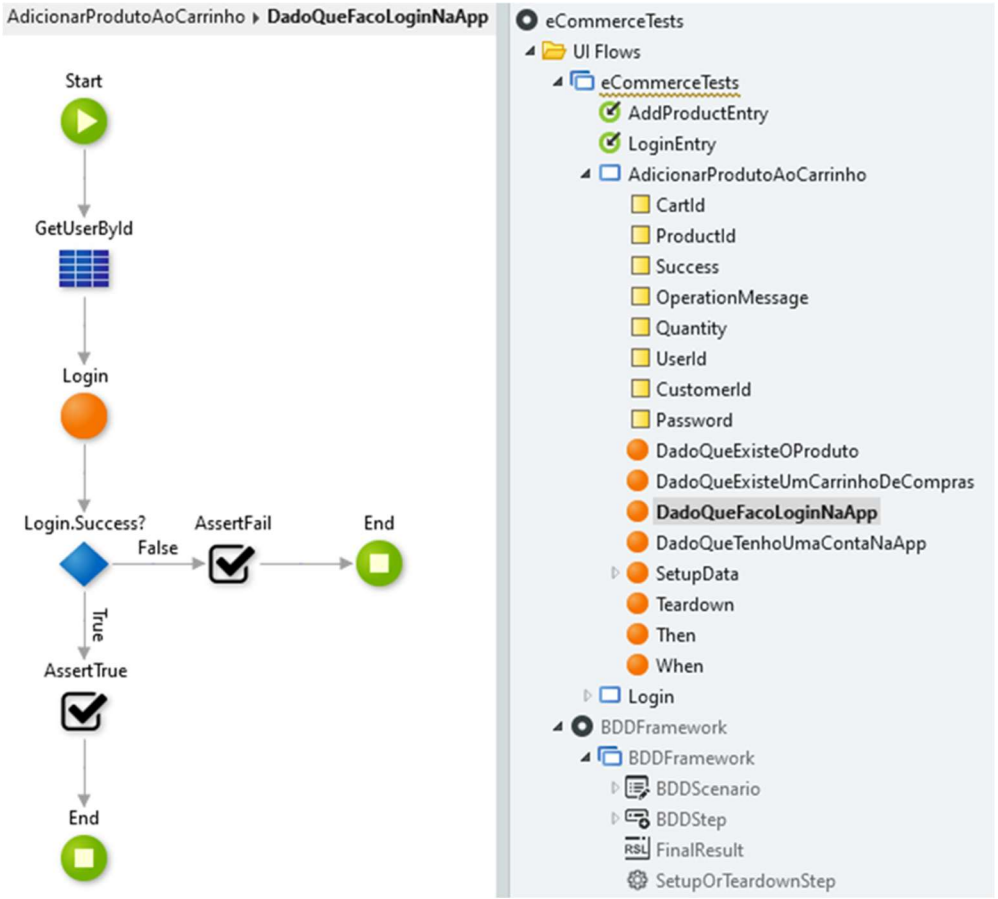

Figura 29 - Lógica do segundo passo da primeira etapa do segundo cenário.

A ação associada (Figura 30) ao terceiro passo da primeira etapa do segundo cenário "E que tenho um carrinho de compras" contém a lógica que irá verificar se o carrinho criado existe ou não na base de dados. Para isto, será feita uma pesquisa na base de dados pelo identificador do carrinho criado anteriormente que foi guardado na variável "*CartId*", se existir o passo será executado com sucesso, se não falhará.

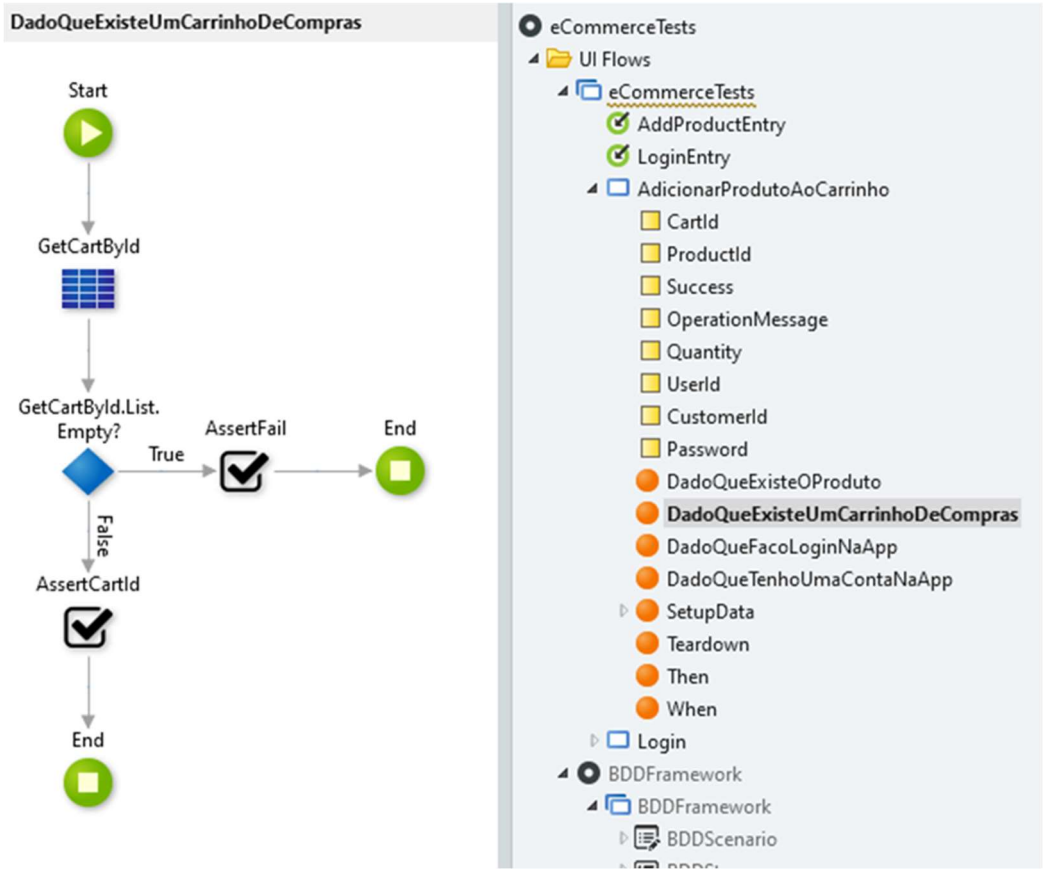

Figura 30 - Lógica do terceiro passo da primeira etapa do segundo cenário.

O último passo da primeira etapa "E que existe um produto com o nome '*Astica – Chardonnay*'", pretende verificar se o vinho chamado *Astica – Chardonnay* existe na lista de produtos. Para isto, é feita uma pesquisa na base de dados pelo nome do produto e caso este exista o passo será executado com sucesso (Figura 31).

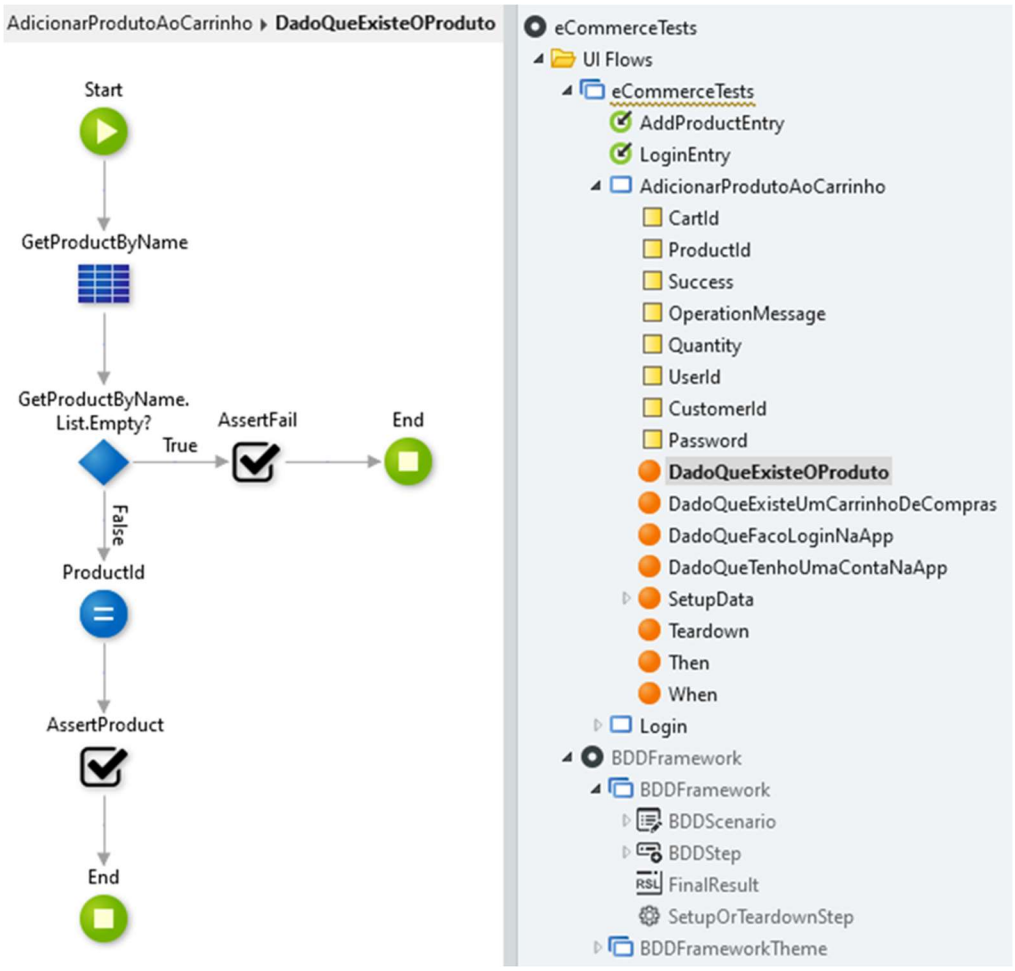

Figura 31 - Lógica do quarto passo da primeira etapa do segundo cenário.

Na segunda etapa "Quando adiciono o produto ao carrinho", Figura 32, o produto '*Astica – Chardonnay*' será adicionado ao carrinho. Para isto será utilizada a ação "*Cart_AddProduct*" que foi desenvolvida previamente para a aplicação. Nesta ação serão também guardadas informações como a quantidade, que será 1, a mensagem devolvida pela ação "*Cart_AddProduct*", e se a ação foi ou não executada com sucesso. As variáveis com esta informação serão utilizadas posteriormente em verificações.

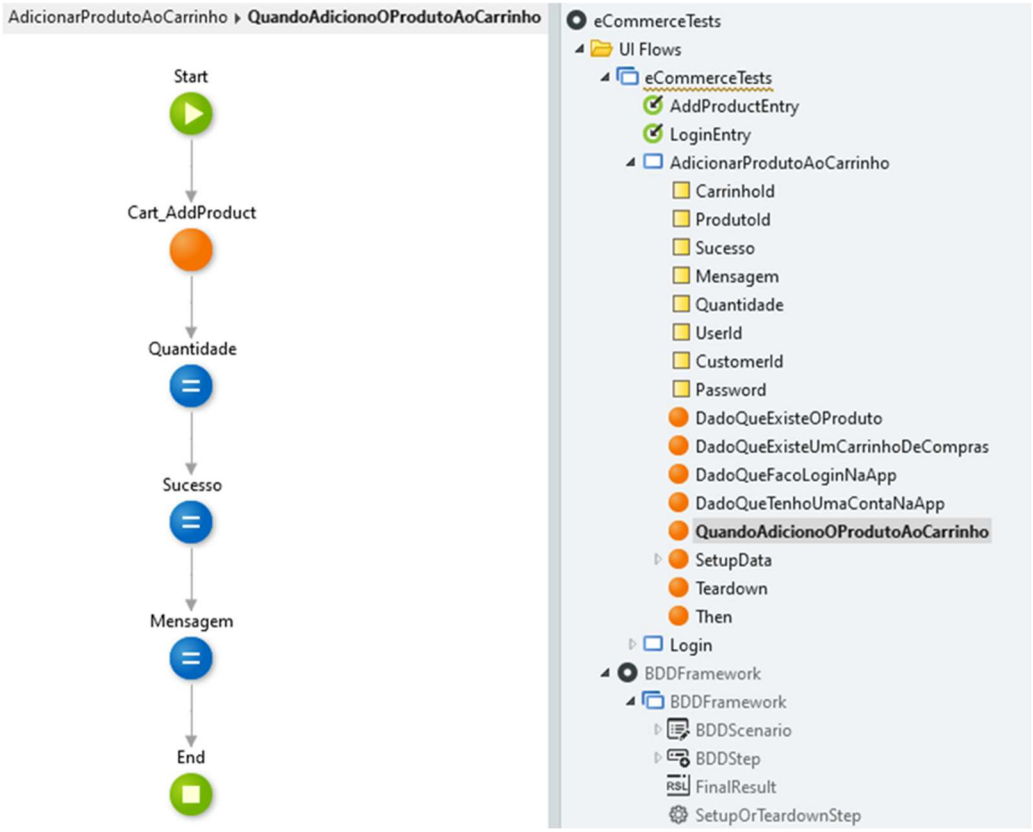

Figura 32 - Lógica do primeiro passo da segunda etapa do segundo cenário.

A última etapa deste cenário é composta por dois passos. No primeiro, "A operação deve ser bem-sucedida", serão validadas algumas das informações guardadas anteriormente. Como é possível ver na Figura 33, é verificado se a operação foi executada com sucesso através da variável *Sucesso* (verdade caso tenha sido executada com sucesso), a quantidade adicionada, que deve ser 1, e a mensagem retornada pela ação de adição do produto no carrinho, que deve ser igual à guardada na variável *Mensagem* que deverá ser igual a "1 *Astica - Chardonnay added to your shopping cart*".

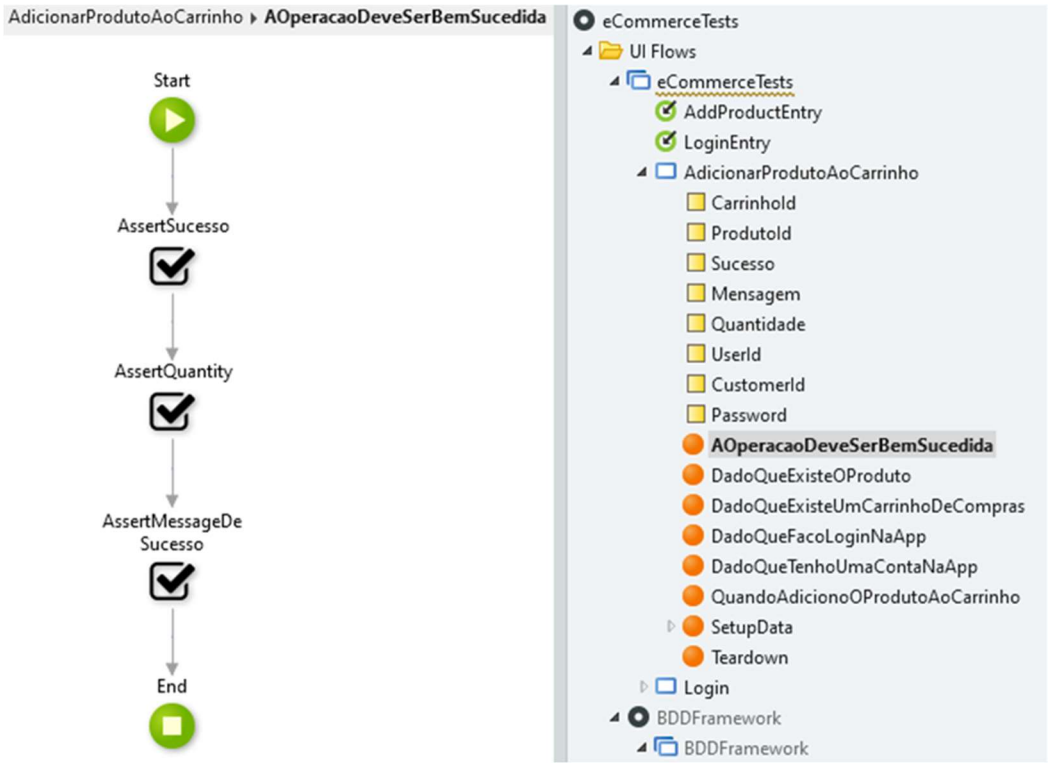

Figura 33 - Lógica do primeiro passo da terceira etapa do segundo cenário.

O último passo da última etapa, "O carrinho deve ser atualizado corretamente", consiste em verificar na base de dados se o carrinho foi atualizado corretamente após a adição do produto. Para isso, será executada uma pesquisa à base de dados que retornará todos os produtos que estão no carrinho de compras cujo identificador é igual ao que está guardado na variável *CarrinhoId*, que foi preenchida anteriormente. Após esta pesquisa serão feitas três asserções. Na primeira é validado se apenas existe um produto no carrinho de compras. Na segunda, valida-se se o nome do produto que existe é igual a "*Astica – Chardonnay*". Finalmente, na terceira é validado se a quantidade do produto no carrinho é igual a 1. Toda esta lógica pode ser visualizada na Figura 34.

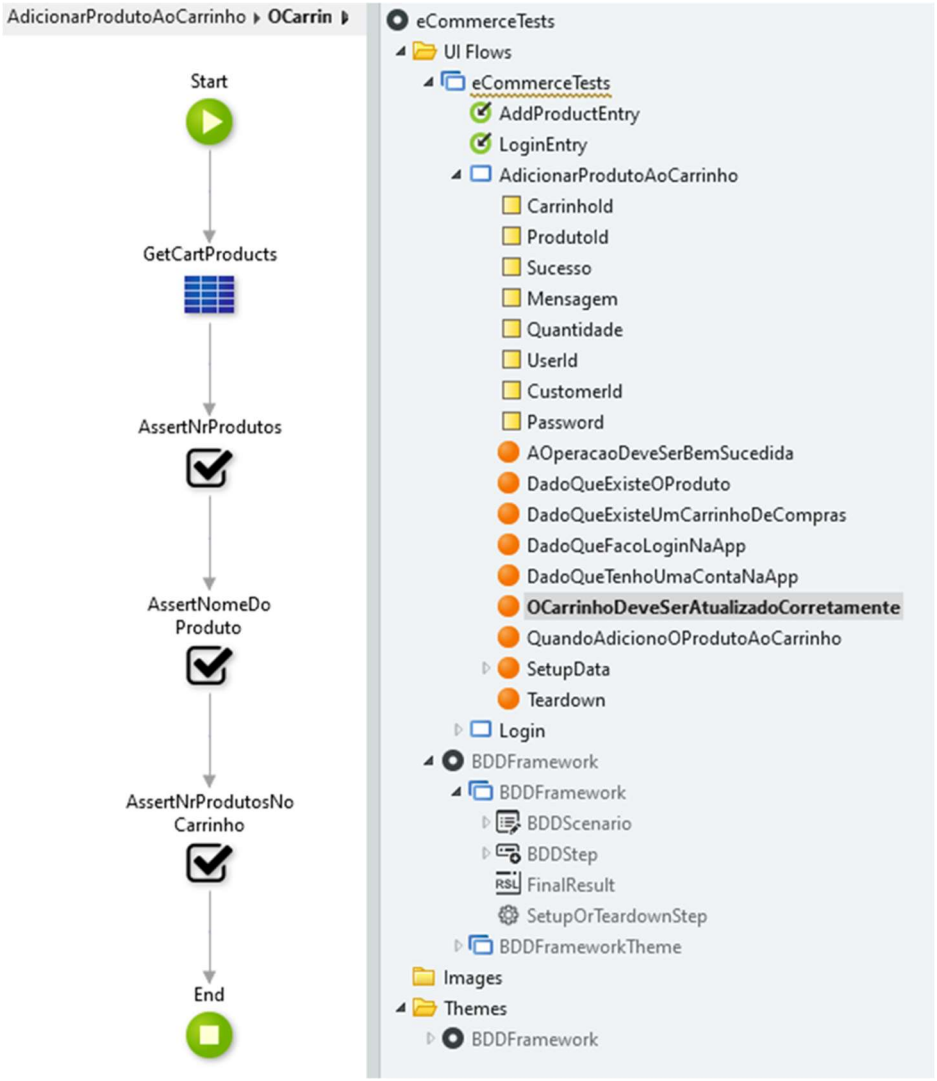

Figura 34 - Lógica do segundo passo da terceira etapa do segundo cenário.

Também, à semelhança do que aconteceu no primeiro cenário, existe a ação de *teardown* que é usada para eliminar os dados que foram criados para a execução do teste. Neste caso, será feito o *logout* do utilizador criado, e este será também eliminado assim como o *customer* associado. Por fim, o produto adicionado anteriormente será removido do mesmo e o carrinho será apagado (Figura 35).

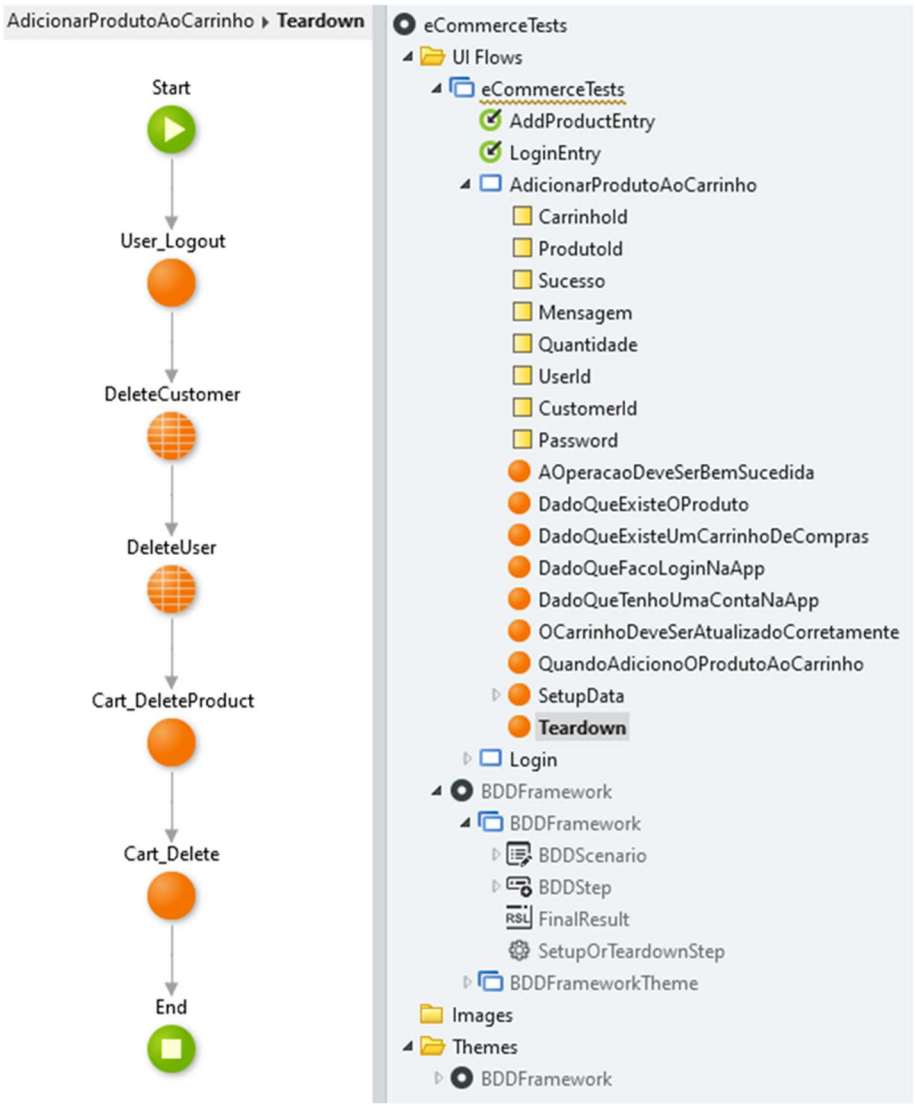

Figura 35 - Lógica de *teardown* do segundo cenário.

3º CENÁRIO

Este cenário é um pouco diferente dos dois demonstrados anteriormente pois, representa um caso de teste a uma API.

A API que irá ser utilizada para o teste é a *restcountries.eu*. É uma API REST pública que tem como objetivo retornar dados sobre um dado país.

Para o propósito deste exemplo, será utilizado um cenário de teste onde são solicitados dados sobre um país específico, neste caso Portugal, e é validado se a capital do país está correta, ou seja, se a capital retornada é Lisboa.

Para isso, será necessário fazer uma chamada GET para o seguinte *endpoint*:

```
https://restcountries.eu/rest/v2/name/{name}
```

De modo a utilizar o país “Portugal”, como exemplo, a chamada tem de ser executada da seguinte forma:

```
https://restcountries.eu/rest/v2/name/{Portugal}
```

Após a execução da chamada GET, será retornada uma resposta JSON como a seguinte:

```
[
   {
      "name":"Portugal",
      "topLevelDomain":[...],
      "alpha2Code":"PT",
      "alpha3Code":"PRT",
      "callingCodes":[...],
      "capital":"Lisbon",
      "altSpellings":[...],
      "region":"Europe",
      "subregion":"Southern Europe",
      "population":10374822,
      ...
   }
]
```

Como é possível observar, na resposta JSON um dos campos é a capital do país.

A implementação deste cenário de teste com a *BDDFramework* está representada na Figura 36.

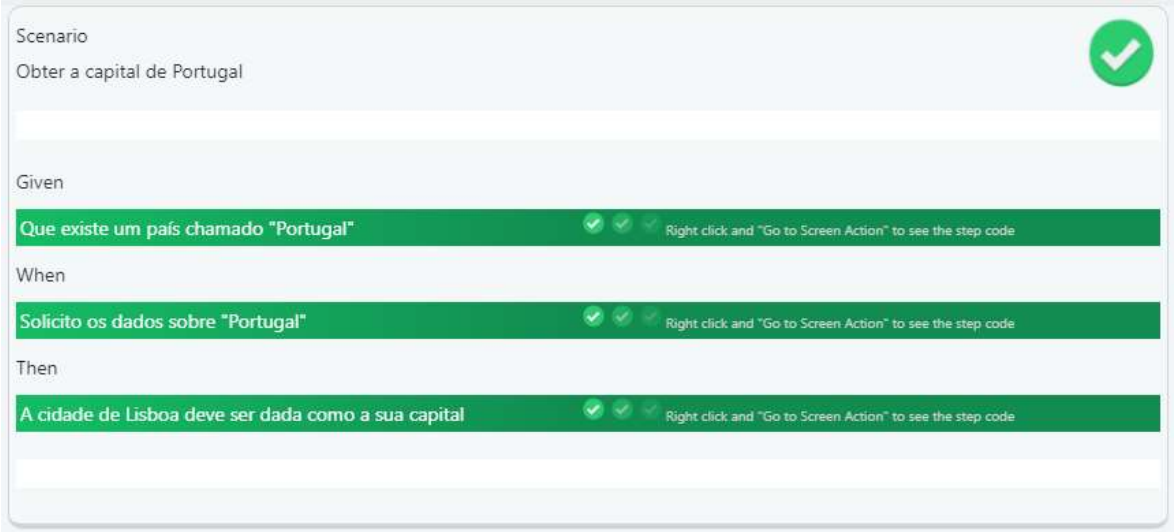

Figura 36 - Estrutura do ecrã de teste do terceiro cenário no OutSystems *Service Studio*.

Given

Para esta etapa poderíamos utilizar outra API que dado o nome do país, neste caso Portugal, validasse a existência do mesmo, contudo admitimos que existe e não é necessário nenhum desenvolvimento associado na ação desta etapa.

When

A etapa "Quando solicito os dados sobre "Portugal", Figura 37, irá estar associada a uma ação "*GetCapitalName*". Esta ação irá conter outra ação desenvolvida num módulo específico para as chamadas a APIs que irá possuir ações públicas que chamarão os vários métodos da API, de modo a serem seguidas as boas práticas mostradas anteriormente. Caso a abordagem não fosse esta, poderia ser necessário fazer o teste às várias funções que fizessem chamadas à API.

A ação "*GetCapitalName*" recebe o nome do país que o utilizador pretende procurar, neste caso "Portugal".

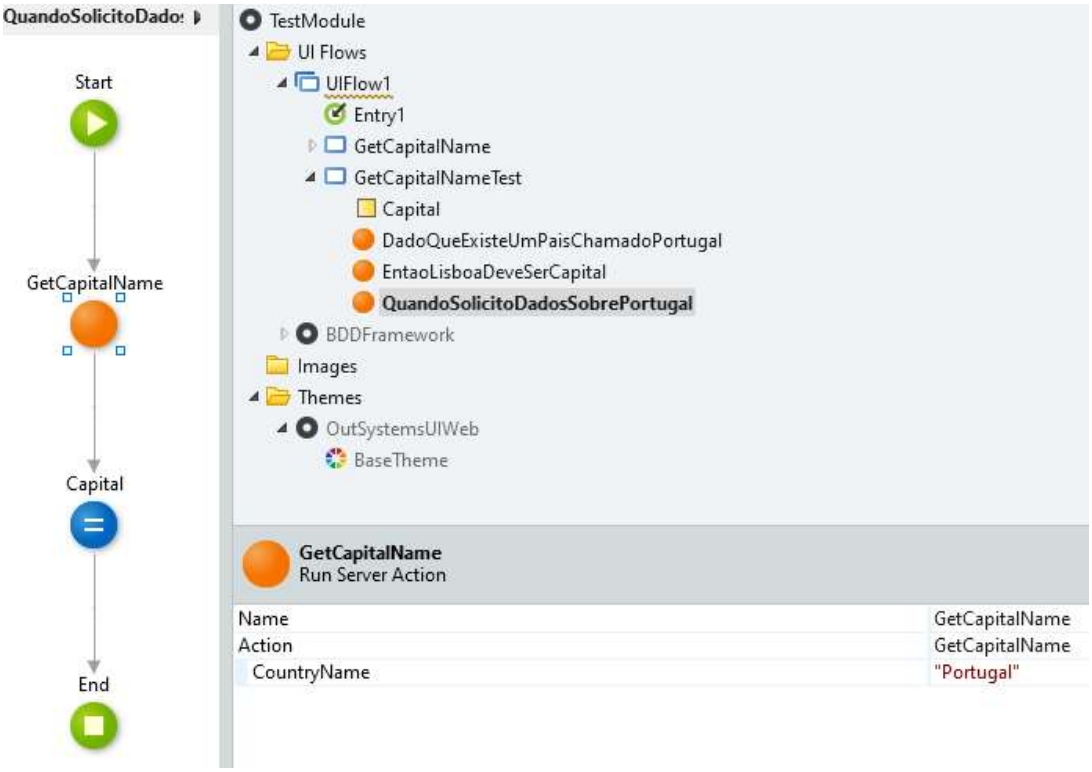

Figura 37 - Lógica da segunda etapa do terceiro cenário.

Then

O nome da capital devolvida pela chamada à API é guardado numa variável, que está declarada dentro do ecrã, que será utilizada na etapa final "Então a cidade de Lisboa deve ser dada como a sua capital" para validar o campo capital como é possível observar na Figura 38.

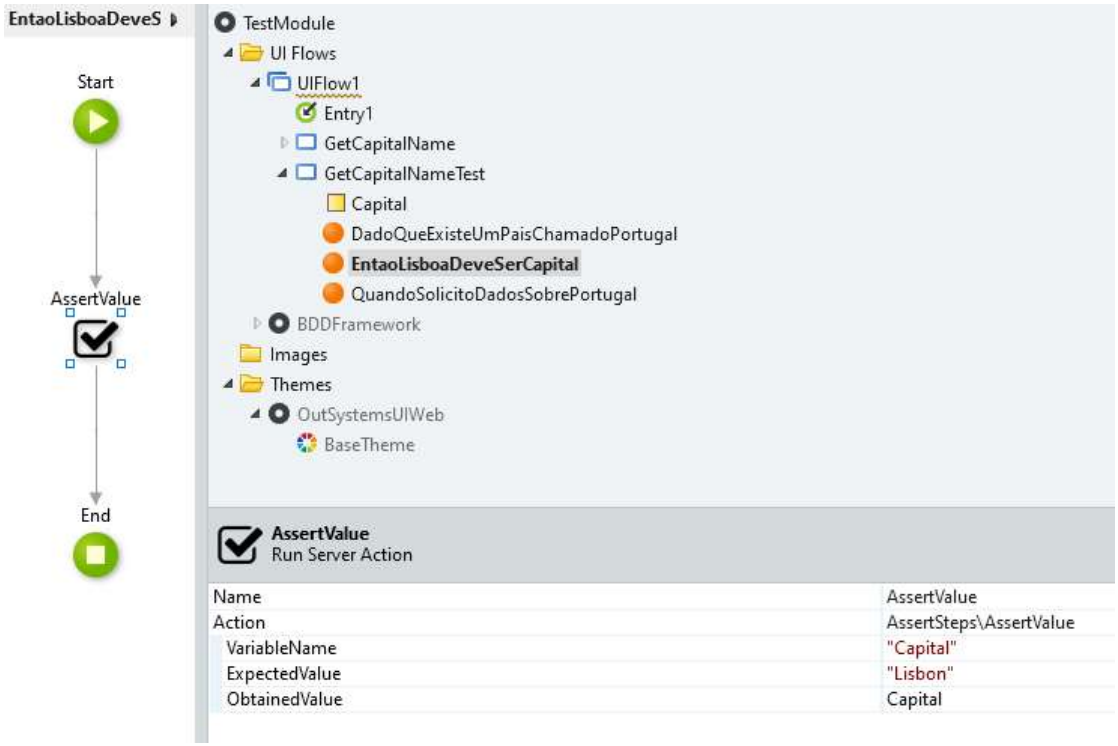

Figura 38 - Lógica da última etapa do terceiro cenário.

Observações finais

No primeiro e segundo cenário implementado com a *BDDFramework* a boa prática aplicada é o facto da lógica estar a ser reutilizada uma vez que as ações utilizadas para o teste são as mesmas que foram desenvolvidas para as funcionalidades em si, na aplicação. Aplicar esta boa prática permite aos *developers* que não tenham de modificar a lógica implementada nos testes pois, ao modificarem as ações correspondentes às funcionalidades estão também a atualizar o teste. A não utilização desta boa prática poderia refletir-se nos resultados dos testes porque o *developer* poderia, por exemplo, esquecer-se de atualizar a lógica implementada nos testes. Outra vantagem da utilização desta boa prática é que o *developer* não tem de perder mais tempo a adicionar a nova lógica no teste.

Já no terceiro cenário a boa prática mencionada anteriormente também se aplica ao facto da API utilizada ser isolada num módulo diferente para que os *developers* não tenham que implementar a lógica associada à API sempre que a necessitam. O módulo da API já contém a chamada a API bem como o método correspondente, contém também uma ação pública que contém o método da API, desta forma o *developer* apenas tem de fazer alterações no módulo da API e não em todos os lugares onde a API é utilizada.

5.3.2. Usando o *Ghost Inspector*

Como foi explicado na secção 3.2, para ser possível utilizar esta ferramenta é necessário ter instalada no *browser* a extensão do *Ghost Inspector* que permite a gravação dos testes. Após a instalação da extensão, será adicionado um ícone próprio à barra de marcadores do *browser* (Figura 39) que, quando clicado, começa a gravação do teste. Após iniciar a gravação o *tester* pode executar validações de vários tipos tal como foi descrito na secção 3.2.

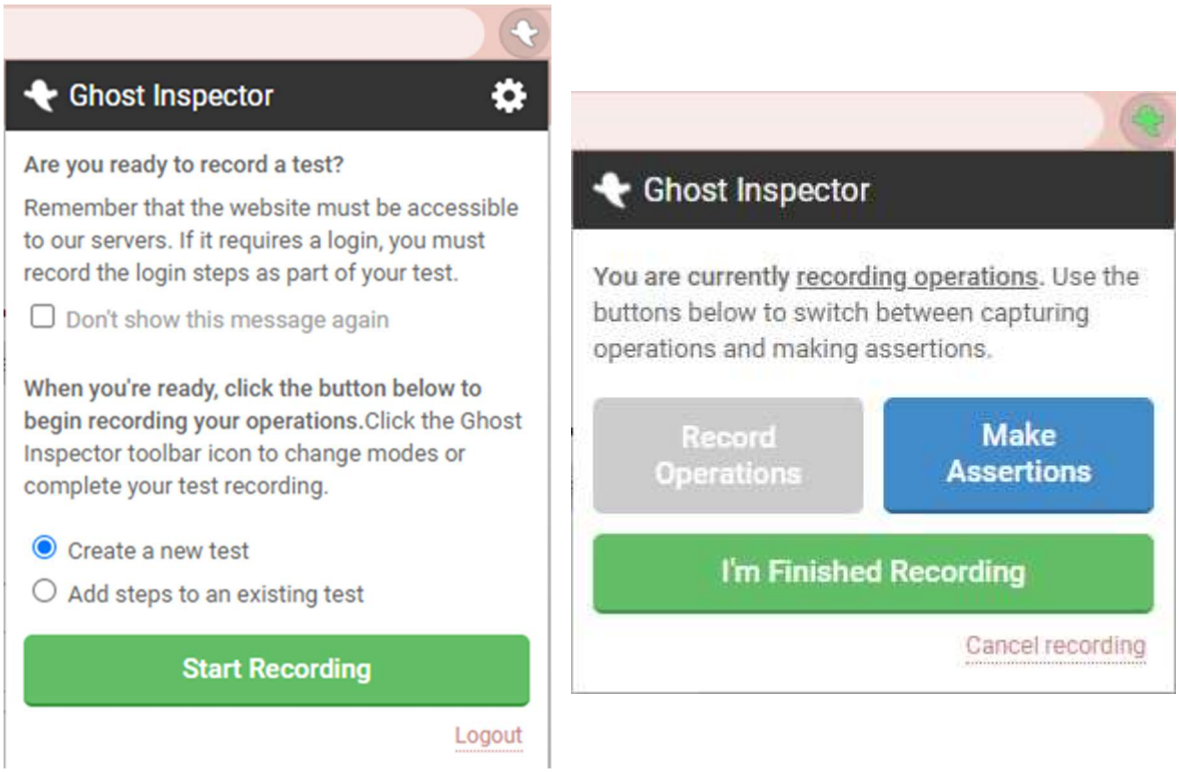

Figura 39 - Extensão do *Ghost Inspector* para o browser e após iniciar a gravação.

1º CENÁRIO

Para o primeiro cenário o *tester* apenas tem de executar o teste na aplicação que, neste caso é fazer *login* na aplicação. Contudo, para realizar as validações do teste é necessário fazer mais algumas ações durante a execução do mesmo.

Ao contrário da *BDDFramework*, o *Ghost Inspector* não permite fazer *setup* dos dados necessários para executar o teste, ou seja, o *tester* já terá de conhecer as credenciais de um utilizador existente na aplicação para conseguir iniciar o teste. Deste modo, o primeiro passo da primeira etapa do teste que é "Dado que tenho conta no '*The Wine Club*'" não será testado. Esta validação só se vai verificar caso o *login* seja bem-sucedido, visto que isso significa que o utilizador está registado na aplicação.

Figura 40 - Resultado do primeiro cenário (todos os passos executados com sucesso)

Passando para a segunda etapa do teste ("Quando"), o primeiro passo diz respeito à frase "Quando eu clico em *login*". O *tester* deve primeiro fazer a validação de que na página existe a palavra "*Login*". Para isso, terá de clicar no botão "*Make Assertion*" e selecionar a palavra *login* e, deste modo, esta validação será gravada no teste. De seguida, o *tester* irá clicar em *login* e será redirecionado para a página de *Login* onde terá de verificar se os *inputs* para inserir as credenciais estão disponíveis na página e preenchê-los, uma vez que, o segundo passo desta etapa é "E preencho o nome de utilizador e palavra-chave". Para o último passo da segunda etapa "E clico em *Sign In*" o *tester* deve seguir o mesmo processo, verificar se o botão existe na página e clicar no mesmo.

Como é possível verificar na Figura 40, o *script* do *Ghost Inspector* é escrito com base nos identificadores dos elementos presentes na página. Caso não tenham sido usadas as boas práticas descritas no capítulo anterior, podem acontecer vários erros em testes seguintes visto que os identificadores podem alterar ou poderão existir seletores iguais para elementos diferentes o que pode levar a que o elemento encontrado não seja o correto. Para evitar esta situação, devem ser dados nomes aos elementos no *Service Studio* de modo a conseguir escrever os seletores de *javascript* de forma mais especifica e correta.

Na Figura 41 podem ser visualizadas as diferenças nos seletores de *javascript*, após serem atribuídos nomes aos elementos com o que os *testers* e utilizadores da aplicação interagem. Apesar de ainda serem muito extensos, já terminam com os nomes dados aos elementos na plataforma *OutSystems*, como por exemplo "*Input_Email*", "*Input_Password*", etc.

Figura 41 - Diferenças nos seletores de JS após alterações no Service Studio.

Apesar de na figura anterior já poderem ser observadas algumas boas práticas, ainda podem ser feitas melhorias à performance do teste, para diminuir a probabilidade de este falhar devido ao facto de não identificar corretamente um determinado elemento. Para isso, e para que os *testers* não tenham de estar sempre a alterar o *script* do teste, devido a mudanças nos identificadores gerados automaticamente, estes podem ser editados e melhorados. Neste caso, o seletor pode ser escrito de forma mais especifica, dizendo para fazer as pesquisas dos *links*, *inputs* e botões pelo final do identificador de cada elemento uma vez que esta última parte do identificador nunca vai mudar a menos que seja alterado pelo *developer* no *Service Studio*. Na Figura 42, é possível ver o novo *script* após ter sido editado e melhorado.

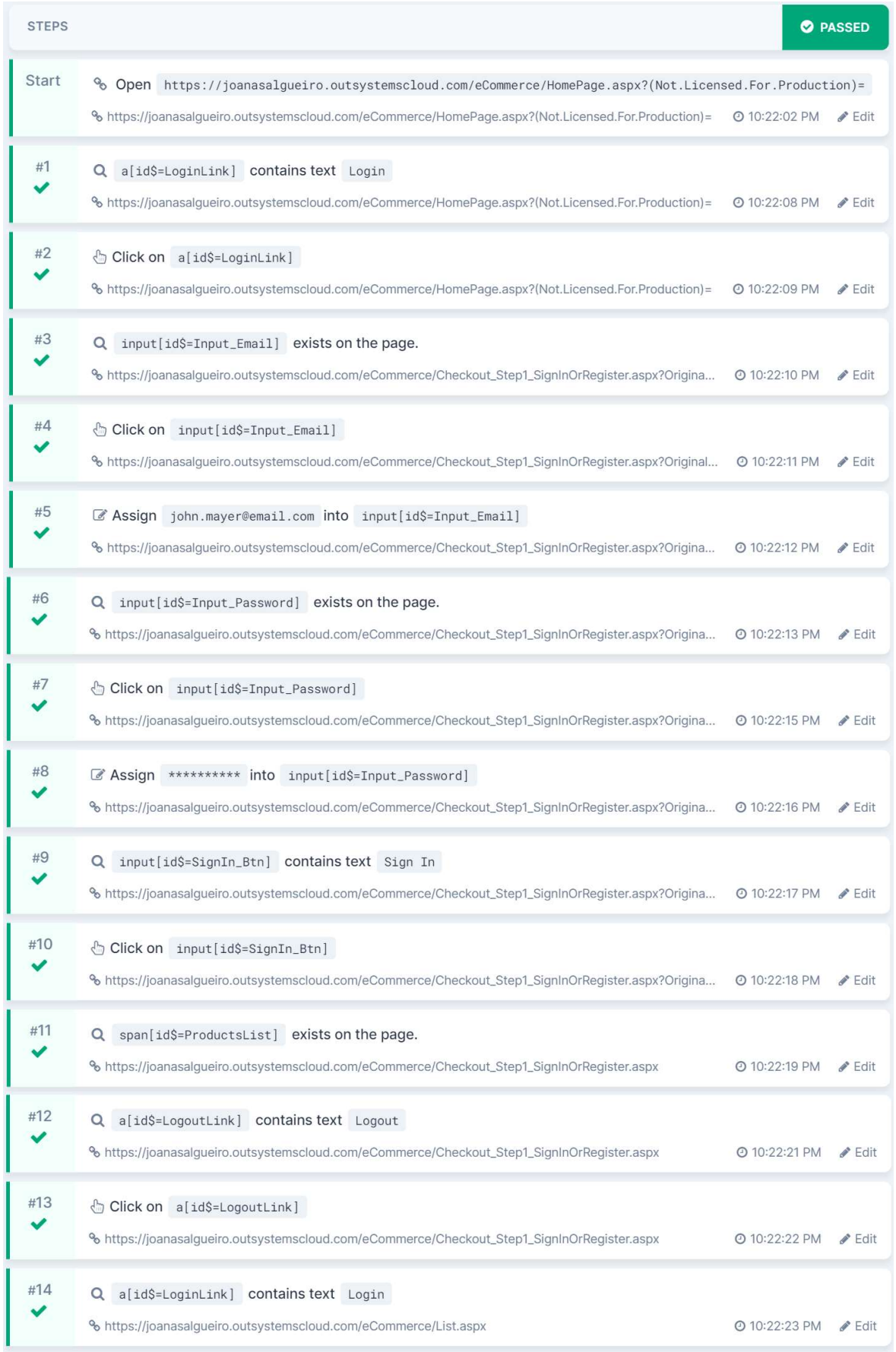

Figura 42 - Mudança dos seletores de *JavaScript* pelo *tester*.

2º CENÁRIO

Para este cenário, e como foi explicado anteriormente, o *tester* tem de reproduzir todos os passos necessários para executar o teste no *browser* de modo que seja criado o *script* de teste pela ferramenta. Neste cenário, o teste consiste em um utilizador iniciar sessão com as suas credenciais na aplicação, adicionar um produto ao carrinho e verificar que o carrinho foi atualizado corretamente.

Tal como no cenário anterior, será necessário existir um utilizador registado com credenciais na aplicação para conseguir iniciar o teste. Assim, não será validado o primeiro passo da primeira etapa do teste: "Dado que tenho conta no '*The Wine Club*'". Esta validação só se vai verificar caso o login seja bem-sucedido, visto que isso significa que o utilizador está registado na aplicação.

Para o segundo passo "E faço *login* na aplicação", e à semelhança do cenário anterior, o *tester* deve verificar que a palavra "*Login*" existe na página, clicar no *link*, preencher os campos de texto relacionados com o nome do utilizador e a palavra-chave e clicar no botão "*Sign In*".

Apesar de o utilizador possuir um carrinho, este não está visível no ecrã por ainda não terem sido adicionados produtos. Assim, como estes testes são ao nível do ecrã, e como o utilizador não tem um carrinho visível, não é possível verificar se existe um carrinho para o utilizador.

O último passo da primeira etapa deste cenário diz respeito à verificação do produto chamado "*Astica - Chardonnay"* na lista de produtos. Para isto, o *tester* deve utilizar o campo de pesquisa e pesquisar o produto pelo nome, caso a lista retorne um produto com esse nome então o passo é executado com sucesso.

Na segunda etapa deste cenário apenas existe um passo que é o de adicionar o produto ao carrinho, para este passo o *tester* apenas tem de clicar no botão "*Add to cart*", verificando a sua existência anteriormente.

Para fazer as validações da última etapa do cenário, o *tester* deve validar que o carrinho de compras possui os valores corretos, ou seja, que o nome do produto que está no carrinho é igual a "*Astica – Chardonnay*", que a quantidade do mesmo é 1 e que o preço é 12.43$.

Na Figura 43 é possível ver o resultado deste cenário executado com sucesso no *Ghost Inpector*.

Figura 43 - Resultado do segundo cenário (todos os passos executados com sucesso).

3º CENÁRIO

A API referida não tem um ecrã associado. Assim, como o *Ghost Inspector* apenas executa testes ao nível do ecrã, para executar um teste para este cenário foi necessário desenvolver ecrã em que o utilizador pode inserir o nome do país que quer saber a capital e ao clicar no botão "Obter capital" será retornado nome da capital do país. A alternativa seria fazer o teste a um cenário mais complexo que, num dos seus passos, fizesse a chamada à API. Para evitar maior complexidade no cenário de teste, optou-se pelo desenvolvimento de um ecrã que implementa uma chamada à API. Neste caso, será utilizado novamente o país "Portugal" como exemplo, como é possível ver na Figura 44.

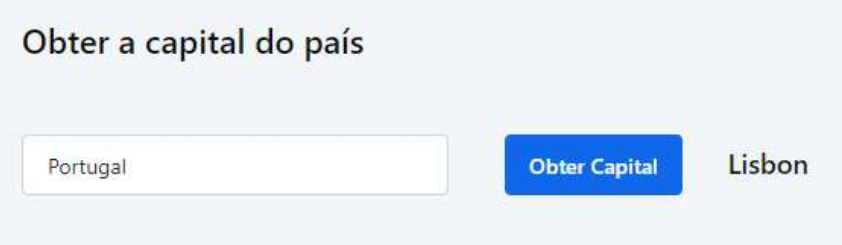

Figura 44 - Página web para executar o 3º cenário.

Figura 45 - Execução do teste correspondente ao 3º cenário.

Na Figura 45 é possível ver a execução do teste com a ferramenta *Ghost Inspector* já com as melhores práticas, visto que este ecrã já foi desenvolvido com esse propósito e seguindo as melhores práticas descritas anteriormente.

Observações Finais

Para os testes executados ao nível do UI, utilizando o *Ghost Inspector,* a boa prática utilizada tem a ver com as validações feitas ao nível do UI. O facto de o *developer* dar nomes lógicos a cada elemento presente no ecrã acrescenta um grande valor ao teste pois, caso os elementos não fossem identificados pelo nome dado e, fossem selecionados apenas pelo nome dado de forma aleatória, os testes poderiam falhar por não encontrarem os elementos no ecrã.

5.3.3. Usando a *Tricentis Tosca*

Tal como na ferramenta *Ghost Inspector*, a *Tricentis Tosca* também disponibiliza uma extensão para os *browsers* de modo a gravar a navegação do utilizador na aplicação. Com esta extensão é possível fazer verificações durante a execução do teste, como por exemplo verificar a presença de inputs, e fazer a gravação normal do teste

Através da plataforma do *Tricentis Tosca*, chamada *Tosca Commander*, os *testers* conseguem começar a testar a aplicação alvo. Para isso os utilizadores podem criar os casos de teste manualmente ou gravá-los através do browser.

Neste caso, será utilizada a gravação automática tal como foi feito na ferramenta anterior e, desta forma, o *tester* apenas tem de clicar na opção "*Automated Test Case*" que pode ser visualizada na Figura 46 que irá abrir o *browser* onde o *tester* pode iniciar o teste com a extensão previamente instalada.

Na imagem também é possível ver a secção de *Test Cases*, nesta seção irão estar os três cenários de teste com os respetivos casos de teste que já foram descritos anteriormente e que de seguida serão executados a partir desta ferramenta.

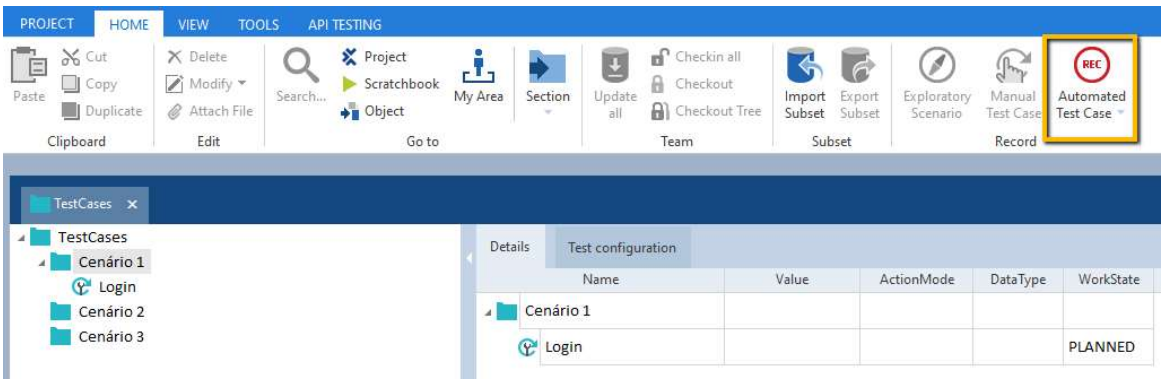

Figura 46 - Opção de gravação de teste na plataforma *Tricentis Tosca*.

A principal diferença entre esta ferramenta e as descritas anteriormente é que esta possibilita a execução de ambos os tipos de testes, ou seja, de testes de serviço e lógica de negócio e também testes de interface do utilizador. Para a execução dos testes de serviço devem ser expostas APIs que representem as ações executadas na aplicação, ou

seja, devem ser desenvolvidas novas ações que exponham APIs que contenham a lógica executada em cada teste. Quanto aos testes de interface do utilizador o processo é bastante semelhante ao executado através do *Ghost Inspector*.

1º CENÁRIO

Teste de Interface do Utilizador

Neste primeiro cenário, o teste de interface do utilizador segue praticamente o mesmo processo dos testes executados com o *Ghost Inspector*.

Após o *tester* executar o teste na aplicação e parar a gravação, será redirecionado para a plataforma onde poderá rever todos os passos executados no teste, como é possível ver na Figura 47. O utilizador acede à página inicial da aplicação *eCommerce* e verifica se existe um *link* "*Login*", se existir clica nele e é redirecionado para a página de introdução das credenciais de *login*, onde são verificadas a existência dos campos de texto para a introdução das mesmas. Após introduzir as credenciais o utilizador clica no botão "*Sign In*", verificado previamente, e de modo a verificar que o *login* foi executado com sucesso deve estar presente na página "*Welcome John Mayer*" em que o nome diz respeito ao utilizador que fez *login*.

Nesta mesma figura é também possível verificar que tal como no *Ghost Inspector* estão presentes as verificações de elementos no teste através dos seletores de *javascript* que apesar de já terminarem com o nome dado ao elemento nó código ainda poderiam ser melhorados como foi explicado anteriormente. Com esta ferramenta a edição manual dos testes também é possível, por isso, os *testers* poderiam alterar os seletores de modo a executarem um teste de modo seguro.

Login				COMPLETED
Abrir aplicação				
Url	https://joanasalgueiro.outsystemscloud.com/eCommerce/HomePage.aspx	Input	String	
UseActiveTab	True	Input	String	
Verificar a existência do link "Login"				
Login	Login	Verify	String	
Welcome, John Mayer | My Info | Logout			String	
Clicar em "Login"				
Login	X	Input	String	
Welcome, John Mayer | My Info | Logout			String	
Verificar campos de textos e botão				
wt11$RichWidgets_wt1$block$wtMainContent$wtMainContent$wt1$wtInput_Email		Verify	String	
wt11_RichWidgets_wt1_block_wtMainContent_wtMainContent_wt1_wtUserLogin		Select	String	
Sign In	Sign In	Verify	String	
wt11$RichWidgets_wt1$block$wtMainContent$wtMainContent$wt1$wtInput_Password			Password	
Introduzir credenciais de login				
wt11$RichWidgets_wt1$block$wtMainContent$wtMainContent$wt1$wtInput_Email	john.mayer@email.com	Input	String	
wt11_RichWidgets_wt1_block_wtMainContent_wtMainContent_wt1_wtUserLogin			String	
Sign In			String	
wt11$RichWidgets_wt1$block$wtMainContent$wtMainContent$wt1$wtInput_Password	*****	Input	Password	
Clicar em "Sign In"				
wt11$RichWidgets_wt1$block$wtMainContent$wtMainContent$wt1$wtInput_Email			String	
wt11_RichWidgets_wt1_block_wtMainContent_wtMainContent_wt1_wtUserLogin			String	
Sign In	X	Input	String	
wt11$RichWidgets_wt1$block$wtMainContent$wtMainContent$wt1$wtInput_Password			Password	
Verificar sucesso do login				
Login			String	
Welcome, John Mayer | My Info | Logout	InnerText == Welcome, John Mayer | My Info | Logout	Verify	String	

Figura 47 – *Script* do primeiro cenário de teste (UI).

A ferramenta também apresenta o teste na forma de diagrama de controlo de fluxo de forma a facilitar a leitura a um utilizador menos técnico ou até mesmo para o *tester*

consultar o fluxo do teste de forma rápida e simples, como é possível verificar na Figura 48.

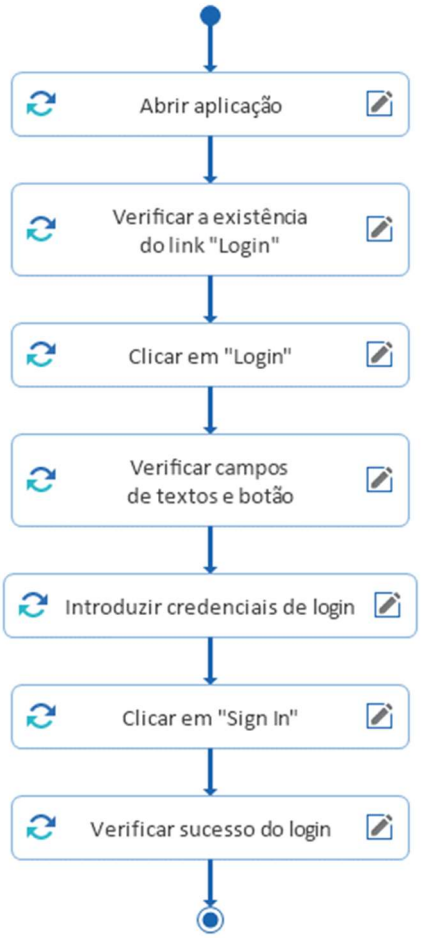

Figura 48 - Diagrama de controlo de fluxo do 1º cenário (UI).

Após o *tester* ter o *script* de teste concluído (Figura 47) pode executar o teste a qualquer momento clicando em "Run in ScratchBook". O resultado desta execução, quando executado com sucesso, será semelhante ao que pode ser visto na Figura 49.

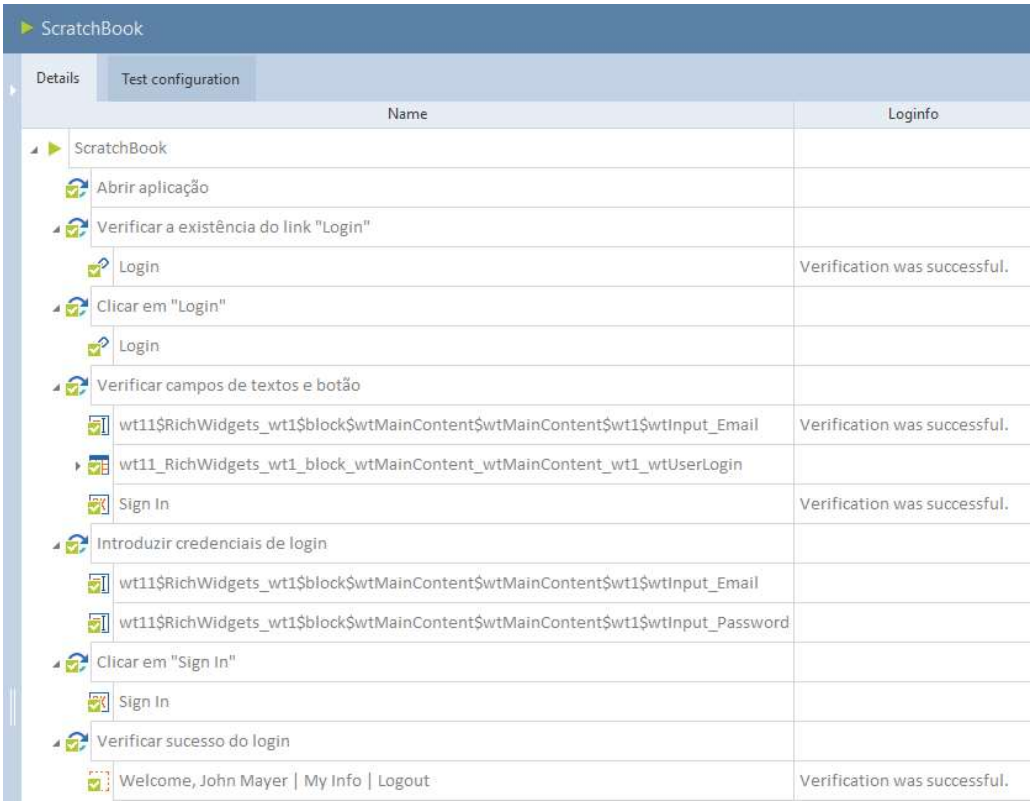

Figura 49 - Execução do 1º cenário (UI) com sucesso.

No caso de o *tester* introduzir credenciais erradas, por exemplo o email, o teste irá falhar, pois o utilizador não existe na aplicação e como consequência o utilizador não será redirecionado para a página inicial da aplicação onde pode é feita a verificação de que o *login* foi executado com sucesso. Este cenário pode ser visualizado na figura seguinte (Figura 50).

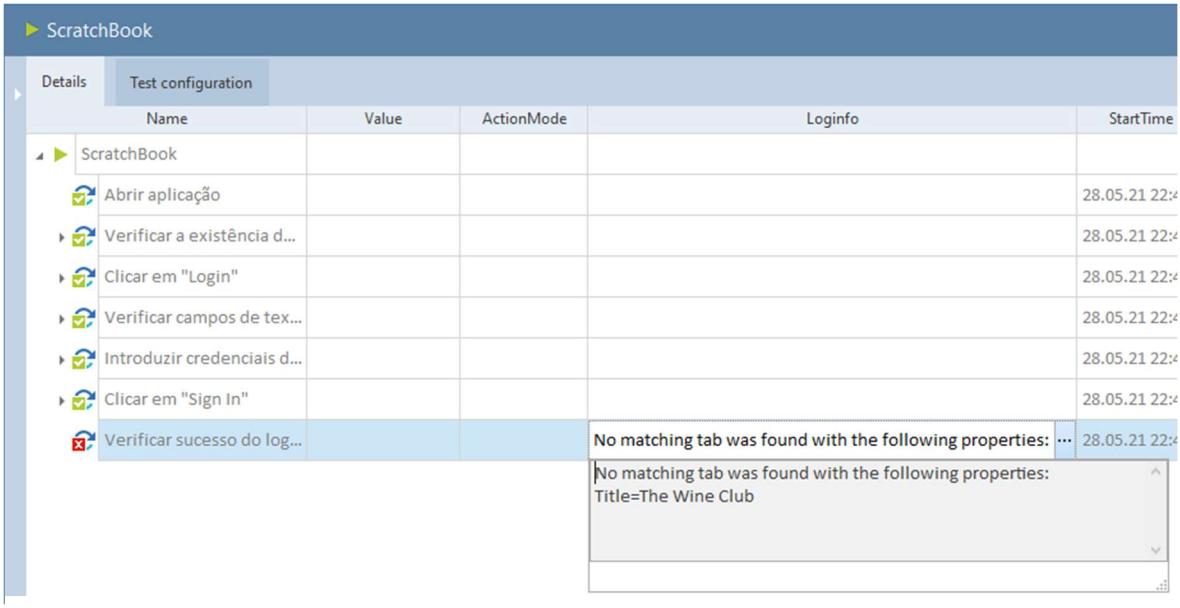

Figura 50 - Execução do 1º cenário (UI) sem sucesso.

Teste de Serviço

Como foi referido anteriormente, para conseguir executar testes da lógica de negócio da aplicação tal como acontece com a ferramenta *BDDFramework,* é necessário isolar a lógica necessária em ações que vão estar consequentemente expostas em métodos de uma API.

Para o *tester* conseguir executar um teste de serviço tem de se dirigir à secção "API Testing" e clicar na opção "API Scan" como é possível ver na Figura 51.

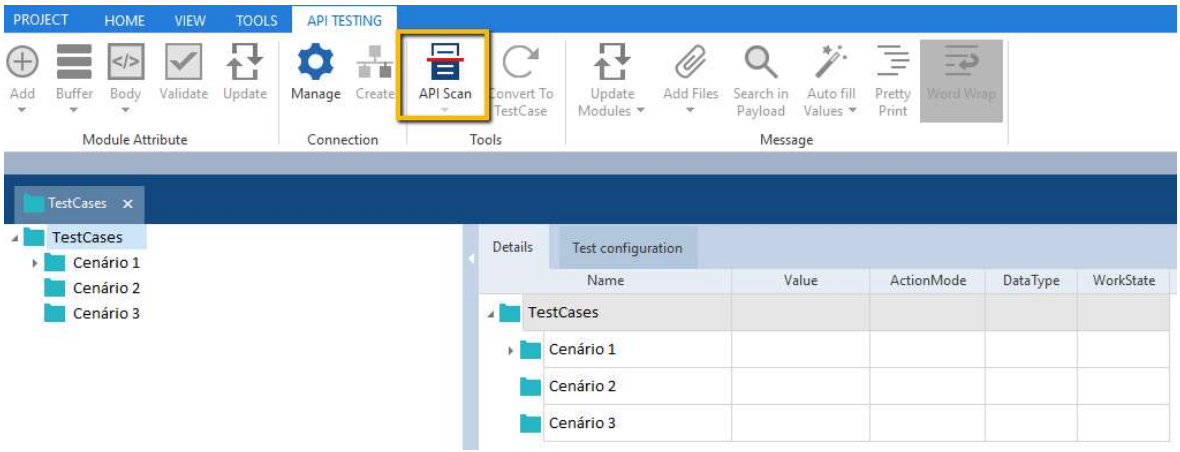

Figura 51 - *API Testing* no *Tricentis Tosca*.

No caso do primeiro cenário foi criada uma API que terá um método chamado *Login* (Figura 52), que irá receber o email do utilizador que irá fazer login para verificar a sua existência na base de dados e, caso seja encontrado será executado o *login* com as credenciais providenciadas.

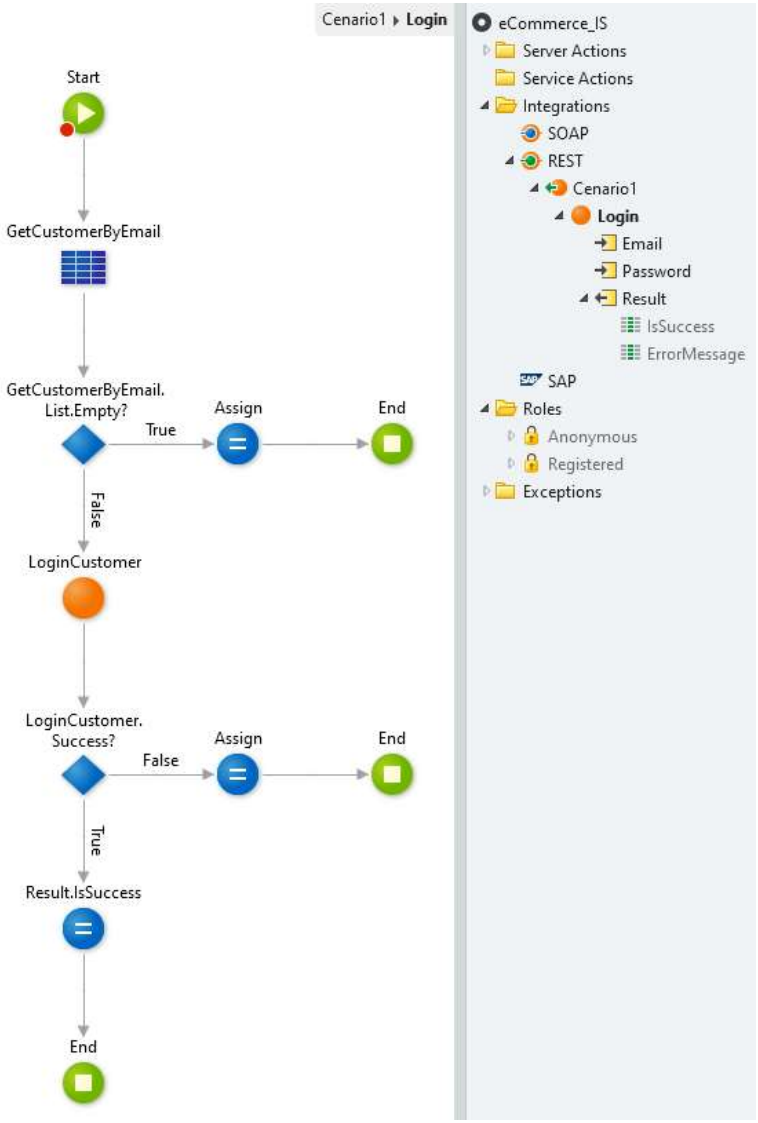

Figura 52 - Ação correspondente ao método da API do primeiro cenário.

Após já existirem as APIs necessárias para o cenário de teste, os *testers* podem então executar o teste clicando na opção "API Scan" que irá abrir a janela mostrada na Figura 53. O *tester* deve preencher as informações necessárias como o *endpoint* e a *resource* e também deverá enviar o *request* com os parâmetros necessários há execução do teste, neste caso, o email e *password* do utilizador.

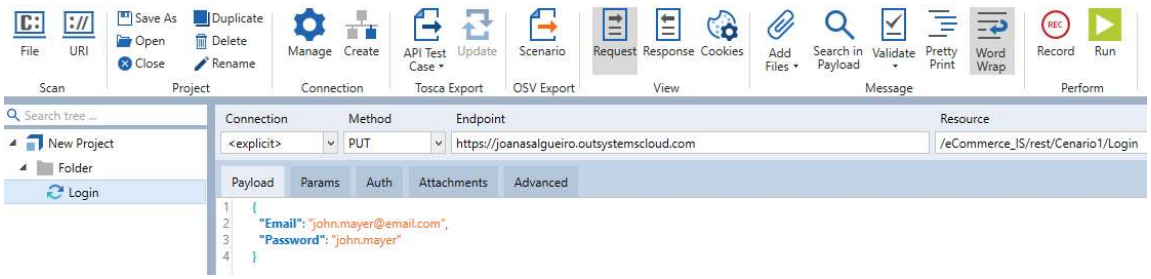

Figura 53 – Teste de API do 1º Cenário.

Após o *tester* clicar em "*Run*" será executado o teste e, caso seja executado com sucesso, o resultado será o seguinte (Figura 54):

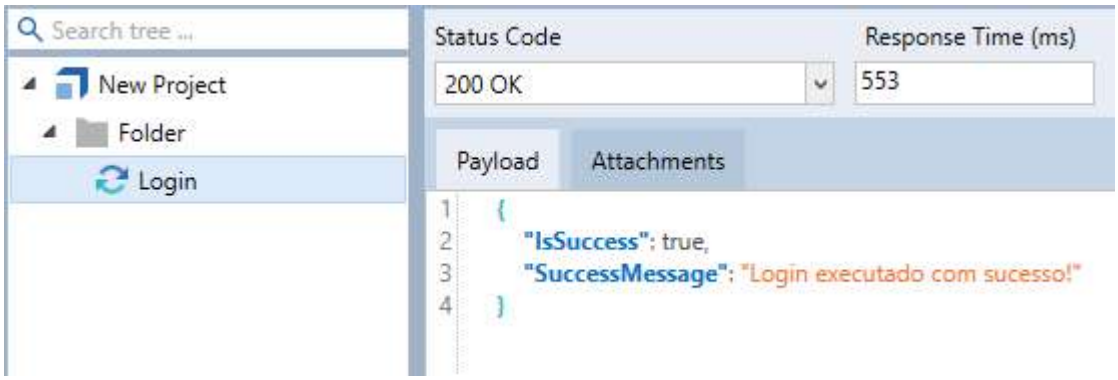

Figura 54 - Resultado da execução do 1º cenário quando executado com sucesso.

Caso o *tester* indique credenciais erradas o teste será executado sem sucesso e terá um resultado semelhante ao seguinte (Figura 55):

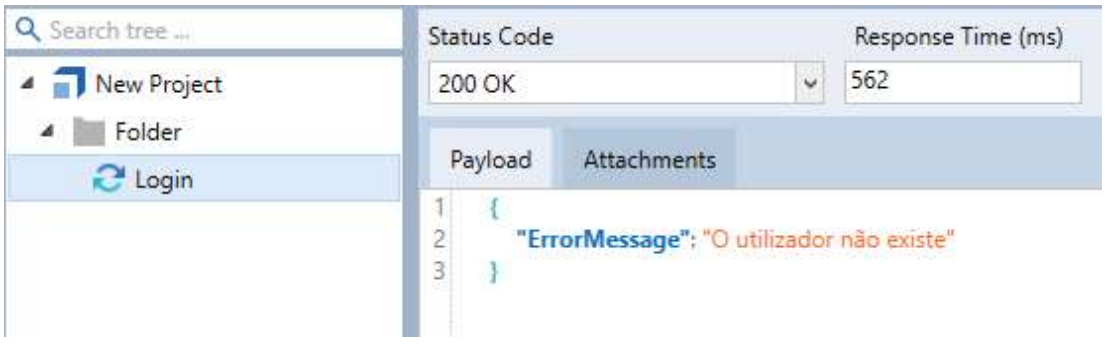

Figura 55 - Resultado da execução do 1º cenário quando executado com sem sucesso.

Após o *tester* concluir o *scan* da API deve então construir um caso de teste tal como mostrado nos testes de UI. Para isso o *tester* apenas tem de exportar o teste como é possível ver na Figura 56.

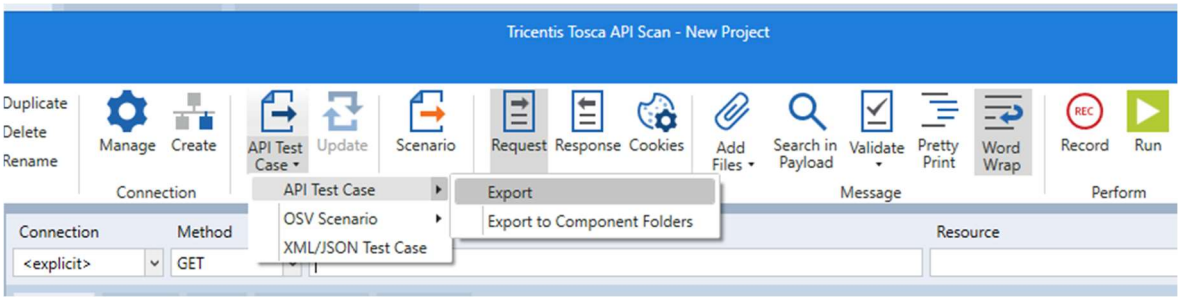

Figura 56 - Exportar teste de API para um caso de teste.

Após ter criado o teste, o *tester* tem de adicionar os passos que pretende executar/verificar durante a execução do teste. A plataforma não o faz automaticamente porque o utilizador pode querer customizar os passos executados no teste.

De modo a adicionar os passos e/ou verificações o *tester* tem de clicar em "*Add*" e selecionar os passos que pretende, como é possível ver na Figura 57.

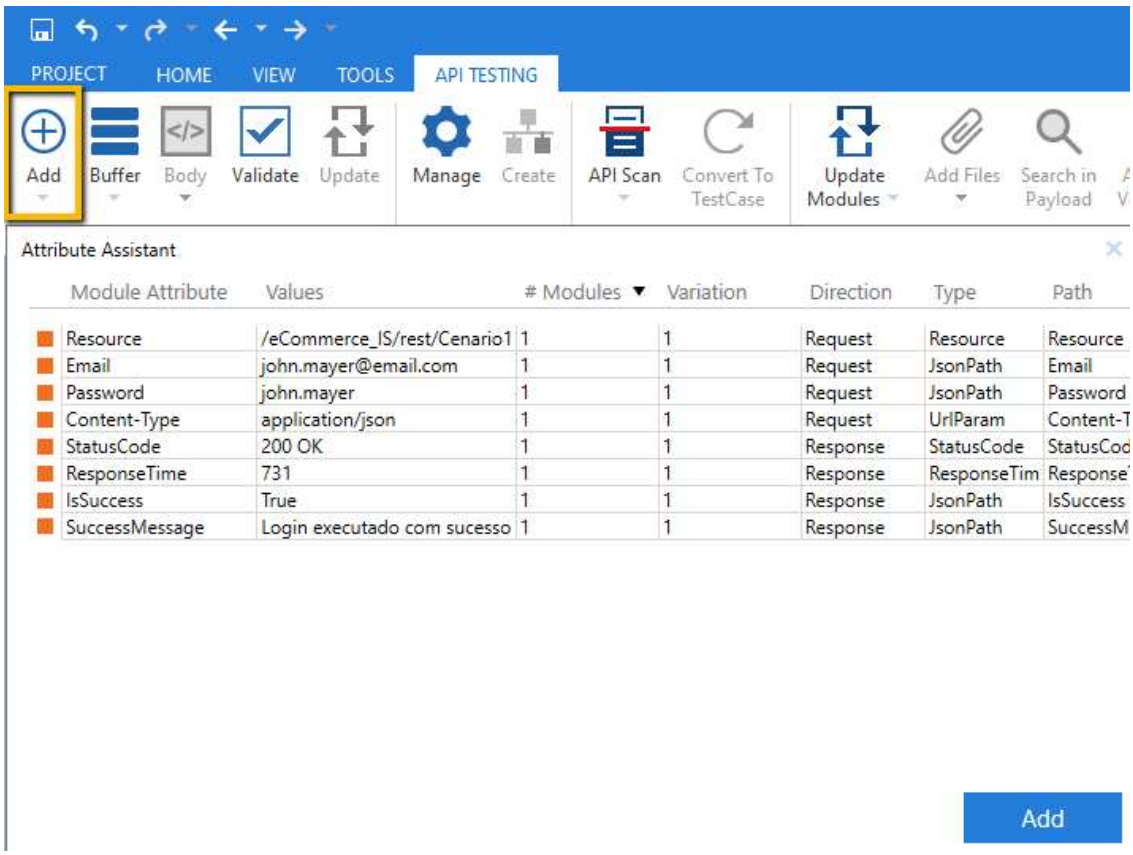

Figura 57 - Adicionar passos/verificações ao caso de teste

Após serem adicionados os passos pretendidos, neste caso todos, o *script* do caso de teste será semelhante à Figura 58. Resumidamente, a figura mostra que o *tester* introduz os campos "*resource*", "email", "*password*" e "*content-type*", que são os parâmetros necessários à execução do teste e, após ser executado o *login* são feitas

verificações às variáveis retornadas pela API que são o "*status code*", "*IsSuccess*" e a "*SuccessMessage*".

Name	Value	ActionMode	DataType	WorkState
TestCases				
Login				COMPLETED
Login Request				
Resource	/eCommerce_IS/rest /Cenario1/Login	Insert	String	
Email	john.mayer@email.com	Insert	String	
Password	john.mayer	Insert	String	
Content-Type	application/json	Insert	String	
Login Response				
StatusCode	200 OK	Verify	String	
ResponseTime	<= 731	Verify	Numeric	
IsSuccess	True	Verify	Boolean	
SuccessMessage	Login executado com sucesso!	Verify	String	

Figura 58 – *Script* do caso de teste 1º cenário (API).

Agora que o caso de teste já está finalizado o *tester* pode executá-lo e ver se este é ou não executado com sucesso. Para executar o teste, o *tester* tem que selecionar o caso de teste e clicar em "*Run in Scratchbook*". Para que o teste seja executado com sucesso as variáveis devem retornar valores específicos, o *StatusCode* deve ser igual a 200, o *IsSuccess* deve ser igual a *true* e a *SuccessMessage* deve ser igual a "*Login* executado com sucesso". O resultado pode ser visualizado na Figura 59.

ScratchBook

Details | Test configuration

Name	Value	ActionMode	Loginfo	StartTime	Duration	Detail
ScratchBook					00:00.977	
Login Request				28.05.21 21:3...	00:00.661	
Resource			Ok	28.05.21 21:3...	00:00.106	
Email			Ok	28.05.21 21:3...	00:00.110	
Password			Ok	28.05.21 21:3...	00:00.110	
Content-Type			Ok	28.05.21 21:3...	00:00.110	
Login Response			Server Response Time: 683 ms	28.05.21 21:3...	00:00.316	
StatusCode			Verification was successful.	28.05.21 21:3...	00:00.294	
ResponseTime			Verification was successful.	28.05.21 21:3...	00:00.294	
IsSuccess			Verification was successful.	28.05.21 21:3...	00:00.294	

Figura 59 - Resultado da execução do 1º cenário (API) com sucesso.

Tal como verificado no mesmo teste, mas de UI, o teste também pode falhar, por exemplo, se o *tester* introduzir um email incorreto, uma vez que a variável *IsSuccess* irá retornar *false* em vez de *true* bem como a mensagem de erro será diferente. Este cenário pode ser visto na Figura 60.

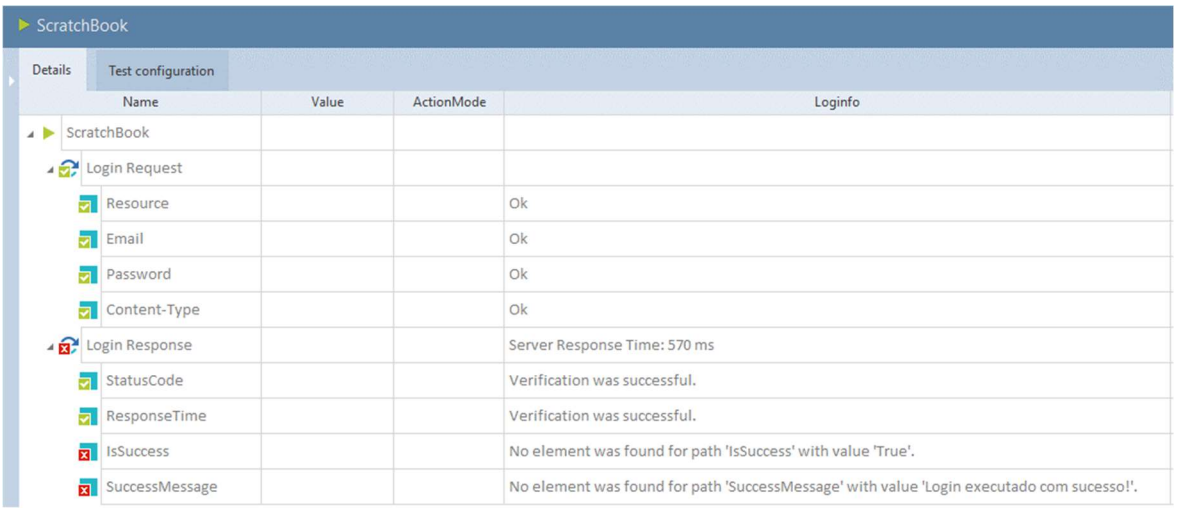

Figura 60 - Resultado da execução do 1º cenário (API) sem sucesso.

2º CENÁRIO

Teste de Interface do Utilizador

Para este cenário, e como foi explicado anteriormente, o *tester* tem de reproduzir todos os passos necessários para executar o teste no *browser* de modo que seja criado o *script* de teste pela ferramenta. Neste cenário, o teste consiste em um utilizador iniciar sessão com as suas credenciais na aplicação, adicionar um produto ao carrinho e verificar que o carrinho foi atualizado corretamente.

Como já existe um cenário correspondente ao *login* na aplicação este mesmo pode ser reutilizado como um passo deste segundo cenário, o que faz com que o *tester* não tenha que realizar essa parte do teste e tenha apenas de utilizar o teste do cenário anterior. De modo a criar casos de teste reutilizáveis na *Tricentis*, o *tester* tem de criar uma biblioteca que irá conter todos os testes reutilizáveis, após criar a biblioteca apenas tem de copiar o teste que pretende, neste caso o teste de *login* correspondente ao cenário anterior (Figura 61). Após estar criado apenas tem de arrastar o teste reutilizável para o caso de teste pretendido e quando o executar será executado também o teste reutilizável, ou seja, o *login*.

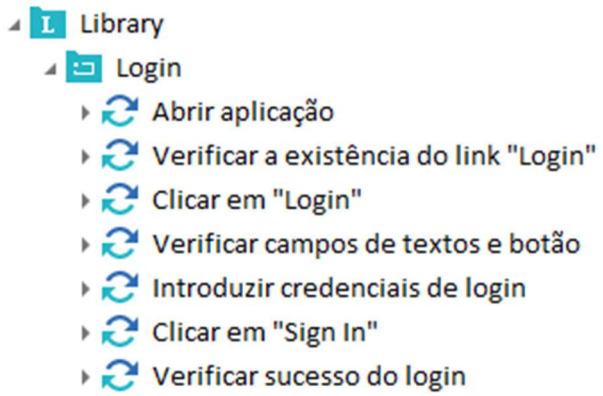

Figura 61 - Biblioteca de testes reutilizáveis.

Após o login na aplicação o utilizador deve pesquisar pelo produto "*Astica – Chardonnay*", esta pesquisa irá retornar o produto mencionado e por isso o *tester* irá verifica se o produto existe na lista retornada. Caso exista o produto e também o botão "*Add to Cart*" o utilizador irá clicar no botão e irá ser redirecionado para a página do carrinho de compras. Nesta página existem algumas verificações que devem ser feitas, tais como a quantidade adicionada que deve ser igual a 1, o produto adicionado que deve ter o nome "*Astica – Chardonnay*" e o preço do mesmo que deve ser de 12.43$.

O caso de teste correspondente a este cenário de UI será igual ao apresentado na Figura 62.

Name	Value	ActionMode	DataType	WorkState
AdicionarProdutoCarrinho				COMPLETED
Login_Reference				
Abrir aplicação				
Verificar a existência do link "Login"				
Clicar em "Login"				
Verificar campos de textos e botão				
Introduzir credenciais de login				
Clicar em "Sign In"				
Verificar sucesso do login				
Abrir aplicação				
Url	https://joanasalgueiro.outsystemscloud.com/eCommerce/HomePage.aspx?(Not.Licensed.For.Production)=	Input	String	
UseActiveTab	True	Input	String	
Verificar produto				
wt1$RichWidgets_wt1$block$wtHeader$wtHeader$wtInput_Search	Astica - Chardonnay	Input	String	
Search	X	Input	String	
Astica - Chardonnay	Astica - Chardonnay	Verify	String	
Add to cart	Add to cart	Verify	String	
Adicionar produto ao carrinho				
wt1$RichWidgets_wt1$block$wtHeader$wtHeader$wtInput_Search			String	
Search			String	
Astica - Chardonnay			String	
Add to cart	X	Input	String	
Verificar carrinho de compras				
wt14_RichWidgets_wt1_block_wtMainContent_wtMainContent_wt12_wtCartLineTable		Select	String	
$1		Select	String	
Product	Product Astica - Chardonnay 2009	Verify	String	
<Cell>			String	
<Row>			String	
<Col>			String	
$12.43	InnerText == $12.43	Verify	String	
23	1	Verify	String	

Figura 62 – *Script* do caso de teste 2º cenário (UI).

Na Figura 63, é possível ver o diagrama de controlo de fluxo do segundo cenário. Este contém todos os passos do caso de teste em forma de esquece para facilitar a sua leitura.

Figura 63 - Diagrama de controlo de fluxo do 2º cenário.

Com o *script* de teste pronto a executar o *tester* pode fazê-lo e caso seja executado com sucesso o resultado pode ser visualizado na figura seguinte (Figura 64).

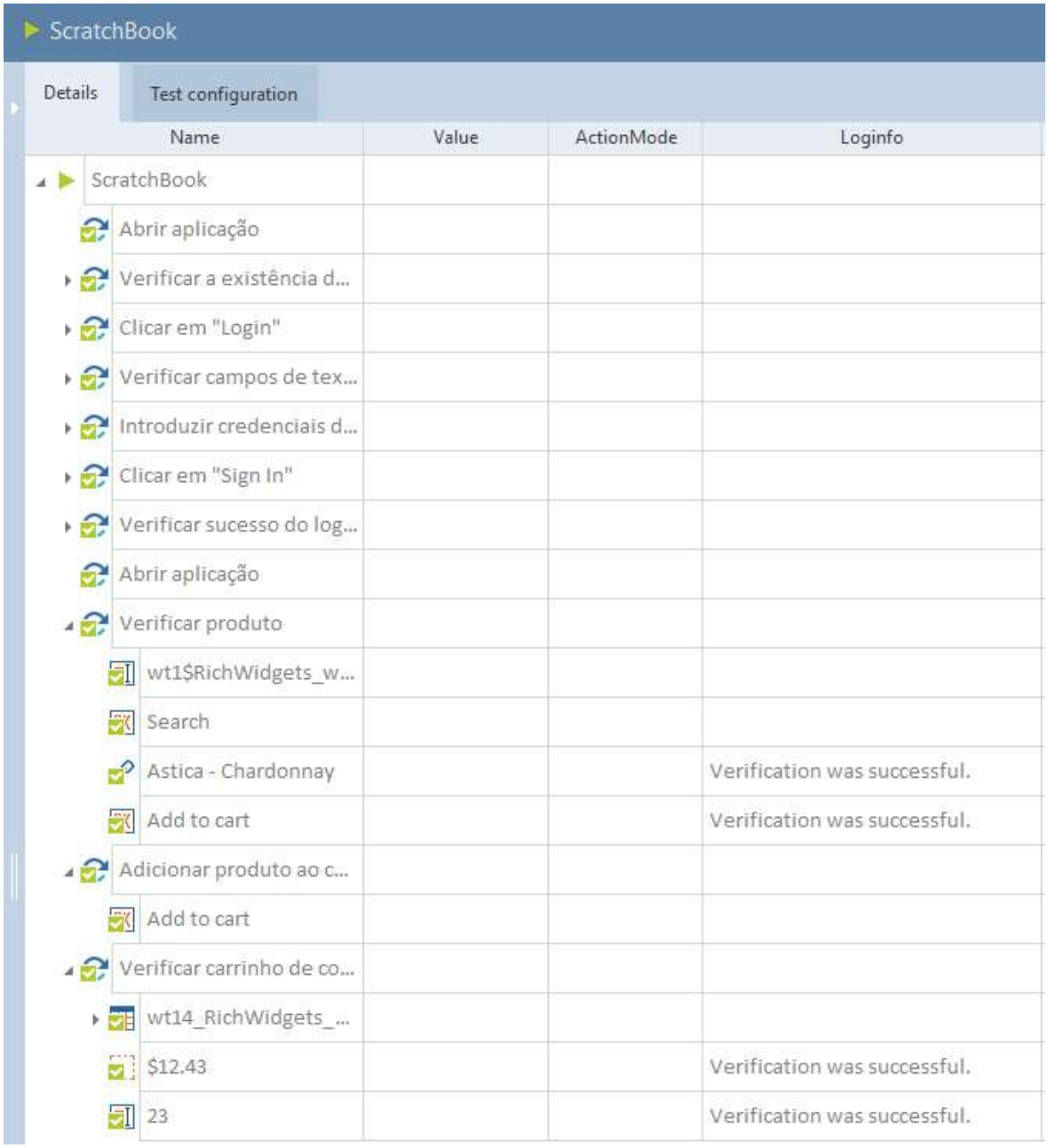

Figura 64 - Execução do 2º cenário (UI) com sucesso.

Um possível cenário de falha para este cenário é o *tester* procurar por um produto diferente, se isto acontecer a verificação do produto na lista após a procura irá falhar, como é possível ver na Figura 65.

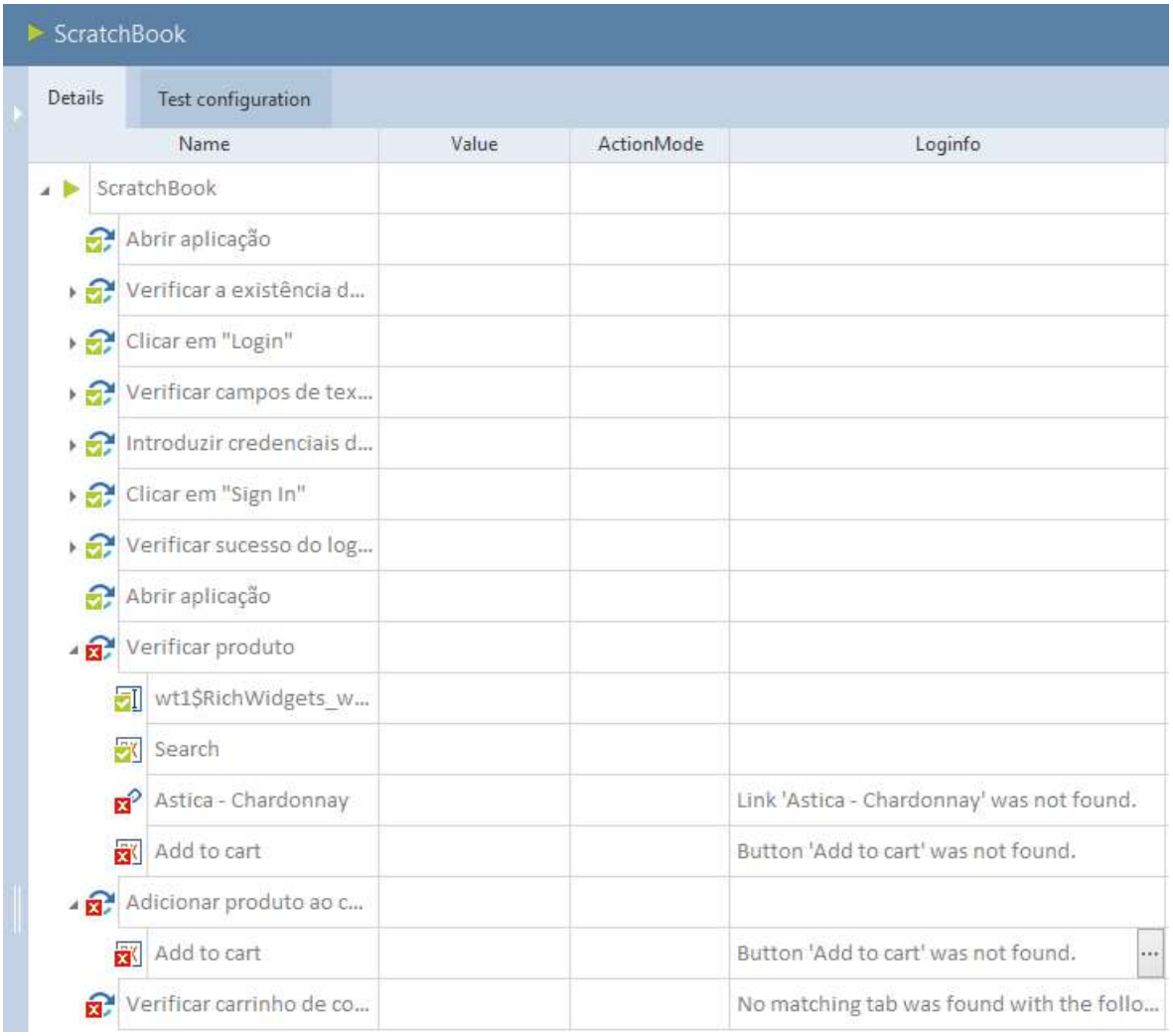

Figura 65 - Execução do 2º cenário (UI) sem sucesso.

Teste de Serviço

Do mesmo modo que foi executado para o cenário anterior, será criada uma API chamada "Cenario2" que irá possuir um método chamado "AdicionarProdutoAoCarrinho" (Figura 66). Este método vai conter a lógica necessária para adicionar um produto ao carrinho onde:

1. Será feita uma pesquisa na base de dados pelo nome do produto introduzido pelo utilizador
2. Caso o produto seja encontrado será adicionado ao carrinho
3. No carrinho serão efetuadas algumas verificações, tais como, quantidade de produtos, nome do produto e quantidade adicionada do produto

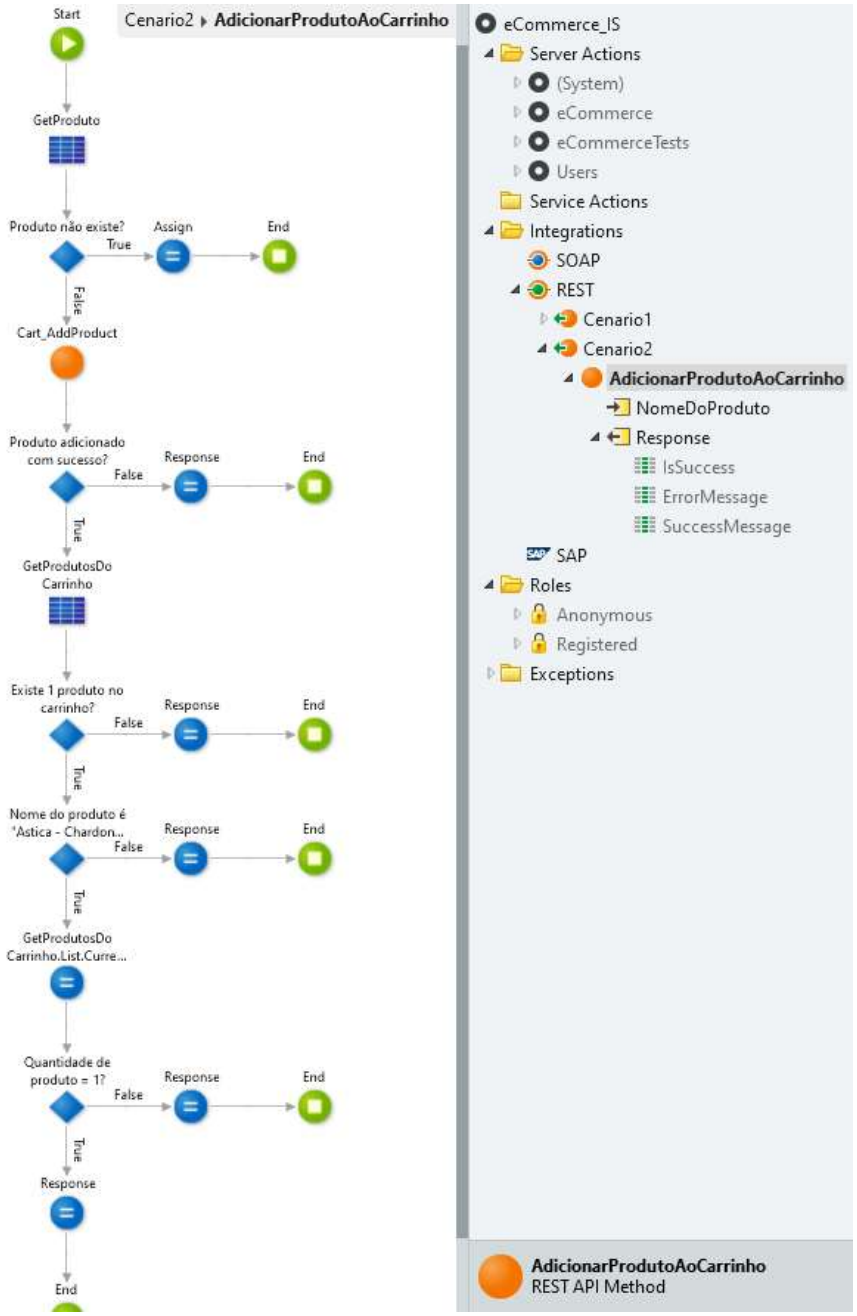

Figura 66 - Ação correspondente ao método da API do segundo cenário.

Após criada a API necessária para este cenário, o tester pode criar o teste na plataforma *Tricentis*. Para isso apenas tem de fazer scan da API como já foi explicado anteriormente. O *tester* deve preencher as informações necessárias como o *endpoint* e a *resource* e também deve enviar como parâmetro o nome do produto que pretende adicionar no carrinho, neste caso "*Astica – Chardonnay*" (Figura 67).

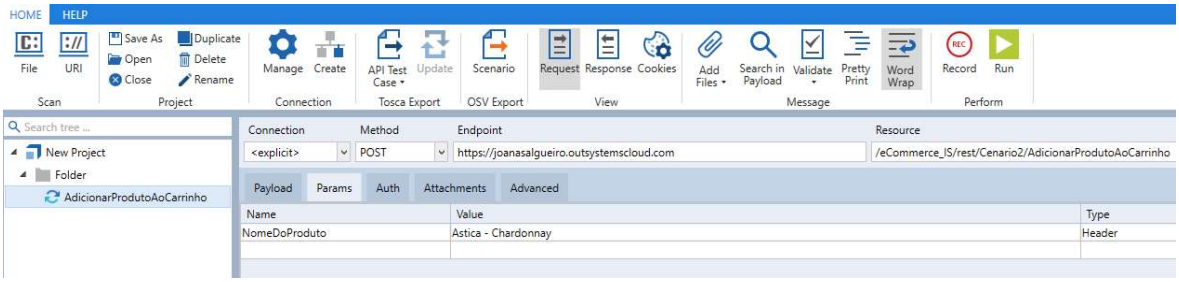

Figura 67 - Teste de API do 2º Cenário.

Após o *tester* clicar em "*Run*" será executado o teste e, caso seja executado com sucesso o resultado será o seguinte (Figura 68):

Figura 68 - Resultado da execução do 2º cenário quando executado com sucesso.

Caso o *tester* indique um produto que não exista na base de dados o teste será executado sem sucesso e terá um resultado semelhante ao seguinte (Figura 69):

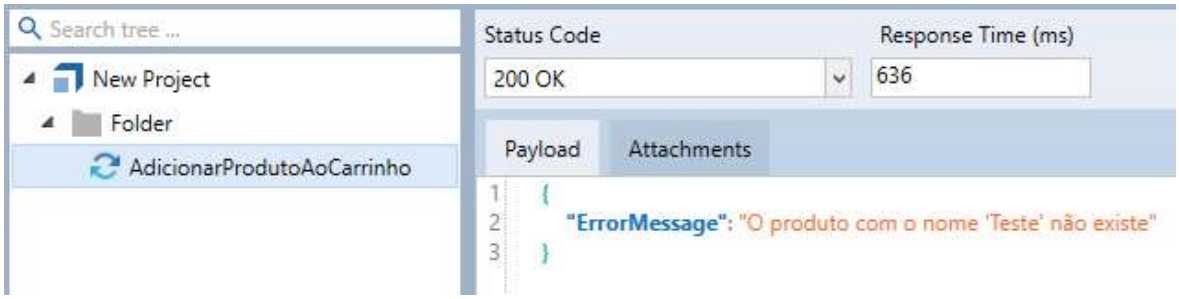

Figura 69 - Resultado do 2º cenário quando executado sem sucesso.

Após a conclusão do *scan* da API o *tester* deve exportar o teste de API criado para um caso de teste e consequentemente adicionar os passos que pretende executar e validar quando o teste é executado. Após serem adicionados os passos pretendidos, neste caso todos, o caso de teste será semelhante à Figura 70. Resumidamente, a figura mostra que o *tester* introduz os campos "*resource*" e "NomeDoProduto", que são os parâmetros necessários à execução do teste. Por fim, as verificações feitas no final são o "*status code*", a variável boleana "*IsSuccess*" e a "*SuccessMessage*".

Name	Value	ActionMode	DataType	WorkState
AdicionarProdutoAoCarrin...				COMPLET...
AdicionarProdutoAoCa...				
Resource	/eCommerce_IS/rest/Cenario2 /AdicionarProdutoAoCarrinho	Insert	String	
NomeDoProduto	Astica - Chardonnay	Insert	String	
AdicionarProdutoAoCa...				
StatusCode	200 OK	Verify	String	
ResponseTime	<= 523	Verify	Numeric	
IsSuccess	True	Verify	Boolean	
SuccessMessage	O produto 'Astica – Chardonnay' foi adicionado ao carrinho com sucesso	Verify	String	

Figura 70 – *Script* do caso de teste 2º cenário (API).

O *tester* pode agora executar o teste através da opção "Run in Scratchbook", para que o teste seja executado com sucesso a variável "*StatusCode*" deve ser igual a 200, a variável "*IsSuccess*" deve ser igual a *True* e a "*SuccessMessage*" deve ser igual a "O produto '*Astica – Chardonnay*' foi adicionado ao carrinho com sucesso". O resultado da execução do teste com sucesso pode ser visualizado na Figura 71.

ScratchBook

Details | Test configuration

Name	Value	ActionMode	Loginfo
ScratchBook			
AdicionarProdutoAoCa...			
Resource			Ok
NomeDoProduto			Ok
AdicionarProdutoAoCa...			Server Response Time: 112 ms
StatusCode			Verification was successful.
IsSuccess			Verification was successful.
SuccessMessage			Verification was successful.

Figura 71 - Resultado da execução do 2º cenário (API) com sucesso.

Caso o tester introduza um produto que não exista na base de dados o teste irá falhar uma vez que não irá verificar os valores *True* para a variável *IsSuccess* nem a mensagem "O produto '*Astica – Chardonnay*' foi adicionado ao carrinho com sucesso", como é possível ver na Figura 72.

Figura 72 - Resultado da execução do 2º cenário (API) sem sucesso.

3º CENÁRIO

Teste de Interface do Utilizador

À semelhança dos cenários anteriores o *tester* deve então gravar a navegação no *browser* de modo a poder ser executado mais tarde através da plataforma *Tricentis*. O *script* do teste está representado na Figura 74.

O utilizador deve aceder à aplicação e verificar a existência da caixa de texto onde será introduzido o nome do país, neste caso, Portugal e também deve verificar a existência do botão "Obter Capital". Após a verificações o utilizador introduz "Portugal" na caixa de texto e clica no botão de modo a obter o resultado. O resultado deve conter a palavra "*Lisbon*", uma vez que Lisboa é a capital de Portugal.

GetCapitalPortugal				COMPLETED
Abrir aplicação				
Url	https://joanasalgueiro.outsystemscloud.com/TestModule/	Input	String	
UseActiveTab	True	Input	String	
Verificar caixa de text e botão de procura				
Nome do País		Verify	String	
Obter Capital	Obter Capital	Verify	String	
Lisbon			String	
Verificar capital de Portugal				
Nome do País	Portugal	Input	String	
Obter Capital	X	Input	String	
Lisbon	InnerText == Lisbon	Verify	String	

Figura 74 - *Script* do caso de teste do 3º cenário (UI).

O diagrama de controlo de fluxo do teste correspondente ao terceiro cenário é apresentado na imagem seguinte (Figura 73):

Figura 73 - Diagrama de controlo de fluxo do 3º cenário (UI).

Após concluído o *script* do teste, o *tester* pode passar à execução do teste e na Figura 75 é possível verificar o resultado da execução do teste com sucesso.

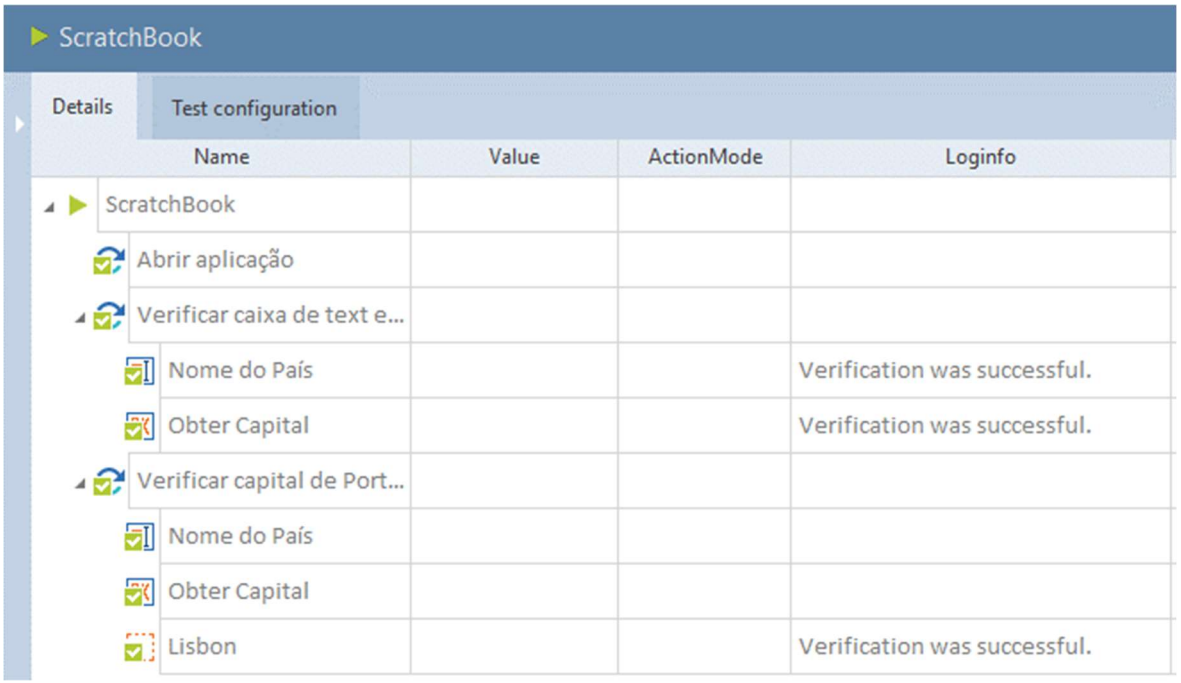

Figura 75 - Execução do 3º cenário (UI) com sucesso.

No caso de o *tester* introduzir um país inválido o teste será executado com falha porque não será devolvida uma capital. Este cenário está representado na Figura 76.

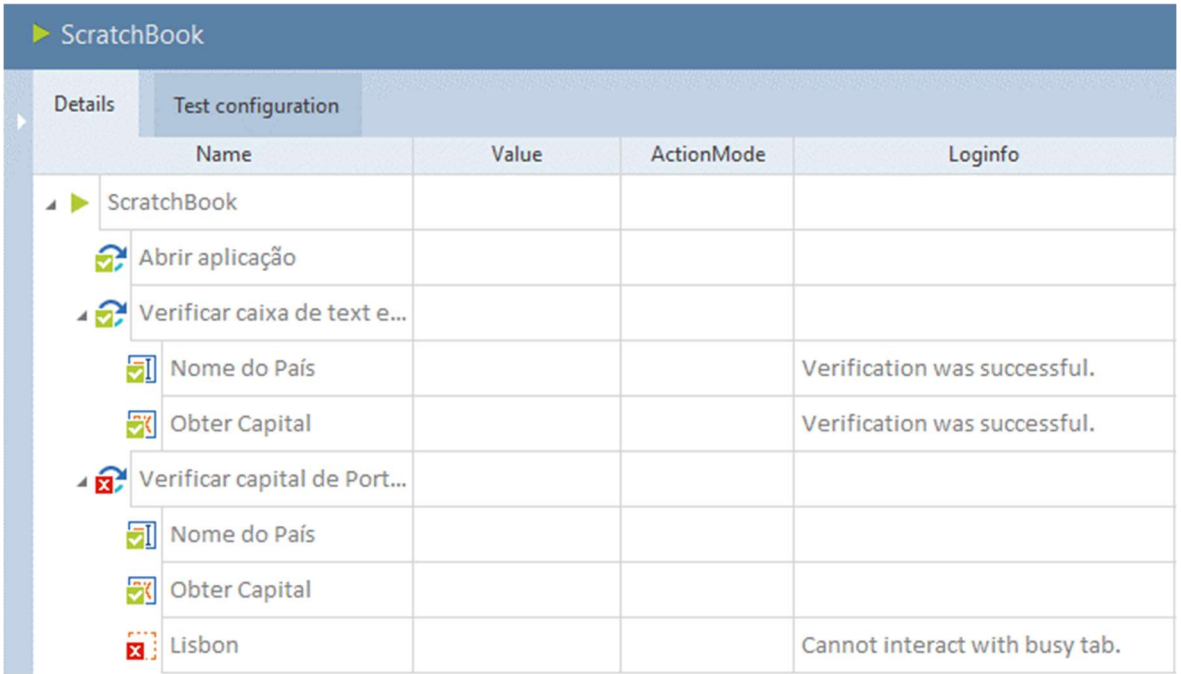

Figura 76 - Execução do 3º cenário (UI) sem sucesso.

Teste de Serviço

Neste cenário foi exposta uma API que possui um método chamado "GetCapitalPaís" que devolve a capital de um dado país como se pode verificar na Figura 77.

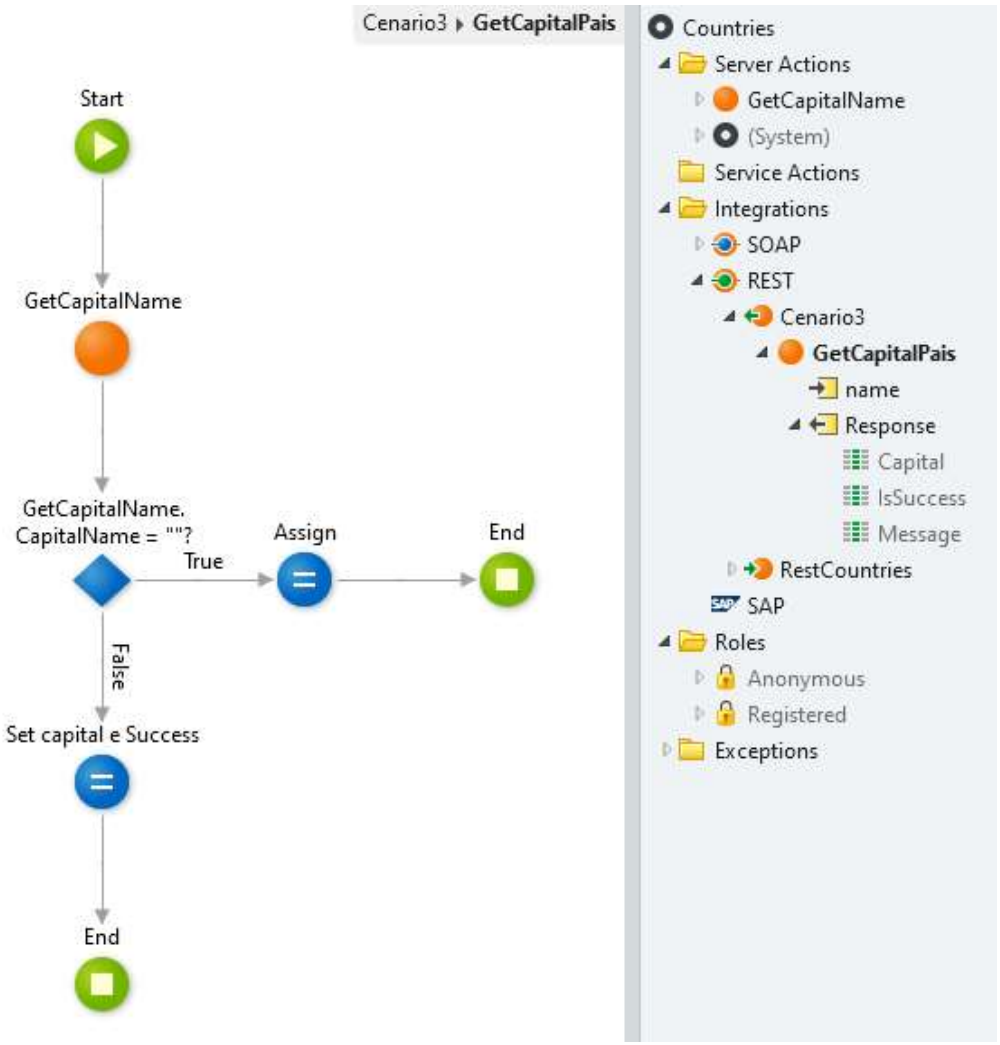

Figura 77 - Ação correspondente ao método da API do terceiro cenário.

De forma semelhante ao que foi explicado para os cenários anteriores, o *tester* deve aceder à secção "*API Testing*" e utilizar a opção "*API Scan*" para criar o teste deste cenário e preencher as informações necessárias para a execução do teste. Neste caso, para além do *endpoint* e da *resource*, o *tester* deve introduzir o nome do país que deseja saber a capital (Figura 78).

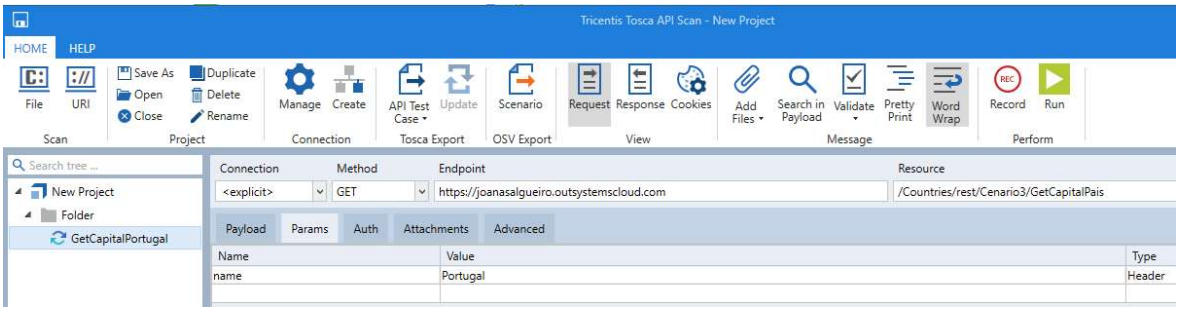

Figura 78 - Teste de API do 3º cenário.

No caso de ser introduzido um país válido, neste caso Portugal, o resultado do teste será o seguinte (Figura 79):

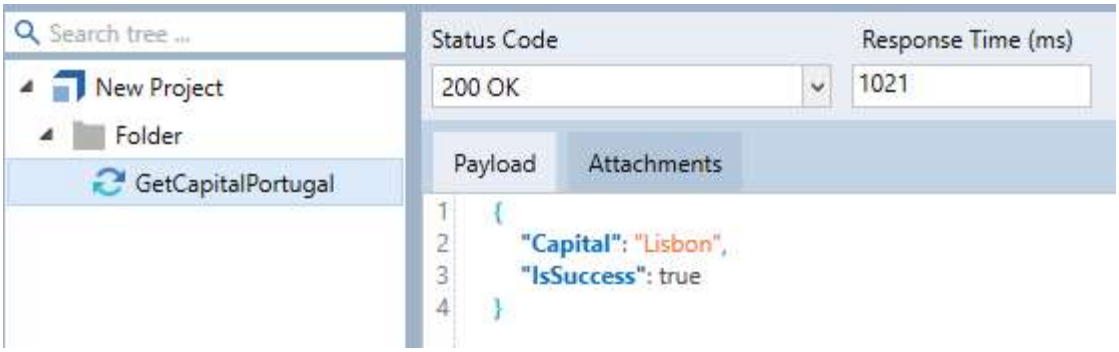

Figura 79 - Resultado da execução do 3º cenário quando executado com sucesso.

Caso o tester introduza um país inválido ou que não esteja configurado na API e não possua uma capital definida, o resultado irá ser diferente do anterior, como é possível ver na Figura 80.

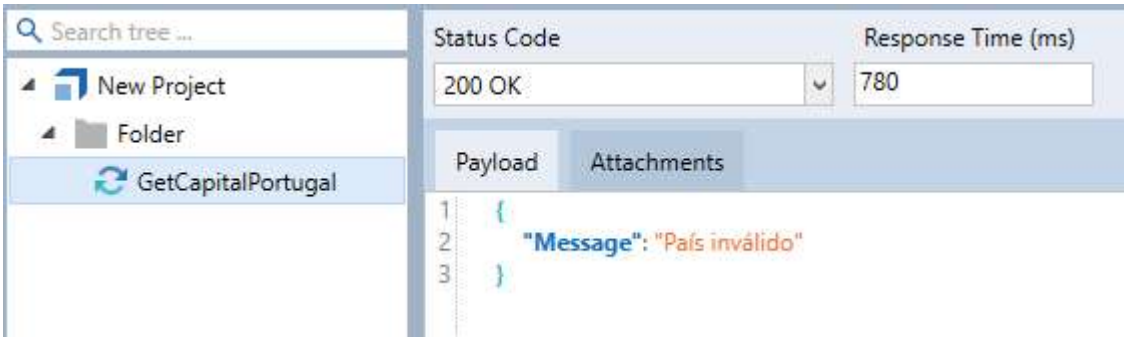

Figura 80 - Resultado da execução do 3º cenário quando executado sem sucesso.

Agora que o tester já fez o *scan* do teste pode construir o caso de teste de forma automatica exportando o teste e selecionando os passos/verificações que pretendem, como foi mostrado para o primeiro cenário. O script do caso de teste terá o aspeto da Figura 81, em que o *tester* introduz os campos "*resource*" e o nome do país que são os parâmetros necessários à execução do teste e, após este ser executado é verificado o "*status code*" que deve ser igual a 200, bem como as variáveis "*IsSuccess*" e "*Capital*" retornadas pela API.

Name	Value	ActionMode	DataType	WorkState
GetCapitalName				COMPLET...
GetCapitalPortugal Req...				
Resource	/Countries/rest/Cenario3/GetCapitalPais	Insert	String	
name	Portugal	Insert	String	
GetCapitalPortugal Res...				
StatusCode	200 OK	Verify	String	
ResponseTime	<= 553	Verify	Numeric	
Capital	Lisbon	Verify	String	
IsSuccess	True	Verify	Boolean	

Figura 81 – *Script* do caso de teste 3º cenário (API).

Agora que o caso de teste já está finalizado o *tester* pode executá-lo e ver se este é ou não executado com sucesso. Para executar o teste, o *tester* tem que selecionar o caso de teste e clicar em "*Run in Scratchbook*". O resultado será o seguinte (Figura 82):

ScratchBook

Details | Test configuration

Name	Value	ActionMode	Loginfo
ScratchBook			
GetCapitalPortugal Request			
Resource			Ok
name			Ok
GetCapitalPortugal Response			Server Response Time: 482 ms
StatusCode			Verification was successful.
ResponseTime			Verification was successful.
Capital			Verification was successful.
IsSuccess			Verification was successful.

Figura 82 - Resultado da execução do 3º cenário (API) com sucesso

Da mesma forma que foi executado um caso de teste falhado para UI também pode ser reproduzido esse mesmo caso no teste de API, basta que o *tester* introduza um país inválido. O resultado pode ser visto na Figura 83.

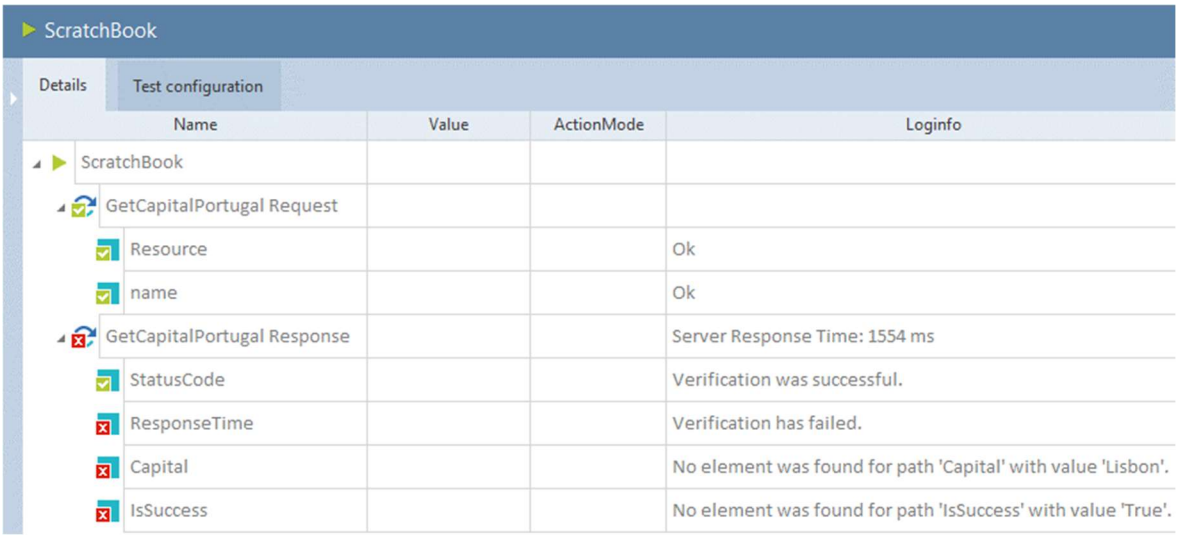

Figura 83 - Resultado da execução do 3º cenário (API) sem sucesso.

Observações finais

Visto que esta ferramenta possui a vantagem de permitir executar tanto testes ao nível do UI como ao nível de serviços e API as boas práticas mencionadas anteriormente para as outras ferramentas são também aplicadas aqui.

Nos testes de serviços a boa prática aplicada é a reutilização de lógica desenvolvida posteriormente para as funcionalidades da aplicação o que poupa tempo aos *developers* e previne erros nos testes. No caso da API, esta é também isolada num módulo e apenas a ação correspondente é chamada, neste caso, pela API que é exposta para ser utilizada no teste.

Nos testes ao nível de UI são aplicadas as boas práticas relacionadas com as validações dos elementos no ecrã como já foi referido anteriormente no caso dos cenários implementados com o *Ghost Inspector*.

5.4. Discussão e Análise de Resultados

Existem algumas conclusões que podem ser tiradas após a implementação e execução dos cenários de teste com as diferentes plataformas, tanto a nível de tipos de testes executados como a nível da experiência necessária.

É preciso ter em consideração que as três ferramentas foram usadas para implementação e execução de testes para níveis de teste distintos. Primeiramente, a *BDDFramework* apenas permite implementar e executar testes unitários/componentes que testam a lógica de negócio associada às várias funcionalidades da aplicação a serem testadas. A segunda ferramenta, o *Ghost Inspector*, ao contrário da *BDDFramework* não permite executar testes unitários/componentes, mas permite testar a aplicação ao nível da interface (testes de sistema) através da gravação do script de teste pelo utilizador através do *browser*. Por fim, a *Tricentis Tosca* é uma ferramenta mais completa e permite executar ambos os tipos de testes mencionados anteriormente, testes unitários/componentes e testes de integração, através de *APIs* expostas que contém a lógica a ser testada e testes de sistema através da gravação do teste pelo utilizador no *browser* tal como no *Ghost Inspector*.

Os testes implementados com a *BDDFramework* são normalmente da responsabilidade dos *developers* das aplicações a serem testadas e obrigam a ter conhecimento *OutSystems*. O facto de os testes serem da responsabilidade dos *developers* faz com que estes tenham que dedicar tempo às atividades de teste durante o projeto, ou seja, tempo extra de desenvolvimento. Em relação ao *Ghost Inspector*, os testes podem ser implementados e executados pelos *testers* uma vez que apenas é necessário gravar o teste através do browser. Contudo, os *testers* devem ter conhecimento de *JavaScript* de modo a entender e a poder editar os seletores selecionados pela ferramenta de forma automática. A ferramenta *Tricentis Tosca* permite executar ambos os tipos de teste. Ambos os testes podem ser implementados e executados pelos *testers*, mas, no caso dos testes unitários é necessário que os *developers* implementem as *APIs* essenciais à execução dos testes para que estes possam ser implementados e executados. No caso dos testes de sistema o processo é semelhante ao *Ghost Inspector*.

A nível de complexidade, todas as ferramentas são simples de utilizar, contudo no caso da *BDDFramework* é necessário ter conhecimento *OutSystems* até porque é uma ferramenta desenvolvida em *OutSystems* e para aplicações *OutSystems*. O *Ghost Inspector* é a mais simples de utilizar, tem uma interface bastante intuitiva e simples. Em relação à *Tricentis Tosca*, é uma ferramenta com bastantes funcionalidades e com uma plataforma própria o que faz com que se torne uma ferramenta mais complexa, mas existem muitos cursos online que permitem aprender a trabalhar com ela.

Uma das vantagens que existe na *BDDFramework* e que é uma desvantagem para o *Ghost Inspector* e para a *Tricentis Tosca* é o custo. A *BDDFramework* não possui nenhum tipo de custo para ser utilizada e é atualmente a ferramenta aconselhada para implementar e executar os testes para aplicações *OutSystems*. Tanto o *Ghost Inspector*

como a *Tricentis Tosca*, apesar de terem períodos gratuitos, possuem custos para a sua utilização e de custo elevado o que faz com que os clientes não tenham tanto interesse pois, acrescenta custos ao projeto.

Em suma, nem sempre é uma questão de escolher qual das ferramentas escolher e usar, uma vez que elas são usadas em níveis de teste distintos e, por isso, complementam-se. Isto acontece principalmente em relação à *BDDFramework* pois é a atualmente a única ferramenta de teste, desenvolvida em *OutSystems*, que permite a implementação de testes unitários em *OutSystems*. Já em relação às restantes duas ferramentas de teste, a mais completa acaba por ser a *Tricentis Tosca,* contudo, o cliente pode não querer utilizar devido ao incremento no valor monetário do projeto.

6. Opinião dos profissionais das TI sobre testes em *low-code*

Neste capítulo é apresentado o inquérito que foi elaborado para complementar a discussão resultante da execução dos cenários de teste apresentados no capítulo anterior. O inquérito foi dirigido a profissionais da área das Tecnologias de Informação, com experiência em desenvolvimento *OutSystems*, com o objetivo de analisar a perceção que estes têm sobre a importância dos testes de software no desenvolvimento *low-code* e sobre a influência das boas práticas de desenvolvimento no processo de automatização de testes.

6.1. Objetivo do inquérito

O inquérito tem como objetivo identificar e caracterizar a experiência e importância atribuída pelos participantes às práticas de teste de software, a sua perceção sobre a influência que a descrição das funcionalidades (e.g., *user stories*) e as boas práticas no desenvolvimento têm sobre a prática da automatização de testes, bem como caracterizar a utilização de ferramentas/*frameworks* para testes por eles usadas.

6.2. Participantes

O inquérito é dirigido a profissionais que trabalhem na área das Tecnologias da Informação em atividades relacionadas com a utilização da plataforma de desenvolvimento *low-code OutSystems*.

O inquérito foi implementado no *google forms* e foi enviado para profissionais que usam a plataforma *OutSystems* de 4 empresas de software.

6.3. Estrutura do inquérito

O inquérito está organizado em duas secções. A primeira secção destina-se à caracterização dos respondentes quanto à sua experiência na área das Tecnologias da Informação e, em especial, com plataformas de desenvolvimento *low-code* e na área de testes e qualidade de software. Esta secção resume-se às seguintes 7 questões:

1. Qual a sua idade?
    a. 20 anos ou menos
    b. 21-25 anos
    c. 26-30 anos
    d. 31-35 anos
    e. 36-40 anos
    f. Mais de 40 anos
2. Quantos anos de experiência tem na área das TI?
    a. 0-2 anos

b. 3-5 anos
c. 6-10 anos
d. 11-15 anos
e. 16-20 anos
f. Mais de 20 anos

3. Das seguintes tecnologias/ferramentas de desenvolvimento de software indique aquelas que usa/usou na sua atividade profissional.
    a. Java
    b. C#
    c. JS
    d. Python
    e. PHP
    f. HTML/CSS
    g. Outra
4. Das seguintes plataformas de desenvolvimento *Low-Code* indique aquelas que usa/usou na sua atividade profissional.
    a. OutSystems
    b. Mendix
    c. Appian
    d. PowerApps
    e. Outra
5. Atualmente, a qual das seguintes atividades dedica mais tempo da sua atividade profissional?
    a. Analista
    b. Software *engineering*
    c. Software *architect*
    d. Desenvolvedor/*Developer*
    e. Testador/*Tester*
    f. Líder/Gestor de equipa
    g. Outra
6. Caso a sua atividade profissional envolva desenvolvimento, para que plataformas desenvolve?
    a. Web
    b. Android
    c. iOS
    d. Multiplataforma
    e. Outra
7. Quais das seguintes metodologias de desenvolvimento é mais comum nos projetos em que participa na sua atividade profissional?
    a. Scrum
    b. Waterfall
    c. Lean
    d. Kanban

e. *Extreme Programming* (XP)
f. Outra

A segunda secção destina-se apenas aos participantes que têm alguma experiência em atividades de teste. O seu objetivo é permitir uma caracterização dos respondentes quanto à perceção que têm sobre as atividades de teste, e sobre a forma como a descrição das funcionalidades e as práticas de desenvolvimento influenciam as atividades de automatização de teste e também obter uma caracterização sobre as ferramentas/*frameworks* para testes que usam. Esta secção possui 9 questões, que são as seguintes:

1. Para si, qual/quais das seguintes frases melhor descreve a atividade de teste?
    a. Os testes não são importantes, e apenas servem para atrasar o projeto
    b. Os testes são importantes, mas não são adequados quando se usam metodologias ágeis
    c. Os testes devem ser feitos apenas por pessoas dessa área e os *developers* não precisam de fazer testes
    d. Os testes são importantes e devem ser realizados independentemente da metodologia de desenvolvimento utilizada
2. Quando se encontra a preparar os testes, com que frequência tem dificuldade em avaliar o que deve, e como deve, ser testado?
    a. Nunca
    b. Raramente
    c. Algumas vezes
    d. Muitas vezes
    e. Sempre
3. A forma como as funcionalidades são descritas (casos de uso, *user stories*, etc.) contribuem para facilitar a conceção dos testes?
    a. Discordo muito
    b. Discordo
    c. Não concordo nem discordo
    d. Concordo
    e. Concordo muito
4. A forma como o código é desenvolvido contribui para facilitar a atividade de teste (escrever, implementar, e executar os casos de teste)?
    a. Discordo muito
    b. Discordo
    c. Não concordo nem discordo
    d. Concordo
    e. Concordo muito

5. Para a escrita dos casos de teste recorre a alguma técnica conhecida ou escreve os casos de testes de acordo com a sua perceção (bom-senso) do que deverá ser mais adequado testar?
    a. Usando o bom-senso
    b. Usando técnicas de conceção reconhecidas. Quais?
    c. Outra
6. Nas suas atividades de teste, quando realiza testes funcionais, a que nível os realiza mais frequentemente?
    a. Testes unitários/componentes
    b. Testes de integração
    c. Testes de sistema
7. Caso na sua atividade profissional inclua implementação e execução de testes, e caso utilize alguma ferramenta de teste, indique qual/quais das seguintes ferramentas/plataformas já usou/usa.
    a. BDDFramework
    b. Ghost Inspector
    c. Tricentis Tosca
    d. Leapwork
    e. Outra
8. Caso utilize a *BDDFramework* na sua atividade de teste como classifica a sua experiência com esta ferramenta?
    a. Muito negativa
    b. Negativa
    c. Nem negativa nem positiva
    d. Positiva
    e. Muito positiva
9. Em relação à sua resposta da questão anterior indique qual ser o aspeto mais positivo (ponto forte) e o aspeto mais negativo (ponto fraco) da ferramenta que usa.

6.4. Análise dos resultados dos inquéritos

No total foram obtidas 27 respostas ao inquérito.

Com a primeira pergunta, relativa à idade dos participantes é possível verificar que maior parte dos participantes do inquérito (48%) tem entre 26 e 30 anos, sendo que 26% possui entre 36 e 40 anos. O gráfico com o resultado desta questão pode ser visto na Figura 85.

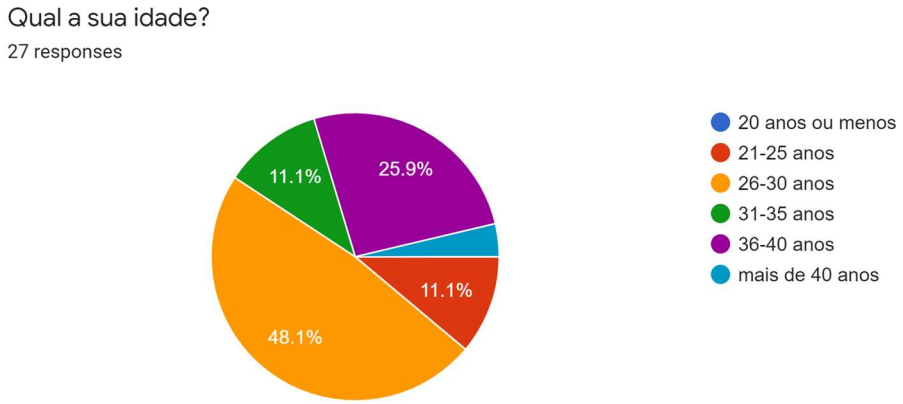

Figura 85 - Gráfico correspondente à pergunta "Qual a sua idade?".

Relativamente à questão sobre quantos anos de experiência na área das Tecnologias de Informação possuem os participantes, as respostas aos inquiridos permitem concluir 88.8% dos respondentes tem entre 3 e 15 anos de experiência, sendo que, 41% possui entre 3 a 5 anos de experiência, 26% entre 6 a 10 anos e 22% entre 11 a 15 anos (Figura 84).

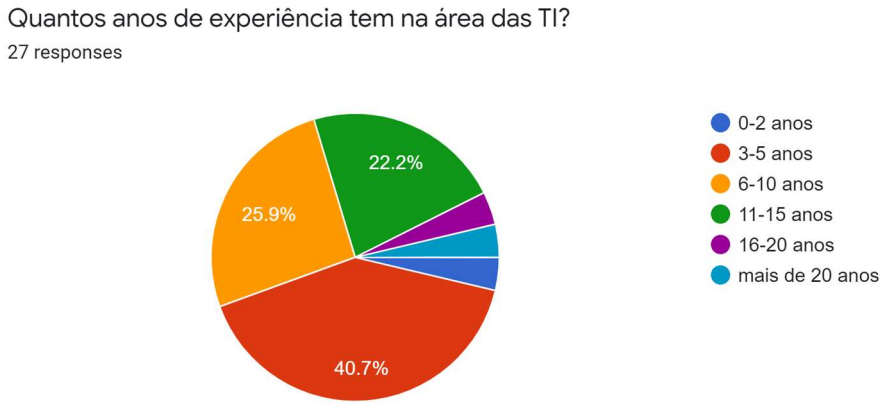

Figura 84 - Gráfico correspondente à pergunta "Quantos anos de experiência tem na área das TI?".

Em relação à terceira pergunta sobre as tecnologias utilizadas, a maior parte dos inquiridos respondeu que usa, ou já usou HTML/CSS, *JavaScript*, C#, Java e PHP, como é possível ver na Figura 86.

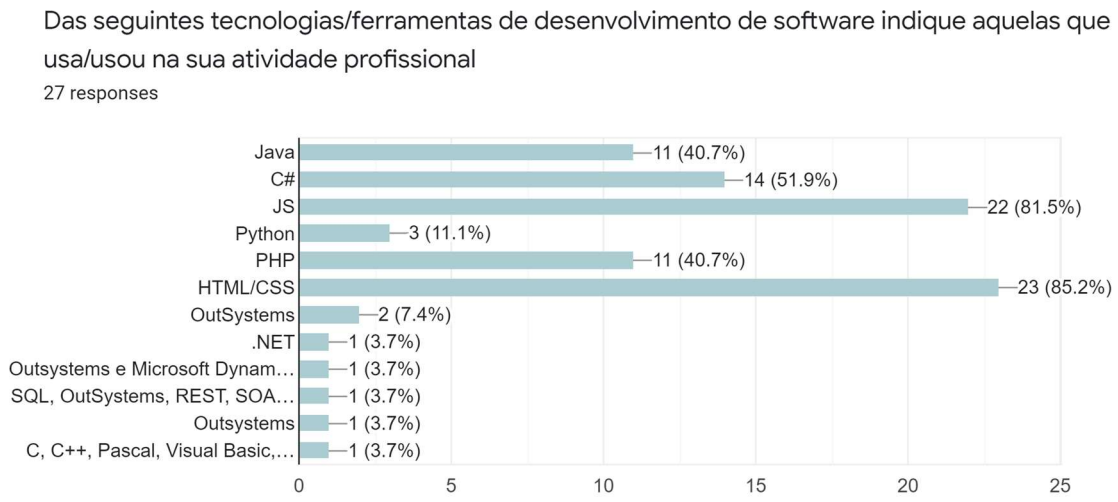

Figura 86 - Gráfico correspondente à pergunta "Das seguintes tecnologias/ferramentas de desenvolvimento de software indique aquelas que usa/usou na sua atividade profissional"

A quarta pergunta, "Das seguintes plataformas de desenvolvimento *Low-Code* indique aquelas que usa/usou na sua atividade profissional", 100% dos participantes responderam que usam, ou já usaram, *OutSystems* e 2 deles também usam, ou já usaram, a plataforma *PowerApps* e a plataforma *Lights witch*, *Radzen* (Figura 87).

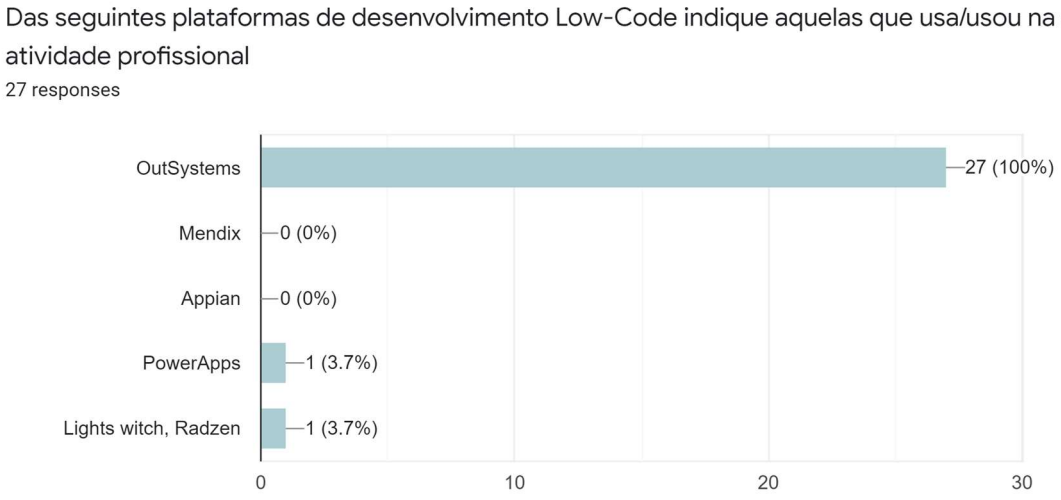

Figura 87 - Gráfico correspondente à pergunta "Das seguintes plataformas de desenvolvimento *Low-Code* indique aquelas que usa/usou na sua atividade profissional".

A quinta pergunta destina-se a perceber a que atividade profissional os participantes dedicam mais tempo atualmente. Na Figura 88, é possível verificar que quase 60% dos respondentes é *developer* e que 22% dos respondentes é líder ou gestor de equipa.

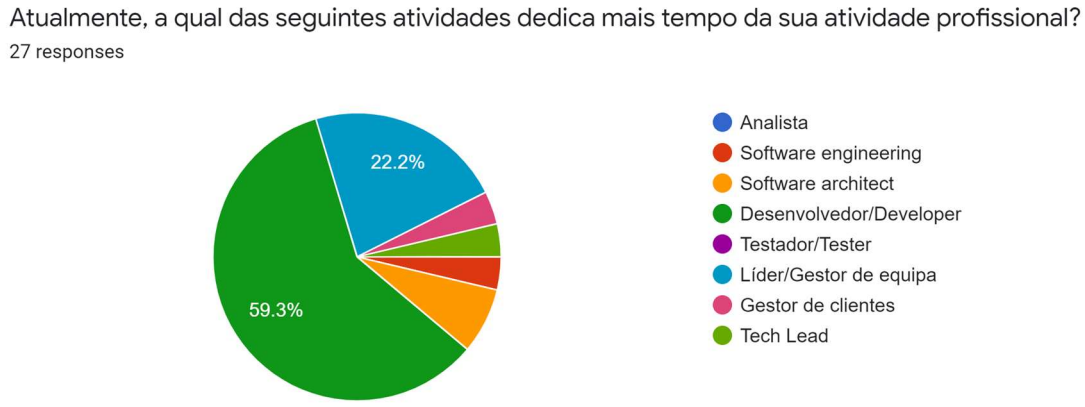

Figura 88 - Gráfico correspondente à pergunta "Atualmente, a qual das seguintes atividades dedica mais tempo da sua atividade profissional?".

A sexta pergunta obteve 26 respostas e estas dividem-se entre “Multiplataforma” (73%) e “Web” (27%), como é possível ver na Figura 89.

Figura 89 - Gráfico correspondente à pergunta "Caso a sua atividade profissional envolva desenvolvimento, para que plataformas desenvolve?".

Com a última pergunta, da primeira secção, pretende-se entender qual a metodologia de desenvolvimento que é mais comum nos projetos dos participantes. Com um total de 27 respostas, apenas 1 dos participantes respondeu “*Lean*” enquanto 26 participantes responderam “*Scrum*” (Figura 90).

Figura 90 - Gráfico correspondente à pergunta "Quais das seguintes metodologias de desenvolvimento é mais comum nos projetos em que participa na sua atividade profissional?".

A segunda parte do inquérito obteve menos respostas uma vez que era dirigida apenas aos profissionais com experiência na área dos testes de software.

A primeira pergunta tem como objetivo entender a forma como os participantes vêm a atividade de testes. Com um total de 25 respostas, 100% dos participantes descreve a atividade de testes como "Os testes são importantes e devem ser realizados independentemente da metodologia de desenvolvimento utilizada" (Figura 91).

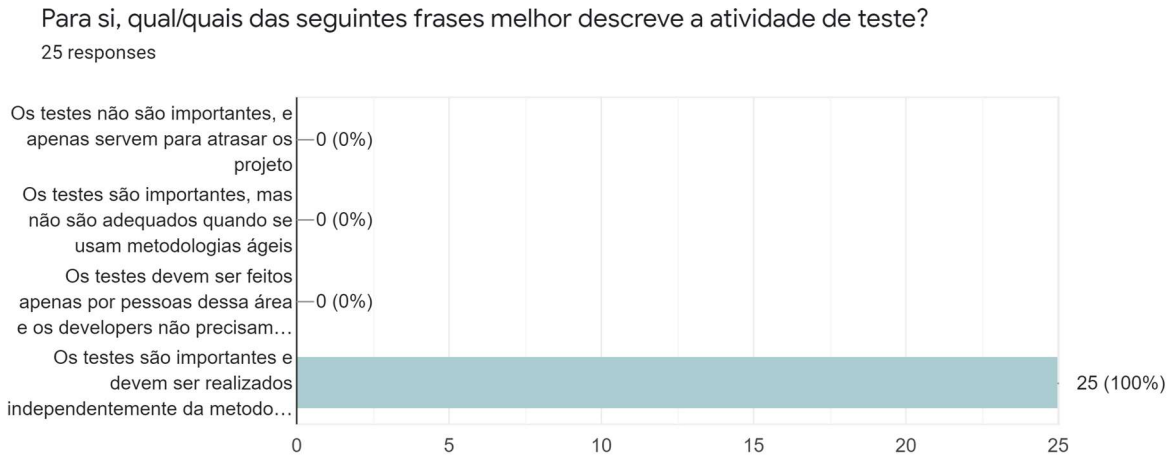

Figura 91 - Gráfico correspondente à pergunta "Para si, qual/quais das seguintes frases melhor descreve a atividade de teste?".

N segunda pergunta, com o objetivo avaliar a dificuldade dos participantes em avaliar o que deve ser testado e como deve ser testado, mais de 50% dos respondentes (13 de 25) respondeu que sente essas dificuldades algumas vezes e ainda 16%, 4 participantes, sente dificuldades muitas vezes o que pode querer dizer que maior parte

destas pessoas não estão totalmente preparadas para desenvolver este tipo de atividade na sua área profissional.

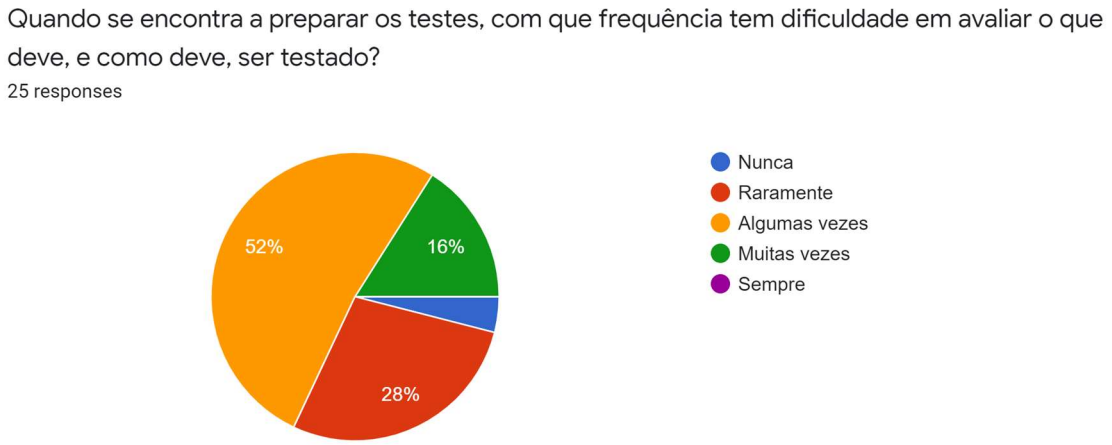

Figura 92 - Gráfico correspondente à pergunta "Quando se encontra a preparar os testes, com que frequência tem dificuldade em avaliar o que deve, e como deve, ser testado?".

Na quarta pergunta, Figura 93, pode-se verificar que em 25 respostas, 13 participantes responderam que concordam muito, e 9 que concordam, que a forma como as funcionalidades são descritas (casos de uso, *user stories*, etc.) contribuem para facilitar a conceção dos testes.

Figura 93 - Gráfico correspondente à pergunta "A forma como as funcionalidades são descritas (casos de uso, *user stories*, etc.) contribuem para facilitar a conceção dos testes?"

À pergunta "A forma como o código é desenvolvido contribui para facilitar a atividade de teste (escrever, implementar, e executar os casos de teste)?", 11 dos 25 participantes responderam que concordam, 8 responderam que concordam muito, 5 nem concordam nem discorda e apenas 1 respondeu que discorda (Figura 94). Ou seja,

76% dos respondentes reconhece que a forma como desenvolvem o seu software tem implicações nas atividades de teste desse software.

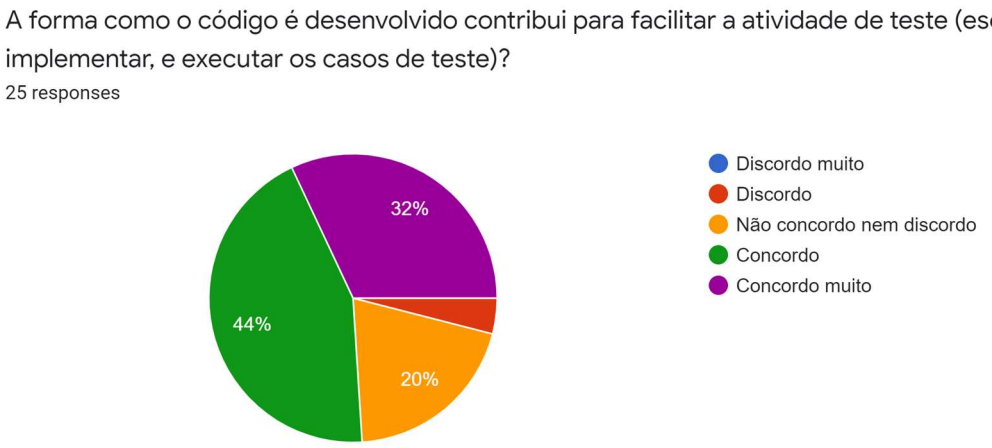

Figura 94 - Gráfico correspondente à pergunta "A forma como o código é desenvolvido contribui para facilitar a atividade de teste (escrever, implementar, e executar os casos de teste)?".

A quinta pergunta, que pretende avaliar as técnicas utilizadas pelos participantes quanto à escrita de casos de teste, 33% dos participantes (8 de 24) responderam que utilizam o bom senso para escrever os casos de teste e 46% (11 de 24) responderam que utilizam técnicas de conceção reconhecidas, tais como, BDD (*Given – When – Then*) e critérios de aceitação das *user stories* (Figura 95).

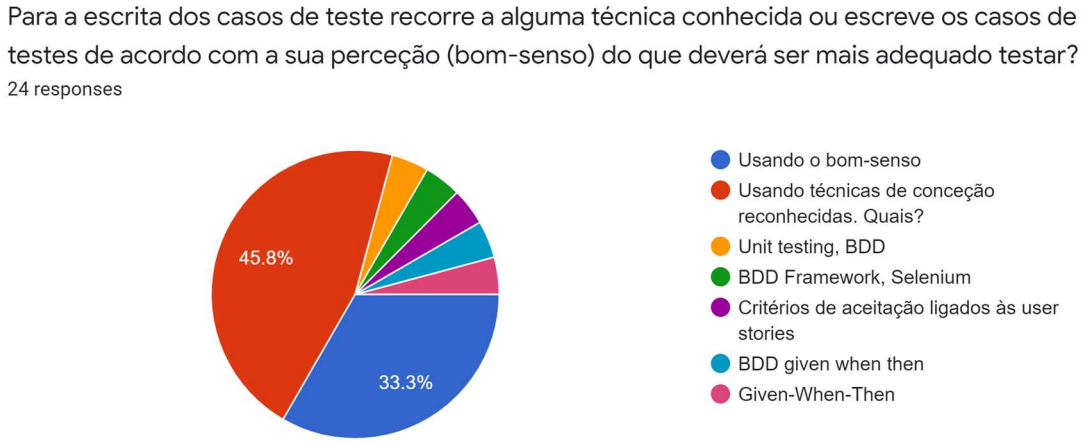

Figura 95 - Gráfico correspondente à pergunta "Para a escrita dos casos de teste recorre a alguma técnica conhecida ou escreve os casos de testes de acordo com a sua perceção (bom-senso) do que deverá ser mais adequado testar?"

À pergunta “Nas suas atividades de teste, quando realiza testes funcionais, a que nível os realiza mais frequentemente?”, 74% dos participantes (17 de 23) respondeu que realiza testes unitários/componentes, 17% (4 de 23) respondeu que realiza testes de sistema e, por fim, 9% (2 de 23) respondeu que realiza testes de integração. Podemos então concluir que a maior parte dos participantes realiza testes unitários durante as suas atividades de teste (Figura 96). Estas respostas parecem estar em linha com o facto de um número significativo dos respondentes ser atualmente developers, sendo, por isso, mais comum a realização de testes unitários/componentes.

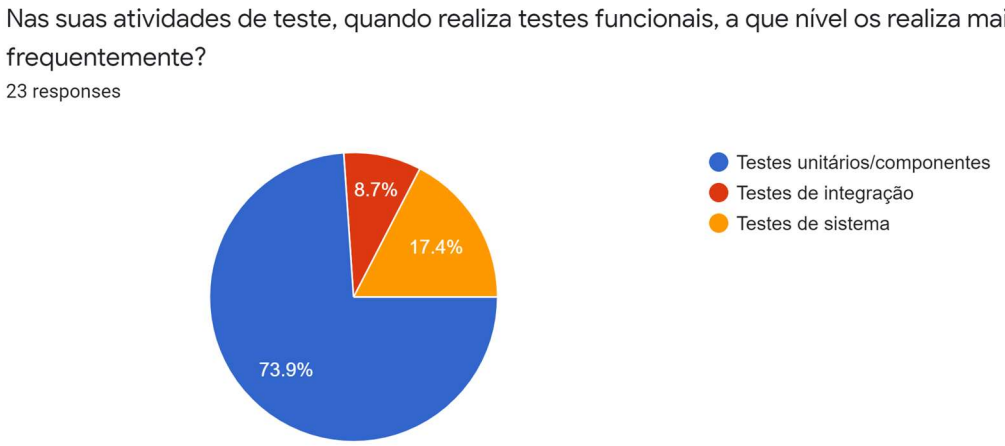

Figura 96 - Gráfico correspondente à pergunta "Nas suas atividades de teste, quando realiza testes funcionais, a que nível os realiza mais frequentemente?".

De modo a tentar perceber se os participantes utilizam alguma das ferramentas de teste apresentadas anteriormente fez-se a pergunta “Caso na sua atividade profissional inclua implementação e execução de testes, e caso utilize alguma ferramenta de teste, indique qual/quais das seguintes ferramentas/plataformas já usou/usa”. 100% dos participantes (14) respondeu que já utilizou a *BDDFramework*, e também 1 participante já utilizou a *Tricentis Tosca* e *Katalon*, é possível ver este resultado na Figura 97.

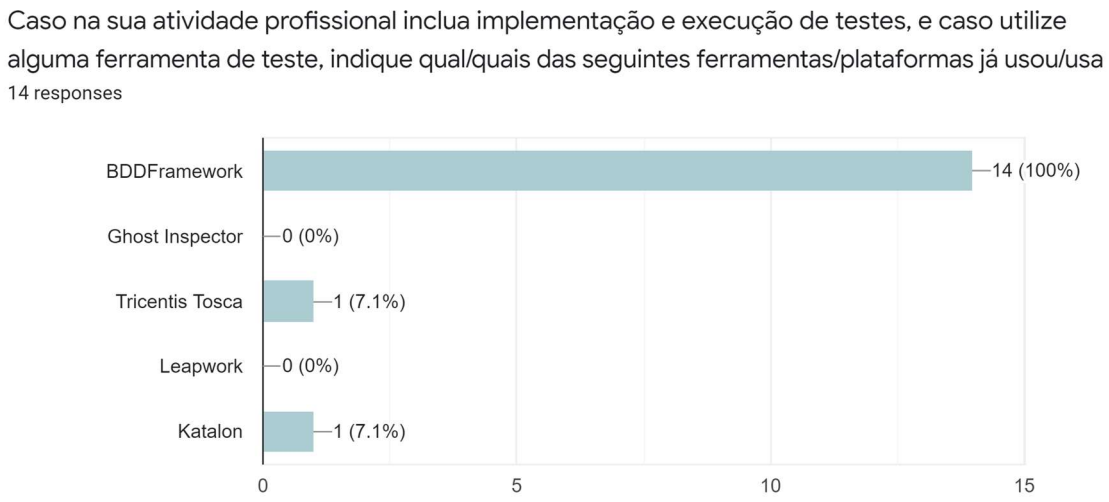

Figura 97 - Gráfico correspondente à pergunta "Caso na sua atividade profissional inclua implementação e execução de testes, e caso utilize alguma ferramenta de teste, indique qual/quais das seguintes ferramentas/plataformas já usou/usa".

Quanto à experiência com a ferramenta *BDDFramework*, pergunta "Caso utilize a *BDDFramework* na sua atividade de teste como classifica a sua experiência com esta ferramenta?", 53% dos participantes (8) respondeu que teve ou tem uma experiência bastante positiva, 33% (5) respondeu que a experiência não foi positiva nem negativa e, por fim, 13%, ou seja, 2 participantes responderam que tiveram ou têm uma experiência muito positiva com a *BDDFramework*, Figura 98.

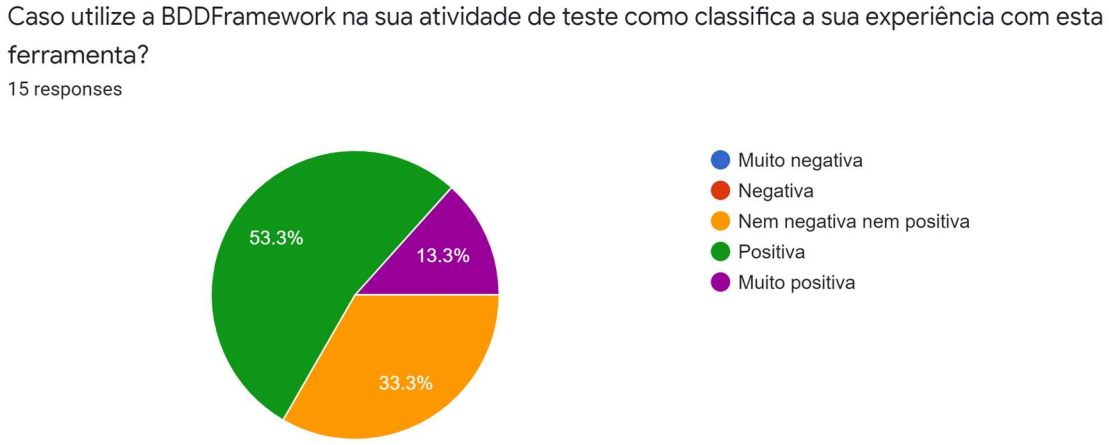

Figura 98 - Gráfico correspondente à pergunta "Caso utilize a BDDFramework na sua atividade de teste como classifica a sua experiência com esta ferramenta?"

Por fim, a última pergunta "Em relação à sua resposta da questão anterior indique qual ser o aspeto mais positivo (ponto forte) e o aspeto mais negativo (ponto fraco) da ferramenta que usa.", é uma pergunta de resposta aberta e os pontos fortes da BDDFramework na opinião dos participantes são:

- Facilidade uso e organização dos testes;
- Os testes são desenvolvidos orientados à *user story*, o que possibilita o mapeamento entre tarefa-teste;

A nível de pontos fracos os participantes referiram os seguintes:

- Grande dependência da *user story*;
- Se a *user story* não estiver bem escrita, os testes podem não ser implementados de forma correta;
- Requer tempo extra para serem implementados, o que pode causar grande impacto no tempo de entrega do projeto;
- Em *agile*, se os requisitos mudarem muito, os testes desenvolvidos poderão tornar-se inúteis e, portanto, há uma perda de tempo;
- Gera um esforço extra na preparação.

No geral, a utilização da *BDDFramework* possui mais pontos fracos do que fortes, na opinião dos respondentes do inquérito, contudo é uma ferramenta de fácil utilização. A questão de os testes estarem relacionados com as *user stories* é também um ponto de discordância entre os participantes pois, uns dizem que possibilita o mapeamento entre tarefa-teste enquanto há outros que dizem que o facto de serem dependentes das *user stories* faz com que possam ocorrer erros de execução caso estas sejam alteradas.

6.5. Impacto dos anos de experiência na área dos testes

Uma das perguntas iniciais do questionário é "Quantos anos de experiência tem na área das TI?". Será que o número de anos de experiência que os participantes possuem interfere de alguma forma com o conhecimento ou técnicas aplicadas ao nível dos testes?

Primeiramente, foi verificada a relação entre o número de anos de experiência e a importância da forma como o código das aplicações é desenvolvido para facilitar a atividade de testes. Como é possível verificar na Figura 99, apenas 15% dos participantes que se possuem entre 11 e 15 anos de experiência discordam que o código desenvolvido pode facilitar a implementação dos testes, o que faz com que se conclua que o número de anos de experiência em nada interfere com este aspeto visto que por vezes as pessoas com menos experiência podem não dar tanta importância aos testes.

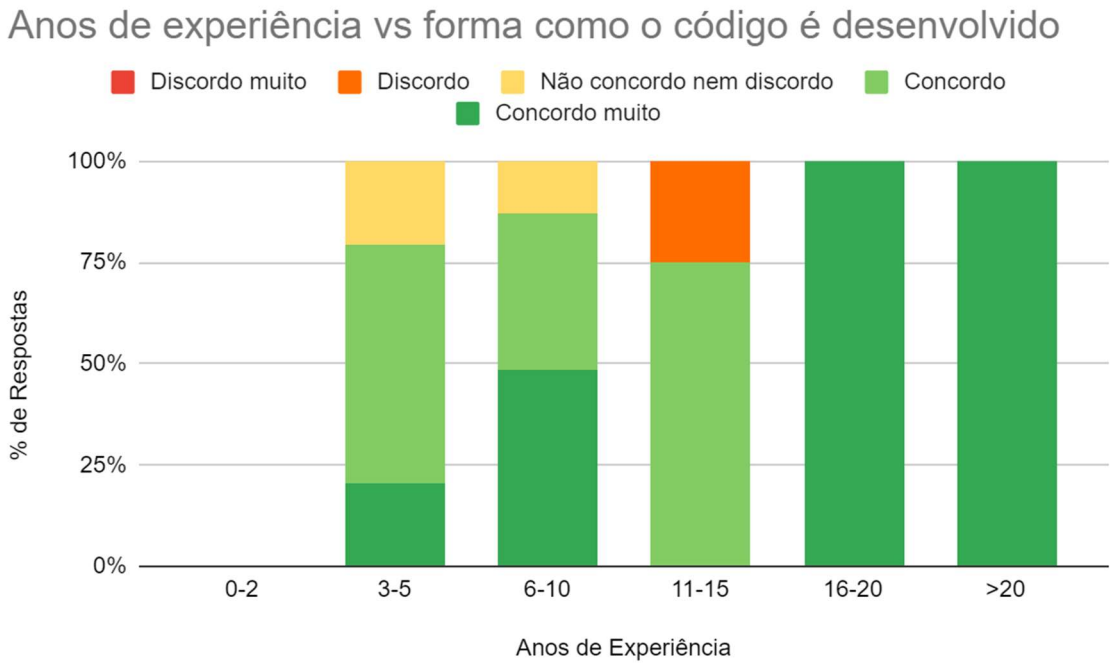

Figura 99 - Anos de experiência vs forma como o código é desenvolvido.

De seguida, na Figura 100, foi analisada a relação entre os anos de experiência e a dificuldade na atividade de testes onde é possível verificar que existem pessoas com mais experiência (6-10 e 11-15) que sentem mais vezes dificuldades do que participantes com entre 3 a 5 anos de experiência. Isto pode dever-se ao facto dos testes em aplicações *OutSystems* não serem tanto a cargo dos *developers* pois, esta é uma atividade recente.

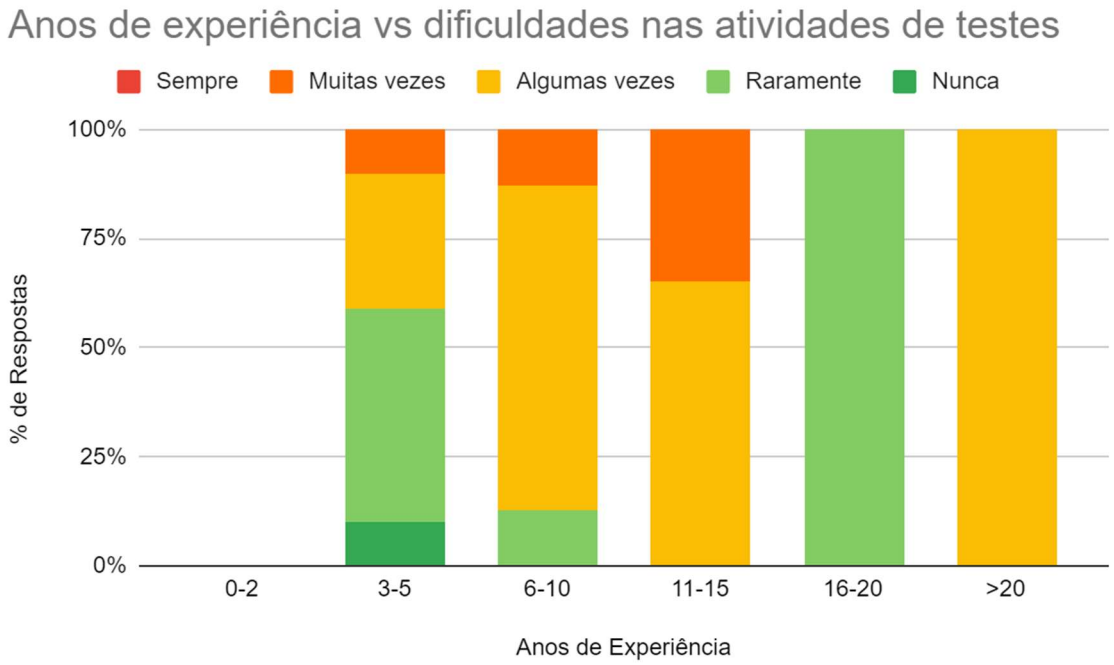

Figura 100 - Anos de experiência vs dificuldades nas atividades de testes.

Por último foi também analisada a relação entre os anos de experiência e modo de escrita dos testes (Figura 101). Neste caso, os dados são bastante semelhantes e verifica-se que muitos participantes ainda utilizam o bom senso como modo de escrita dos testes independente dos anos de experiência. Talvez os participantes devessem ter mais formações nesta área de modo a conhecerem mais técnicas de escrita de testes para melhorarem o seu trabalho.

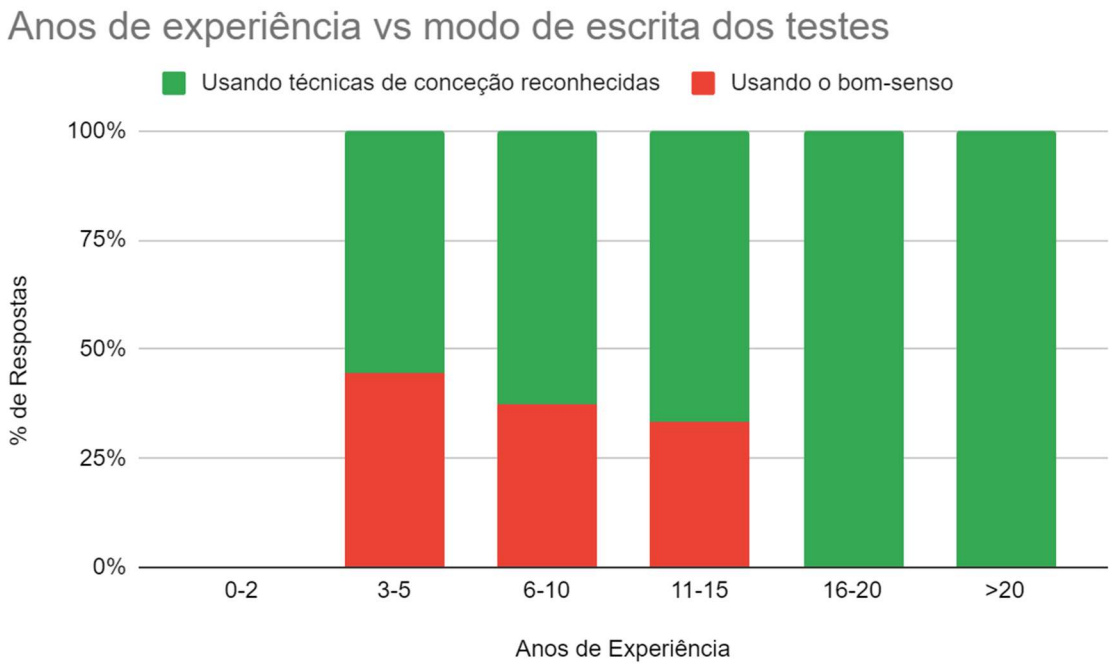

Figura 101 - Anos de experiência vs modo de escrita dos testes

7. Conclusão

Neste trabalho estudaram-se 3 ferramentas de testes de software para realizar testes em aplicações *OutSystems*, foram elas a *BDDFramework*, o *Ghost Inspector* e a *Tricentis Tosca*. A *BDDFramework* é uma ferramenta de automatização de testes desenvolvida em *OutSystems* e, por isso, é aconselhada atualmente para a implementação e desenvolvimento de testes nas aplicações *OutSystems*. O *Ghost Inspector* é uma ferramenta de automatização de testes de software e manutenção de sites, que visa verificar problemas num site ou numa aplicação. A última ferramenta, a *Tricentis Tosca* permite também executar testes de software e é uma ferramenta mais completa visto que permite executar vários tipos de testes.

Foram também estudadas as boas práticas que podem ser aplicadas durante o desenvolvimento das aplicações *OutSystems* de modo a facilitar a implementação e execução dos testes pelos *testers* ou *developers*. Foram identificados 3 tópicos no âmbito das boas práticas a aplicar. O isolamento do domínio que visa separar a lógica de negócio tendo em conta os conceitos em diferentes módulos. O isolamento de APIs que explica que sempre que houver necessidade de consumir um serviço REST/SOAP deve ser através de um módulo *wrapper* que expõe um conjunto de ações publicas que utilizam a API. Por último, a simulação de web UI onde se pretende que os *developers* atribuam nomes aos elementos presentes nos ecrãs de modo a facilitar a sua identificação nos seletores de *JavaScript*.

De modo a avaliar a performance das ferramentas e as implicações das boas práticas, foram implementados 3 cenários de teste a uma aplicação desenvolvida em *OutSystems*. O primeiro cenário correspondia ao *login* de um utilizador na aplicação. Já o segundo cenário demonstrava a adição de um produto ao carrinho de compras e, por fim, o terceiro cenário (teste de API) devolvia a capital do país introduzido pelo utilizador. Com a execução dos cenários de teste nas diferentes ferramentas verificou-se que as ferramentas podem ser todas utilizadas, mas com objetivos diferentes pois, a *BDDFramework* é direcionada para os testes de serviço enquanto o *Ghost Inspector* é específico para testes de UI (testes funcionais). Já a *Tricentis Tosca* é uma ferramenta que permite implementar ambos os níveis de teste. Concluiu-se também que com a *BDDFramework* a implementação dos testes fica a cargo dos *developers* uma vez que é necessário desenvolver ações em *OutSystems* para implementar os testes. Na *Ghost Inspector* e na *Tricentis Tosca* a implementação fica normalmente a cargo dos *testers* uma vez que estes não necessitam de ter contacto com o código da aplicação para a sua implementação. Contudo, para implementar os testes de serviço com a *Tricentis Tosca* os *testers* precisam de possuir a documentação necessária das APIs expostas pelos *developers* de modo a realizar os testes. Para a implementação dos testes com a *BDDFramework*, apesar de ser uma ferramenta fácil de utilizar, é necessário ter conhecimento *OutSystems* até porque é um componente desenvolvido em *OutSystems* e para aplicações *OutSystems*. Tanto o *Ghost Inspector* como a *Tricentis Tosca* não necessitam de conhecimento *OutSystems* mas convém que os *testers* possuam conhecimento de *JavaScript* de modo a otimizar os seletores e scripts de testes. O facto

de ser uma ferramenta mais completa faz com que a *Tricentis Tosca* seja um pouco mais difícil de utilizar, contudo, existem cursos *online* que permitem a sua aprendizagem. Por último, o preço das ferramentas também é um ponto importante no estudo das ferramentas visto que os clientes também tendem a preferir a utilização de ferramentas que não acrescentem valor ao produto. Neste ponto de vista apenas a *BDDFramework* é uma ferramenta de uso gratuito enquanto as outras duas apesar de possuírem períodos de experimentação gratuitos possuem valores associados após esses períodos.

Para compreender a opinião dos profissionais da área das Tecnologias de Informação sobre a importância dos testes de software e sobre a perceção que estes têm sobre a importância das boas práticas de desenvolvimento e a sua influência no processo de automatização de testes, foi realizado um inquérito. O inquérito dividia-se em duas secções e no total foram obtidas 27 respostas. Todos os respondentes trabalham com *OutSystems*, possuem alguma experiência com atividades de testes e utilizam sempre a *BDDFramework* como ferramenta de implementação dos testes. Apesar de ser a ferramenta mais utilizada pelos participantes e de ser de fácil utilização esta, na opinião dos participantes, possui mais pontos fracos do que fortes. Todos reconhecem a importância dos testes independentemente do tipo de aplicação a desenvolver e mais de 50% reconhecem ter frequentemente alguma dificuldade em avaliar o que deve, e como deve, ser testado. Também reconhecem a influência que a forma como as funcionalidades são descritas e a forma como o software é implementado têm sobre o processo da atividade de teste.

Com o facto da ferramenta atualmente utilizada, *BDDFramework*, sugere-se como trabalho futuro a análise e desenvolvimento de melhorias a aplicar nesta ferramenta bem como novas funcionalidades. Acrescentar ou desenvolver uma nova ferramenta que permitisse a implementação de testes de UI em aplicações *OutSystems* seria uma grande vantagem para a plataforma pois, como foi mencionado anteriormente, ambas as ferramentas que permitem este tipo de testes que foram analisadas possuem um custo elevado de utilização.

Bibliografia

[1] OutSystems, "OutSystems Company." https://www.outsystems.com/company/ (accessed Jan. 10, 2020).

[2] J. Proença, "BDD Framework," 2016. https://www.outsystems.com/forge/component-overview/1201/bddframework (accessed Jan. 05, 2020).

[3] E. van; G. D. Black, Rex; Veenendaal, *Programa de Certificação de Testador (Tester) de Nível Foundation*, 3rd ed. 2011.

[4] Devmedia, "O que é a Qualidade de Software," *Qualidade de Software*, 2018. https://www.devmedia.com.br/qualidade-de-software/9408 (accessed Nov. 01, 2019).

[5] Mountain Goat Software., "What is a user story?," *User Stories*, 2019. https://www.mountaingoatsoftware.com/agile/user-stories (accessed Nov. 01, 2019).

[6] H.-W. Jung, "Validating the external quality subcharacteristics of software products according to ISO/IEC 9126," *Comput. Stand. Interfaces*, vol. 29, no. 6, pp. 653–661, 2007, doi: 10.1016/j.csi.2007.03.004.

[7] J. A. Whittaker, "What is software testing? And why is it so hard?," *IEEE Software, Software, IEEE, IEEE Softw.*, vol. 17, no. 1, pp. 70–79, 2000, doi: 10.1109/52.819971.

[8] R. Reinitz and C. Vo, "Follow agile principles," *IBM*, 2020. https://www.ibm.com/garage/method/practices/culture/practice_agile_principles/ (accessed Feb. 25, 2020).

[9] A. Crawford, "DevOps," *IBM*, 2019. https://www.ibm.com/cloud/learn/devops-a-complete-guide (accessed Apr. 03, 2020).

[10] OutSystems, "State of Application Development Report," 2019. [Online]. Available: https://www.outsystems.com/1/state-app-development-insurance/.

[11] Marqual IT Solutions Pvt. Ltd (KBV Research), "Global Low-Code Development Platform Market By Component By Application By Deployment Type By End User By Region, Industry Analysis and Forecast, 2020 - 2026," Report, 2020.

[12] OutSystems, "Low-Code Development Platforms," 2019. https://www.outsystems.com/blog/posts/best-low-code-development-platforms/ (accessed Feb. 05, 2021).

[13] C. Boulton, "What is low-code development? A Lego-like approach to building software," *CIO (13284045)*, 2019. .

[14] J. Idle, "Low-Code rapid application development - So, what's it all about?," *Platinum Business Magazine*, pp. 52–53, 2016.

[15] Appian, "Low-Code Benefits." https://appian.com/platform/low-code-development/low-code-application-development.html (accessed May 04, 2020).

[16] Creatio, "Low code platforms," 2021. https://www.creatio.com/our-

technologies/low-code (accessed Jul. 20, 2021).

[17] G2, "Compare Enterprise Business Low-Code Development Platforms Software," 2021. https://www.g2.com/categories/low-code-development-platforms (accessed Jul. 20, 2021).

[18] AnAr Solutions, "Top 6 Low Code Development Platforms," 2020. https://www.linkedin.com/pulse/top-6-low-code-development-platforms-anar-solutions/ (accessed May 20, 2020).

[19] Software Testing Help, "10 Best Low-Code Development Platforms In 2020," 2020. .

[20] G2, "Compare Low-Code Development Platforms," 2021. https://www.g2.com/categories/low-code-development-platforms (accessed Aug. 20, 2020).

[21] P. Vincent *et al.*, "Gartner Magic Quadrant for Enterprise Low-Code Application Platforms," 2020.

[22] OutSystems, "OutSystems Platform," 2020. https://www.outsystems.com/platform/ (accessed Jan. 10, 2020).

[23] S. Sharma, "OutSystems — An Emerging Low-Code Platform for Rapid Digital Transformation," 2018. https://medium.com/@seema.sharma_51491/outsystems-an-emerging-low-code-platform-for-rapid-digital-transformation-6713bd68945e (accessed Nov. 22, 2020).

[24] S. Hammond and D. Umphress, *Test driven development: the state of the practice*. Tuscaloosa, Alabama, 2012.

[25] N. Singh, "Introduction to Acceptance Test Driven Development (ATDD)," *AgileMania*, 2017. https://agilemania.com/blog/acceptance-test-driven-development-atdd/.

[26] M. Kart, "Behavior-driven development: conference tutorial," *J. Comput. Sci. Coll.*, 2012.

[27] R. Black and G. Coleman, *Agile Testing Foundations : An ISTQB Foundation Level Agile Tester Guide*. 2017.

[28] International Qualification Board for Business Analysis, *Certified Tester Specialist Syllabus Foundation Level Acceptance Testing*. 2019.

[29] J. Klemm, "Ghost Inspector - Automated Website Testing and Monitoring." https://ghostinspector.com/docs/getting-started/ (accessed Nov. 20, 2020).

[30] Tricentis, "Tricentis Tosca." https://www.tricentis.com/products/automate-continuous-testing-tosca/ (accessed Dec. 03, 2020).

[31] J. Proença, "How to Automate BDD Testing in OutSystems, Part 1: An Introduction to the BDDFramework," 2019. https://www.outsystems.com/blog/posts/intro-bddframework-testing/ (accessed Jan. 16, 2020).

[32] Agile Alliance, "Behavior Driven Development (BDD)," 2020. https://www.agilealliance.org/glossary/bdd (accessed Sep. 02, 2020).

[33] M. Homann, "What is Selenium Testing?" https://www.leapwork.com/blog/what-is-selenium-testing (accessed May 03, 2020).

[34] Brian van den Brink, "https://www.valori.nl/en/." https://www.valori.nl/en/ (accessed Mar. 14, 2021).

[35] Tricentis and Valori, "Valori Accelerator for OutSystems," 2021. https://www.tricentis.com/marketplace/valori-accelerator-for-outsystems/ (accessed Mar. 14, 2021).

[36] OutSystems Experts, "The Architecture Canvas," *OutSystems*, 2021. https://success.outsystems.com/Support/Enterprise_Customers/Maintenance_and_Operations/Designing_the_Architecture_of_Your_OutSystems_Applications/The_Architecture_Canvas (accessed Feb. 02, 2020).

[37] Ec. Sample, "The Wine Club," 2014. https://joanasalgueiro.outsystemscloud.com/eCommerce/HomePage.aspx (accessed Jun. 10, 2021).

[38] M. Contributors, "Introduction to the DOM," 2019. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (accessed Jan. 30, 2020).

[39] J. B. Rodrigues, J. F. Gomes, and J. M. Rocha, "Best Practices Test Automation," 2019.