Instituto Superior de Engenharia do Porto

AUTOMATIZAÇÃO DE TESTES À FRAMEWORK myMIS

Diogo Filipe Vieira de Almeida Teixeira

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Arquiteturas, Sistemas e Redes

Orientador: Paulo Alexandre Gandra de Sousa

Júri:

Presidente:

José António Reis Tavares, Doutor, ISEP

Vogais:

Alexandre Manuel Tavares Bragança, Doutor, ISEP

Paulo Alexandre Gandra de Sousa, Doutor, ISEP

Porto, Outubro 2014

Dedicatória

Dedico este trabalho à minha família, que me apoiou durante todo o meu percurso escolar e profissional, sempre na procura do melhor para mim.

Resumo

Atualmente, verifica-se um aumento na necessidade de software feito à medida do cliente, que se consiga adaptar de forma rápida as constantes mudanças da sua área de negócio. Cada cliente tem os seus problemas concretos que precisa de resolver, não lhe sendo muitas vezes possível dispensar uma elevada quantidade de recursos para atingir os fins pretendidos. De forma a dar resposta a estes problemas surgiram várias arquiteturas e metodologias de desenvolvimento de software, que permitem o desenvolvimento ágil de aplicações altamente configuráveis, que podem ser personalizadas por qualquer utilizador das mesmas.

Este dinamismo, trazido para as aplicações sobre a forma de modelos que são personalizados pelos utilizadores e interpretados por uma plataforma genérica, cria maiores desafios no momento de realizar testes, visto existir um número de variáveis consideravelmente maior que numa aplicação com uma arquitetura tradicional. É necessário, em todos os momentos, garantir a integridade de todos os modelos, bem como da plataforma responsável pela sua interpretação, sem ser necessário o desenvolvimento constante de aplicações para suportar os testes sobre os diferentes modelos.

Esta tese debruça-se sobre uma aplicação, a plataforma myMIS, que permite a interpretação de modelos orientados à gestão, escritos numa linguagem específica de domínio, sendo realizada a avaliação do estado atual e definida uma proposta de práticas de testes a aplicar no desenvolvimento da mesma.

A proposta resultante desta tese permitiu verificar que, apesar das dificuldades inerentes à arquitetura da aplicação, o desenvolvimento de testes de uma forma genérica é possível, podendo as mesmas lógicas ser utilizadas para o teste de diversos modelos distintos.

Palavras-chave: Desenvolvimento Orientado a Modelos, Testes genéricos a uma aplicação, Desenvolvimento Ágil

Abstract

Currently the need for software developed according to the Customer specification that can easily adapt to the constant changes in its business area. Each customer has its specific needs, and currently he just cannot afford the expense of the resources needed to achieve the best solution. In order to answer these problems many software development methods and architectures have appeared, allowing the development in a agile way of highly configurable applications, that can be personalized by any of its users.

This dynamism, brought to the applications in the form of models that are customized by its users and interpreted by a generic platform, creates bigger challenges in the moment of developing tests, because there is a much bigger number of variables than in a application with a traditional architecture. It is necessary, in every moment, to guarantee the integrity of all models, as well as the integrity of the platform that is responsible for their interpretation, without a constant need of developing applications to test each model separately.

This thesis focus on a specific application, the platform myMIS, that allows the interpretation of models oriented to business management that are written in a domain specific language, being evaluated the current state of the application and defined a proposal for test practices to apply in the development of this platform.

The resulting proposal of this thesis allowed to conclude that, despite the difficulties caused by the architecture of the application, the development of tests in a generic form is possible, being the same logics applied in the test of many different models.

Keywords: Model Driven Development, Generic application tests, Agile Development

Agradecimentos

Agradeço a Carlos Morais, pela dedicação e ajuda prestada ao longo deste processo.

Agradeço ao meu orientador, Paulo Gandra de Sousa, pela sua colaboração e transmissão de conhecimentos essencial no desenvolvimento desta tese.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

REA	*Resources, Events, Agents*
BAMoL	*Business Application Modeling Language*
TDD	*Test-Driven Development*
ATDD	*Acceptance Test-Driven Development*
BDD	Behavior-*Driven Development*
UI	*User Interface*
API	*Application Programming Interface*
DSL	*Domain Specific Language*
SAF-T	*Standard Audit File for Tax Purposes*
DOM	*Document Object Model*
IDE	*Integrated Development Environment*
XP	*Extreme Programming*
REST	*Representational state transfer*
CRUD	*Create, Read, Update and Delete*
HTTP	*Hypertext Transfer Protocol*
JSON	*JavaScript Object Notation*
AOM	*Adaptive Object Model*

1. Introdução

A procura constante pelo aumento da eficiência com que são executadas as tarefas do dia-a-dia, tanto na vida pessoal como profissional, aumentou a necessidade de produzir software de forma mais rápida, ao mesmo tempo que se espera que o mesmo seja mais eficiente. De forma a garantir a qualidade do produto desenvolvido, é de extrema importância que, desde uma fase inicial o desenvolvimento de uma aplicação siga a metodologia de desenvolvimento de software mais adequada ao problema, sem descurar partes importantes como testes constantes, de forma a garantir a qualidade e documentação, que permite aumentar a eficiência do desenvolvimento do produto.

1.1. O Problema

Concretamente, o problema apresentado nesta tese consiste na necessidade de implementar práticas de testes formais no desenvolvimento de uma aplicação. Esta necessidade foi identificada após ser verificado que as práticas de testes aplicadas pela equipa de desenvolvimento não eram suficientes para testar convenientemente uma aplicação em crescimento constante.

A aplicação em causa é a aplicação myMIS, uma plataforma orientada ao desenvolvimento de sistemas de gestão. A aplicação foi desenvolvida segundo uma arquitetura orientada a modelos, permitindo que os mais variados comportamentos e funcionalidades da aplicação sejam descritos nos modelos, em vez de serem descritos no código. Este estilo de arquitetura permite que a plataforma seja capaz de interpretar modelos com características bastante distintas e orientadas as necessidades de cada cliente, sendo que cada cliente poderá ter uma solução feita à sua medida.

Os modelos são todos desenvolvidos segundo uma linguagem específica de domínio, denominada BAMoL. Esta linguagem foi desenvolvida com base nos princípios do modelo REA (REA Technology - Technology That Understands Your Business, 2007), também este orientado à descrição de sistemas de gestão/contabilidade.

As especificidades desta plataforma trazem bastantes desafios ao desenvolvimento de testes. Fruto da orientação desta aplicação a modelos, é expectável que a mesma seja capaz de interpretar uma grande quantidade de modelos distintos. Do ponto de vista dos testes, esta arquitetura traz uma complexidade acrescida, trazida pela necessidade de os mesmos serem capazes não só de validar as alterações efetuadas na plataforma, mas também a validade das alterações perante os modelos e a validade dos modelos perante a plataforma. Toda a lógica de testes deverá ter como principal objetivo a validação da integridade de todos os modelos da plataforma, escritos, conforme mencionado anteriormente, numa DSL específica, sendo esperado que o desenvolvimento e a execução dos testes sigam a metodologia de desenvolvimento de software adotada pela empresa, o desenvolvimento ágil.

1.2. Objetivos

Os objetivos deste trabalho de investigação podem ser enumerados da seguinte forma:

- Exploração das metodologias de desenvolvimento de software mais relevantes à data, de forma a identificar qual das mesmas se adequa a uma aplicação em específico, a plataforma myMIS;

- Elaboração de uma proposta válida de um módulo de testes que possa vir, de forma gradual, a ser integrado no ciclo de desenvolvimento de software e nas diversas iterações sobre os modelos, permitindo este validar se os modelos são válidos perante a framework existente e/ou se alterações na framework não danificaram a sua integridade;

- Proposta de possíveis formas de integração do desenvolvimento e execução dos testes nas metodologias de desenvolvimento de software e dos modelos em prática na empresa.

1.3. Proposta de Solução

A proposta de solução encontrada passa pela definição de uma metodologia de desenvolvimento a aplicar no desenvolvimento da aplicação, tendo sido escolhida o Behaviour Driven Development (BDD).

Após a definição da metodologia a aplicar, foram avaliadas as alterações e evoluções à plataforma necessárias para a implementação dos princípios desta metodologia de desenvolvimento de software, sendo a mais relevante o desenvolvimento de testes e documentação numa DSL específica, o Gherkin (Cucumber, 2014). Desta forma, foi desenvolvida uma proposta para um módulo de testes, que permite o desenvolvimento e a execução dos testes sobre vários modelos distintos. Estes testes são executados sobre a

plataforma myMIS, nomeadamente sobre uma aplicação cliente, que atua como uma camada de interface com o utilizador, mas também sobre a camada de API da plataforma. Devido às especificidades destes dois ambientes, a proposta apresentada envolveu o desenvolvimento de lógicas de processamento de testes distintas para o cliente de UI e para a API.

Por fim, a proposta de solução é complementada com a definição de uma lógica de integração do desenvolvimento e da execução dos testes no ciclo de desenvolvimento de software e dos modelos. Esta lógica segue os princípios da metodologia de desenvolvimento Continuous Integration, na qual as atualizações à aplicação efetuadas por developers são integradas na solução de forma constante, e disponibilizadas aos clientes ao ritmo que são integradas. No entanto, a aplicação terá de passar por um ciclo de testes antes de ser disponibilizada aos clientes, de forma a validar se a mesma cumpre com os requisitos definidos.

1.4. Estrutura do documento

Este documento é composto por 8 capítulos e um anexo. Os capítulos encontram-se organizados da seguinte forma:

1. Introdução: Neste capítulo é feita uma apresentação do problema que deu origem a esta tese. São, em seguida, enumerados os objetivos que deverão ser atingidos no fim da tese, bem como realizada uma breve descrição da solução encontrada;
2. Contexto: Neste capítulo é feita uma introdução dos temas relevantes para a melhor compreensão do problema e da solução encontrada;
3. Estado da Arte: Neste capítulo são enumeradas as metodologias e ferramentas mais importantes no contexto desta tese, sendo descrita a escolha de quais serão usadas na proposta de solução;
4. Proposta de solução: Neste capítulo é feita a descrição da implementação da solução encontrada;
5. Testes a executar: Neste capítulo é feita a descrição dos testes que vão ser aplicados para validar se a solução encontrada permite cumprir com os objetivos definidos inicialmente;
6. Resultados dos testes: Neste capítulo é feita a análise dos resultados dos testes efetuados à solução encontrada;
7. Conclusões: Neste capítulo é feito um resumo do trabalho realizado e da solução encontrada, sendo descritas as limitações e o trabalho futuro;
8. Referências: Neste capítulo são mencionadas as referências utilizadas neste relatório.

Foi decidido organizar parte da informação do relatório num anexo que complementa a informação inserida nos capítulos.

- Anexo 1- Frases suportadas: Este anexo contém as frases suportadas nos testes para esta proposta de solução.

2. Contexto

2.1. Plataforma myMIS

2.1.1. Introdução

O myMIS é uma solução para desenvolvimento e deployment ágeis de Sistemas de Informação de Gestão. Neste momento existem dois pontos distintos do sistema: a plataforma e a aplicação web para o utilizador final. Nesta tese será dado destaque a estas duas componentes.

A plataforma myMIS é uma framework sustentada por uma abordagem baseada em modelos. Esta abordagem permite retirar a lógica de negócio da plataforma, sendo esta implementada nos modelos. Isto faz com que a aplicação permita dar resposta a um maior número de problemas apresentados, sem a necessidade de alterações na plataforma, o que reduz os tempos de resposta aos problemas apresentados, bem como a possibilidade de ocorrência de instabilidade provocada por alterações frequentes à plataforma. Como cada cliente desta plataforma terá problemas e necessidades diferentes, isso fará com que existam múltiplos modelos no sistema. Para ser possível responder a todas estas soluções é necessário que estes respeitem uma determinada formalidade, que é obtida através da utilização de uma linguagem específica de domínio.

De forma a cada cliente poder ter a sua solução personalizada, a plataforma myMIS aplica o princípio de arquitetura *Multi-Tenancy.* Este princípio define que, de forma a permitir a poupança de recursos e uma maior escalabilidade, os vários clientes (denominados de *tenants*) devem partilhar a mesma aplicação e base de dados (Bezemer & Zaidman, 2010). Desta forma, é possível fornecer aplicações com elevada capacidade de configuração, sem os gastos inerentes à manutenção de várias bases de dados distintas. Este princípio encaixa na arquitetura baseada em modelos da plataforma myMIS, permitindo a partilha da aplicação e do *schema* da base de dados, sendo apenas distinto entre clientes o conteúdo da última

(modelo e dados). De forma a melhor perceber a arquitetura da aplicação, nas secções 2.1.4 e 2.1.5. vai ser realizada a descrição do desenvolvimento de software baseado em modelos, bem como dos princípios do AOM, seguidos na definição da arquitetura da aplicação myMIS.

A linguagem interpretada pela plataforma é a BAMoL (Business Application Modeling Language), uma DSL orientada a aplicações de gestão. Esta é interpretada pelo runtime da plataforma myMIS e que tem como objetivo permitir descrever o mais variado tipo de soluções, através dos já referidos modelos. Esta linguagem baseia-se na framework REA, estendendo os seus conceitos base, e será realizada uma descrição de ambas nas secções 2.1.2. e 2.1.3.

Resumindo, o funcionamento básico da plataforma myMIS pode ser descrito da seguinte forma: a plataforma interpreta a linguagem BAMoL no formato de modelos, em que cada um representa uma solução para um determinado problema. Por sua vez, a BAMoL baseia-se na framework REA, permitindo assim que esta linguagem seja capaz de descrever um variado número de Sistemas de Informação de Gestão.

De forma a desenvolver e editar os modelos, a plataforma myMIS conta com uma ferramenta que permite a modelação dos vários modelos existentes em runtime, sendo as alterações aplicadas de imediato. Esta aplicação, em desenvolvimento constante, permite que os utilizadores editem o modelo de uma forma rápida através do navegador web, sendo no entanto necessários conhecimentos da DSL BAMoL para conseguir usufruir das potencialidades da plataforma.

O desenvolvimento dos modelos descritos segue as orientações dos stockholders “clientes” da aplicação, que descrevem como pretendem a sua aplicação (modelo). Com estas indicações, são identificadas features a adicionar à aplicação, sendo este processo iniciado pela descrição das mesmas à equipa de modelação, que avalia as alterações necessárias ao modelo e, eventualmente, a necessidade de efetuar alterações à framework.

Caso a feature não seja suportada pela plataforma, o desenvolvimento da mesma irá necessitar da intervenção de vários elementos da equipa do myMIS, sendo neste caso, necessária a intervenção da equipa de desenvolvimento de software, e, numa fase posterior, a intervenção do modelador.

No caso da intervenção da equipa de desenvolvimento, esta só é necessária quando a plataforma não suporta as alterações necessárias no modelo. Quando isto acontece, são identificadas as user stories a desenvolver. Após a conclusão das alterações à framework, o modelador realiza as alterações necessárias ao modelo, sendo posteriormente efetuados testes pelo mesmo de forma a verificar se as alterações à framework e/ou ao modelo têm o comportamento esperado.

É de realçar que a execução dos testes também deverá ser executada a cada alteração efetuada na framework, como por exemplo no caso da correção de bugs ou outras otimizações.

Sempre que existam alterações à framework deverão ser executados os testes para todos os modelos existentes.

Desta forma, o relacionamento entre as diversas partes interessadas e a equipa de desenvolvimento traduz-se no seguinte fluxograma:

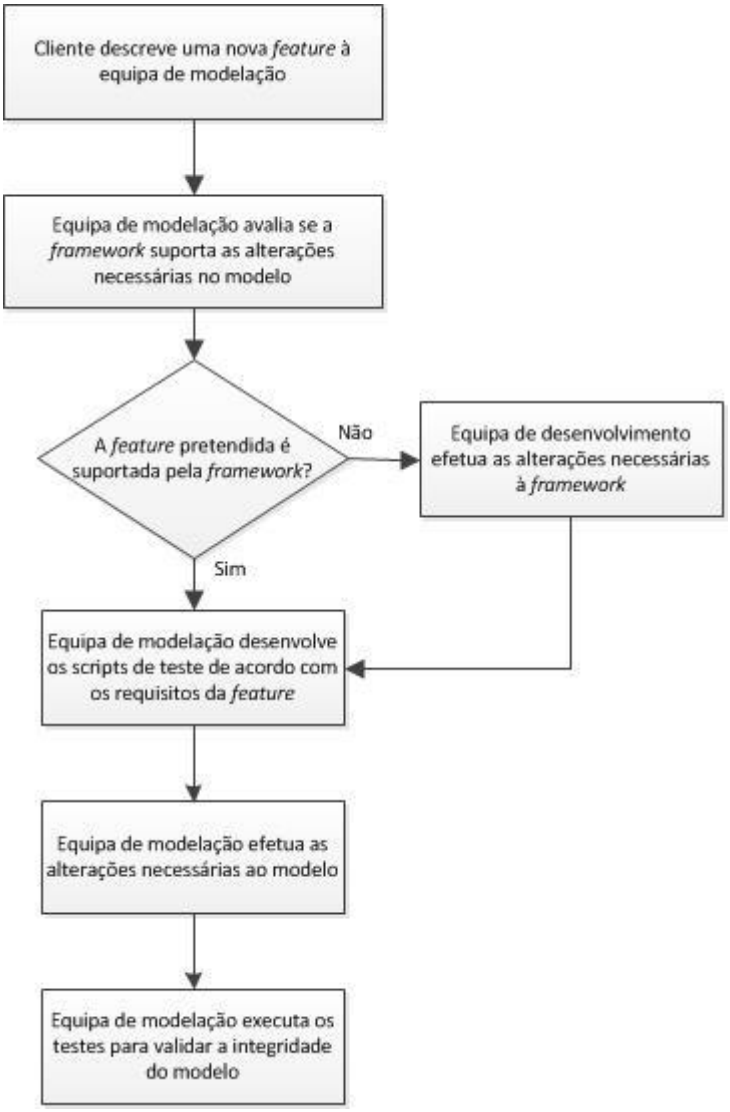

Figura 1 – Processo de resolução de uma *feature*

Por fim, é importante destacar que a plataforma poderá ser complementada com diversas aplicações clientes, de forma a permitir a sua extensão. Estes clientes, que podem ser de vários tipos distintos, funcionam de forma independente à aplicação, sendo à data desta tese o cliente mais relevante a interface com o utilizador.

Todas as aplicações clientes têm de estar registadas e autorizadas perante a plataforma, sendo a comunicação das mesmas com a plataforma realizada através da camada de API da mesma. Esta é do tipo RESTful, que, conforme o nome indica, segue os princípios de arquitetura REST na sua implementação. Esta caracteriza-se pela disponibilização de serviços que permitem executar as quatro operações CRUD através de pedidos HTTP com os métodos base (Get, Post, Put e Delete). A informação transmitida nestes pedidos utiliza o formato JSON (exceto para os casos nos quais são transmitidos ficheiros), sendo o retorno de informação para a aplicação cliente também neste formato. Nos casos em que, por algum motivo o pedido resulta num erro, a informação adicional do erro é também devolvida no formato JSON.

Todos os pedidos HTTP efetuados à API têm de passar pelo mecanismo de autorização OAuth. Este caracteriza-se por permitir acesso a aplicações de terceiros a recursos que se encontram num serviço diferente, através da emissão de tokens de acesso com duração limitada. Os tokens, após serem emitidos, devem ser utilizados pela aplicação cliente para aceder aos recursos do serviço, devendo esta efetuar a gestão dos mesmos e utilizá-los em todos os pedidos efetuados à API.

2.1.2. REA

A framework REA (Resources, Events, Agents) é orientada ao desenvolvimento de sistemas de contabilidade, centrando-se no registo dos diversos eventos económicos de uma organização. Esta assume que existe um número limitado de conceitos presentes em todo o software de contabilidade, e que é possível desenhar uma aplicação facilmente adaptável, sem por em causa o cumprimento das regras de negócio (REA Technology - Technology That Understands Your Business, 2007).
Os conceitos base desta framework são os seguintes: Recursos Económicos, Agentes Económicos, Eventos Económicos, Compromissos e Contratos, sendo que, de acordo com os princípios do REA, com estes conceitos é possível descrever um elevado número de eventos económicos que ocorrem no dia-a-dia de uma empresa. Segue-se uma descrição de cada um destes conceitos e o seu relacionamento na descrição de eventos económicos.

- Recursos Económicos: Entidades do sistema que representam bens com valor comercial e quantidades possíveis de definir. Estes bens estão na posse de Agentes Económicos, sendo trocados em transações financeiras entre dois agentes diferentes. Recursos económicos são, por exemplo, Produtos, Dinheiro e Serviços.

- Agentes Económicos: Entidades do sistema que representam indivíduos ou conjuntos de indivíduos (p.ex.: uma empresa). Estas entidades possuem recursos económicos, e efetuam troca dos mesmos em transações financeiras. Agentes económicos são, por exemplo, Clientes, Empresas e Colaboradores.

- ➔ Compromissos: Como o nome sugere, representa o compromisso assumido entre dois agentes de efetuar uma transação de recursos. O Compromisso é assumido visto, por muitas vezes, a transação não se realizar no momento em que é definida, mas sim num momento posterior. Um bom exemplo é o pagamento de bens adquiridos, que por muitas vezes não é pago a pronto, sendo dado um prazo de pagamento ao comprador, que se compromete a pagar findo esse prazo.

Eventualmente todos os Compromissos darão origem a eventos económicos, que ocorre no momento em que a troca de recursos se efetua. Esta operação de transformação denomina-se de *fulfillment,* e na mesma ocorre um cumprimento do Compromisso, realizado pelo novo evento. Este cumprimento pode ser total ou parcial. É importante realçar que os Compromissos podem ser de dois tipos: incremento ou decremento, mediante representarem uma diminuição ou um aumento nos recursos da empresa.

- ➔ Eventos Económicos: Representa a transação de recursos entre dois agentes, atuando um como fornecedor e outro como recetor. Desta forma, esta transação representa um incremento para o agente recetor do recurso e um decremento para o agente que o fornece. Excetuando alguns casos, como a prestação de serviços, um Evento ocorre no momento em que foi criado, sendo o seu início e fim muito próximos.

Os Eventos, tal como os Compromissos, podem ser de dois tipos: incremento ou decremento, mediante representarem uma diminuição ou um aumento nos recursos da empresa.

- ➔ Contrato: Composto por um conjunto de Compromissos e de regras, representa o que pode acontecer durante uma transação de recursos. As regras especificam o que pode acontecer caso determinadas condições não sejam cumpridas durante a transação, podendo gerar compromissos adicionais. Por exemplo, no âmbito de uma Venda, um contrato é composto por um conjunto de compromissos que representam a troca dos bens por dinheiro, e que podem conter regras que preveem a cobrança de juros caso os artigos não sejam pagos num prazo definido. Isto traduz-se na criação de um novo Compromisso, que representa a dívida dos juros para com a empresa.

Todos estes conceitos relacionam-se para a representação de transações económicas da seguinte forma:

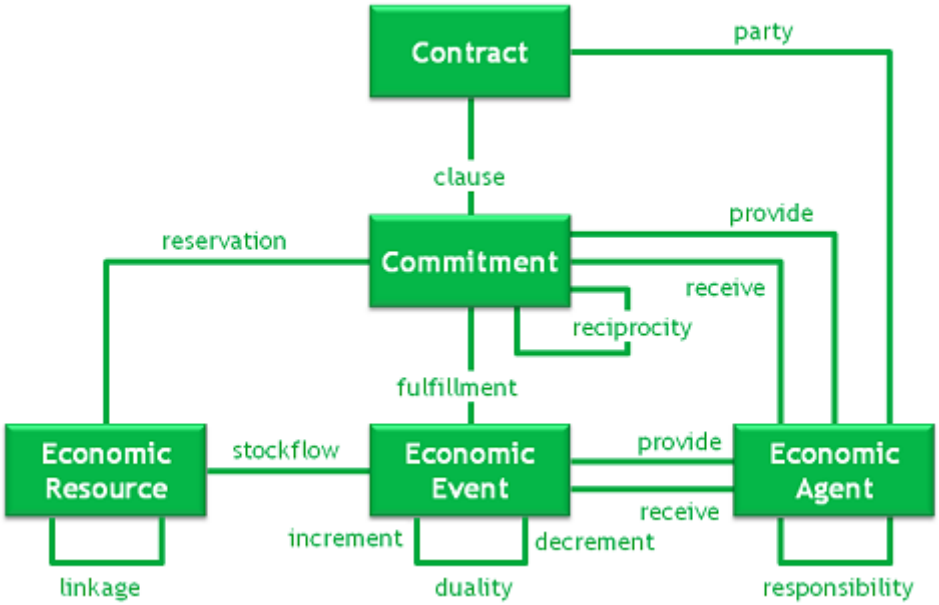

Figura 2 – Modelo REA (REA Technology - Technology That Understands Your Business, 2007)

De forma a exemplificar a relação entre estes conceitos, vamos passar a descrever o caso de uma Venda. Esta é uma transação económica, entre dois agentes (Cliente e Empresa), no qual se trocam dois recursos económicos- Produto e Dinheiro.

De forma a relacionar os agentes e os recursos, precisamos de um Evento Económico e de um Compromisso. O evento, do tipo decremento, é a Venda em si, na qual se representa a troca do produto (recurso) entre o agente fornecedor (empresa) e o recetor (cliente). Por sua vez, o Compromisso (do tipo incremento) representa a Promessa de Pagamento dos produtos, sendo trocado um recurso (dinheiro) entre o Cliente (fornecedor) e a Empresa (recetor). Por fim encontra-se a definição do Contrato, no qual é agrupado o Compromisso de promessa de pagamento, bem como eventuais regras, condições e compromissos resultantes do incumprimento das mesmas.

O exemplo anterior representa um processo de troca (Exchange) de recursos. No entanto, a movimentação de recursos de uma empresa sem sempre ocorre devido à troca de uns recursos por outros, como no caso de fábricas que transformam matéria-prima em produto final. De forma a dar resposta a estes casos o REA suporta, além do processo de troca, o processo de transformação (Conversion). Neste é assumido que o decremento de recursos da empresa pode ocorrer sem a intervenção de um segundo agente, sendo o mesmo compensado por um incremento de outro recurso que ficará sobre o controlo da empresa.

No momento de definição de uma aplicação segundo os princípios do REA, existem algumas regras cuja aplicação pode ser vista no exemplo anterior, e que têm de ser cumpridas:

- Por cada Incremento tem de existir um decremento;

- Cada Evento ou Compromisso tem de conter um agente fornecedor e um agente recetor, bem como um recurso;

2.1.3. Linguagem BAMoL

Os modelos interpretados pela framework myMIS são, conforme descrito anteriormente, escritos na linguagem BAMoL. De forma a melhor perceber a descrição dos modelos que são objeto de testes, vai ser feita nesta secção uma descrição desta linguagem.

O BAMoL baseia-se na framework REA, utilizando a generalidade dos seus conceitos, tendo também sido feitas extensões da mesma. Os conceitos base são os seguintes:

- ➔ Recursos Económicos: Entidades do sistema que representam bens com valor comercial e quantidades possíveis de definir. Estes bens podem ser trocados em transações financeiras entre diferentes indivíduos.

- ➔ Agentes Económicos: Entidades do sistema que representam indivíduos ou conjuntos de indivíduos (p.ex.: uma empresa). Estas entidades possuem recursos económicos, e efetuam troca dos mesmos em transações financeiras.

- ➔ Entidades Definidas p/Utilizador: Entidades do sistema necessárias para as transações financeiras, mas que, no entanto, não podem ser considerados recursos ou agentes. Um exemplo de uma entidade deste tipo utilizada para categorizar um agente seria o Departamento ou o Cargo que ocupa na empresa. Este conceito não faz parte da framework REA.

- ➔ Compromissos: Como o nome sugere, representa o compromisso assumido entre dois agentes de efetuar uma transação de recursos. O Compromisso é assumido visto, por muitas vezes, a transação não se realizar no momento em que é definida, mas sim num momento posterior. Um bom exemplo é o pagamento de bens adquiridos, que por muitas vezes não é pago a pronto, sendo dado um prazo de pagamento ao comprador, que se compromete a pagar findo esse prazo.

- ➔ Eventos Económicos: Representa a transação de recursos entre dois agentes. Esta transação representará um incremento para o agente que recebe o recurso e um decremento para o agente que o fornece. Ao contrário dos compromissos, um Evento ocorre no momento em que foi criado, sendo o seu início e fim muito próximos.

- ➔ Interação: O conceito de Interação permite representar as mais diversas transações de recursos, sobre a forma de um documento, sendo uma interação é composta por três

partes distintas: Cabeçalho, Detalhes e Resumo, como podemos ver na imagem seguinte:

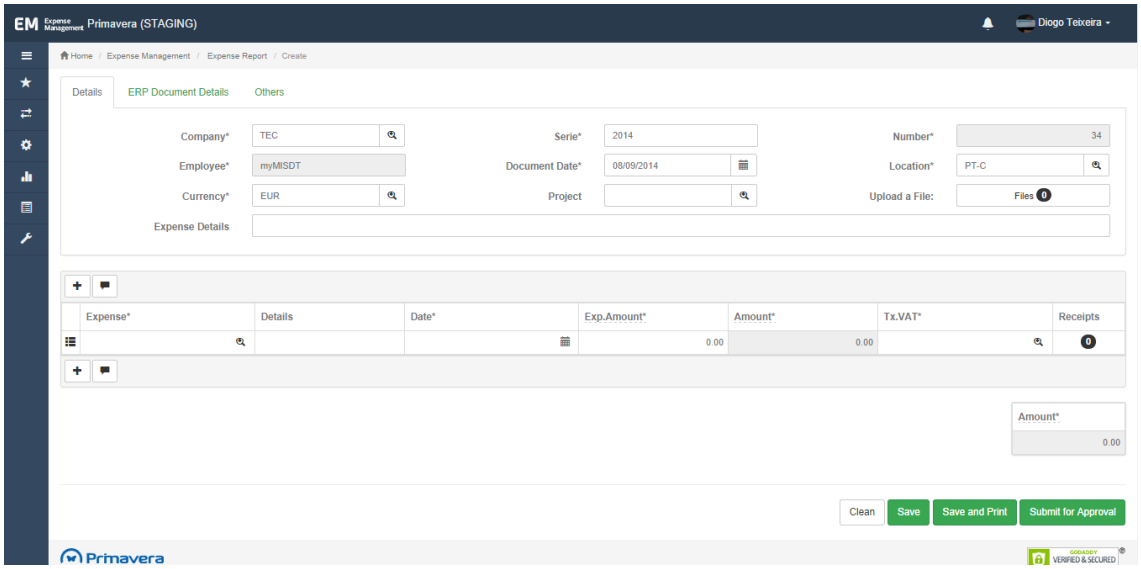

Figura 3 – Representação de uma Interação no cliente de UI

O cabeçalho permite-nos detalhar o contexto da interação, indicando-nos em que data ocorreu, qual a série do documento, entre outras informações importantes divididas por separadores para uma melhor organização, como podemos ver na imagem anterior. Por sua vez, os detalhes (grelha na imagem anterior) e o resumo (canto inferior direito) são compostos por Compromissos e/ou Eventos Económicos relacionados com a Interação em questão, e que permitam a representação de um ou mais eventos económicos. Ainda neste subcapítulo será fornecido um exemplo que demonstrará a relação entre todos estes conceitos. Este conceito é uma extensão à framework REA.

➔ Processo: O conceito de Processo permite agrupar várias interações em processos de negócio. Desta forma, é possível obter uma organização lógica de interações de natureza semelhante. A imagem seguinte demonstra o menu de navegação da aplicação, na qual é possível verificar a organização das interações (Adiantamento Monetário e Relatório de despesas) por processo (Gestão de Despesas):

Figura 4 – Organização dos Processos e Interações

Exemplo

Pretende-se desenvolver um modelo bastante simples, que represente a gestão de vendas de produtos a clientes. De forma a representar esta transação económica, e seguindo os conceitos apreendidos, vamos precisar de criar os seguintes tipos:

- Agentes: Empresa e Cliente;
- Recursos: Produto e Dinheiro;

Após a criação destes tipos, é necessário relacioná-los sobre a forma de uma Interação de forma a representar a Venda. Numa operação de venda são trocados dois recursos (Produto por Dinheiro) entre dois agentes (Empresa e Cliente) devendo, de forma a representar esta troca de recursos, ser criados um Compromisso e um Evento. O Compromisso permite-nos representar a promessa do Cliente pagar os bens à Empresa (é prática comum o Cliente ter um prazo de pagamento), sendo o Evento necessário para representar a saída dos produtos para o Cliente.

No esquema seguinte podemos ver o resumo de todas as Entidades que compõem a interação "Venda". As mesmas deverão ser criadas seguindo o sentido descendente, sendo o Evento Económico "Venda" o último a ser criado.

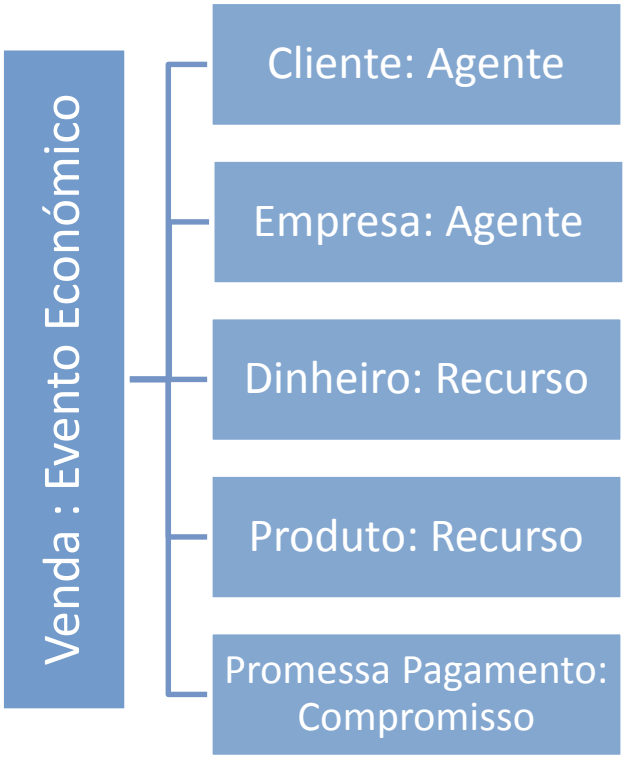

Figura 5 - Representação de um Evento Económico na linguagem BAMoL

Por fim, é importante relevar que, todos os modelos desenvolvidos na DSL BAMoL podem aceder a várias features da plataforma myMIS, sendo as mais relevantes as seguintes:

- Disponibilização de um motor de contabilidade, que permite gerir várias contas associadas às Entidades do modelo, movimentadas quando ocorre um evento económico;
- Acesso à informação dos eventos económicos de forma clara através de painéis de controlo;
- Disponibilização de diferentes perfis de segurança, que condicionam o acesso dos utilizadores a partes do modelo;
- Mecanismo de aprovações de Entidades e Interações completamente configurável;
- Motor de notificações por correio eletrónico;
- Motor de impressões;
- Integração de informação em sistemas externos.

2.1.4. Desenvolvimento Baseado em Modelos

Um modelo, no âmbito do desenvolvimento de software, pode ser descrito como uma forma de representar o funcionamento de um sistema com características e comportamentos bem definidos. Da mesma forma, a escrita de uma nova aplicação também passa pela representação de comportamentos esperados quando nos encontramos perante determinado cenário.

O desenvolvimento de software baseado em modelos permite o desenvolvimento de aplicações tendo em conta modelos com uma estrutura definida antes do início do desenvolvimento da aplicação. Os comportamentos esperados deixam de estar descritos no código da aplicação, para passarem a estar descritos no modelo que é interpretado pela mesma. Ao colocar parte da lógica da aplicação na forma de modelos conseguimos desenvolver aplicações mais flexíveis, com elevada capacidade de reutilização. Esta forma de desenvolvimento obriga a um desenho mais genérico, que permite a interpretação de modelos mais diversificados, que dão resposta a um grande número de problemas sem um aumento exponencial na complexidade do código desenvolvido nem a necessidade de constantes alterações ao mesmo para alterar os comportamentos existentes ou adicionar novos.

Ao desenvolver uma aplicação baseada em modelos, é esperado que esta consiga interpretar vários modelos distintos dentro de um domínio específico, existindo uma forte separação entre o código da aplicação e o domínio. Todos os modelos interpretados pela aplicação deverão ser escritos numa determinada linguagem específica de domínio (DSL) sendo estas, como o nome indica, concebidas para serem utilizadas num domínio de problema em particular. Ou seja, permitem descrever aplicações que são interpretadas pelo domínio a que se referem.

A plataforma é responsável por efetuar a interpretação dos modelos e a transformação nos comportamentos esperados. O desenvolvimento de aplicações desta forma deverá permitir que especialistas com fortes conhecimentos num determinado domínio desenvolvam os modelos sem a necessidade de possuírem quaisquer conhecimentos de programação (Selic, 2003).

Todas estas características do desenvolvimento de software baseado em modelos traduzem-se no desenvolvimento de aplicações de forma mais rápida e mais adequadas aos problemas que cada cliente tem. No entanto, trazem desafios que devem ser previstos e geridos com precaução pelas equipas de desenvolvimento. Estes desafios podem ser divididos em três categorias principais: Modelação da Linguagem, Separação de Conceitos e Gestão dos Modelos (France & Rumpe, 2007).

Os desafios inerentes à primeira categoria passam pela necessidade de disponibilizar aos desenvolvedores dos modelos (modeladores), uma ferramenta que permita esta tarefa, tendo esta que ter em conta a DSL em uso. Esta deverá não só permitir a criação e edição de modelos, mas também possuir lógicas que permitam verificar se um determinado modelo é válido

perante a aplicação. Por sua vez, a Separação de Conceitos reforça a ideia que, permitindo o sistema a modelação de comportamentos em várias áreas distintas da aplicação, pode obrigar à utilização de variações da DSL mediante a área da aplicação. Por fim, a gestão de modelos obriga a ter em conta questões como o controlo de diferentes versões do modelo, identificação de dependências, bem como a necessidade de manter os vários modelos válidos perante a aplicação, tendo em conta as evoluções que esta sofrerá com o tempo.

2.1.5. Adaptive Object Model

Atualmente é possível verificar que existem várias aplicações, nas mais diversas áreas, se destacam pela sua flexibilidade, permitindo ao utilizador um grau de configuração bastante elevado, refletindo-se as suas alterações de forma imediata. Estes sistemas têm uma arquitetura que se caracteriza pela disponibilização do modelo ao utilizador em runtime, de forma a este poder proceder a alterações necessárias ao mesmo (Yoder & Johnson, 2002).

Este estilo de desenho de aplicações denomina-se Adaptive Object Model (AOM). Estes sistemas caracterizam-se pela separação de toda a lógica de negócio do núcleo da aplicação, sendo esta armazenada num formato específico e interpretada em runtime. Esta lógica é armazenada sobre a forma de modelos de objetos, que contêm, por exemplo, a representação de entidades, das suas relações e comportamentos. Por sua vez, a aplicação atua como um interpretador dos modelos, refletindo os comportamentos descritos nos mesmos.

A informação armazenada é designada como *metadata,* e pode ser editada pelos utilizadores, a qualquer momento, para alterar os comportamentos existentes. Após a gravação das alterações, o modelo está pronto para ser interpretado pela aplicação, tornando as mesmas visíveis para todos os utilizadores.

O desenho de aplicações com o estilo AOM diferenciam-se das aplicações object-oriented por não fazerem a definição das várias entidades do sistema de forma explícita, sobre a forma de classes. A definição das mesmas encontra-se na metadata, o que faz com que não sejam necessárias alterações no código da aplicação sempre que se deseje adicionar uma nova entidade ou efetuar alterações nas entidades existentes. De forma a permitir a passagem desta definição na metadata, na arquitetura da aplicação deve ser feita a definição de tipos de entidades, em vez de definir entidades. Estas passam a ser definidas como instâncias do tipo, como podemos ver em seguida:

Neste exemplo temos um sistema com três entidades diferentes: Colaboradores, Fornecedores e Clientes. De acordo com os princípios do REA vistos anteriormente, todos estes são agentes, podendo ser representados de forma simples, com a definição de três classes distintas, que serão por sua vez subclasses da classe Agentes.

Com a lógica do AOM passamos de uma definição de classes estática para uma mais genérica, que nos permite, em runtime e sem a necessidade de alterar o código existente, adicionar novos tipos de entidades e os seus atributos e relações. No caso da definição dos atributos,

esta também existe uma arquitetura específica. Enquanto que numa aplicação com uma arquitetura tradicional os atributos são definidos por cada classe (sendo herdados para as subclasses), no estilo AOM os atributos deverão estar associados aos tipos de entidades criados, como podemos ver no diagrama seguinte:

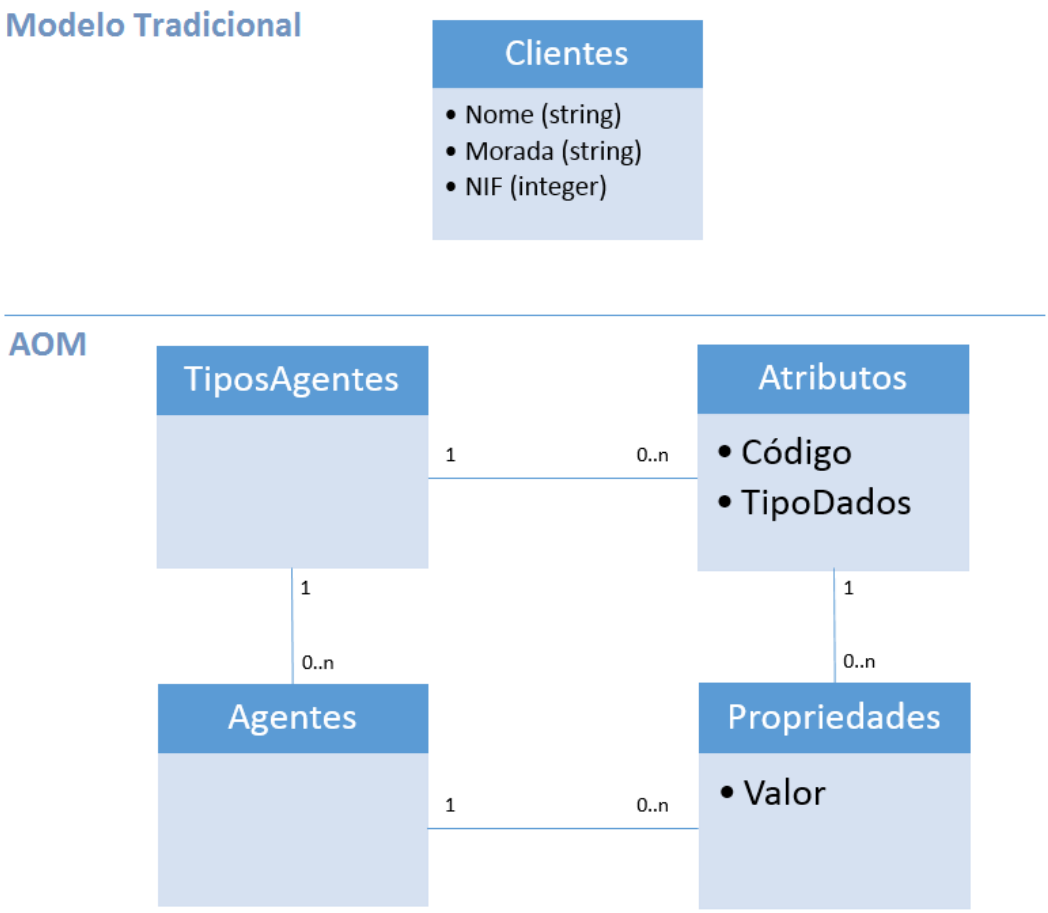

Figura 6 – Modelo tradicional e AOM

No esquema anterior podemos ver, para os modelos tradicionais, que a definição de uma entidade e dos seus atributos é toda realizada na mesma classe, o que obriga a alterações constantes no código sempre que se pretenda adicionar ou alterar um atributo existente. Por sua vez, numa arquitetura AOM, é feita uma divisão entre os tipos de entidades e as entidades propriamente ditas, ou seja, uma divisão entre a metadata e os dados.

Nas classes TiposAgentes são definidos os tipos de Agentes do modelo, como por exemplo Colaboradores, Funcionários e Clientes, sendo os atributos (como o Nome, Morada, etc.) definidos na classe respetiva. Por sua vez, os dados são inseridos sobre as classes Agentes e Propriedades, encontrando-se relacionados com os tipos definidos na metadata.

Além da definição dos atributos, é também importante destacar a forma como são tratadas as relações entre entidades. Estas exigem um desenho mais elaborado, que represente a relação entre as duas entidades.

As arquiteturas do tipo AOM, ao permitir uma maior flexibilidade devido à definição dos comportamentos nos modelos, permitem aproximar as aplicações dos clientes, que passam a poder ajustar as aplicações às suas necessidades. No entanto, de forma a permitir que os mesmos intervenham na aplicação, torna-se necessário disponibilizar uma ferramenta que permita editar a metadata existente, tendo em conta a manutenção da integridade do mesmo perante as regras de negócio. Esta aplicação deverá ter em conta que as alterações no modelo poderão ser realizadas por indivíduos sem experiência de programação, devendo disponibilizar a informação de forma evidente para o utilizador, aproximada da sua linguagem em vez de aproximada à linguagem da aplicação ou à persistência do modelo na base de dados. Deverá também ser dada especial relevância à validação das alterações no modelo perante os dados existentes, de forma a estes não ficarem incoerentes.

Por fim, é relevante realçar que as aplicações que seguem estes princípios de desenho deverão ter especial atenção à camada de interface com o utilizador. Devido à flexibilidade existente na aplicação, esta terá de permitir a demonstração dos modelos aos utilizadores de forma clara para os mesmos, tal como uma aplicação de desenho tradicional.

2.2. Modelos Desenvolvidos

De momento existem vários modelos desenvolvidos sobre a plataforma myMIS, sendo alguns destes desenvolvidos de acordo com as especificações dos clientes, sendo já utilizados pelos mesmos. Outros modelos existentes são desenvolvidos pela equipa de modelação para explorar as potencialidades da plataforma, bem como para efetuar demonstrações dos passos necessários para criar um novo modelo, encontrando-se também outros modelos em processo de desenvolvimento.

É de realçar que os modelos enumerados são apresentados ao cliente como a base da solução. Esta pode ser modificada e complementada de forma a dar resposta ao problema específico do cliente. Em seguida serão enumerados os modelos mais relevantes e as suas características.

2.2.1. Gestão de Vendas

Este modelo foi o primeiro a ser desenvolvido sobre a plataforma myMIS, tendo acompanhado a evolução da plataforma. Como sugerido pelo nome, este modelo permite gerir todo o ciclo

de compras e vendas de produtos, incluindo os processos de gestão de encomendas, orçamentos, e guias de transporte/remessa.

Este modelo, além dos comportamentos descritos sobre a forma de interações e entidades, utiliza também o mecanismo de impressões da plataforma myMIS, bem como o motor de contabilidade da mesma. Este último traz para esta aplicação a capacidade de gerir stocks, contas de clientes/fornecedores, entre outros.

Sendo esta uma aplicação de faturação, teve de passar por uma certificação obrigatória da Autoridade Tributária, sendo atribuído o certificado com o número 1797. Como resultado desta certificação, esta aplicação permite a integração de faturas no serviço "e-Fatura" das finanças, bem como a exportação das mesmas no ficheiro SAF-T (AT, 2014).

Este modelo, entre outros, pode ser complementado com outros, de forma a adicionar funcionalidades à plataforma. Um exemplo de uma adição realizada ao modelo foi um módulo de processamento de salários. Isto permite personalizar a aplicação de cada cliente de forma ágil, podendo o mesmo escolher um dos modelos existentes para complementar o seu ou, em alternativa, efetuar o desenvolvimento.

2.2.2. Gestão de Despesas

Este modelo é de grande importância para a plataforma myMIS, visto ser o primeiro a ser desenvolvido de acordo com as especificações do cliente final, encontrando-se já em utilização pelo mesmo. O objetivo deste modelo é permitir a uma ou mais empresa(s) efetuar a gestão das despesas dos seus funcionários, permitindo que seja efetuado um reembolso sobre as mesmas.

Este modelo é composto por duas interações distintas: Relatório de Despesas e Adiantamentos de Capital, permitindo a primeira a gestão de despesas já ocorridas, sendo a segunda orientada à disponibilização de capital (p.ex. ajudas de custo) a um colaborador da empresa para despesas que irão ocorrer. Em seguida será realizada uma breve descrição de cada uma destas interações.

Iniciando pelo Relatório de Despesas, este permite que um colaborador registe as suas despesas, à medida que estas ocorrem ou várias de períodos diferentes ao mesmo tempo. Existem vários tipos de despesas pré-definidos, devendo o utilizador indicar o tipo adequado e preencher os dados respetivos. Estão previstos alguns casos particulares, como os de despesas pagas por cartão de crédito ou associadas a um determinado projeto.

Após a inserção das despesas pelo colaborador, as despesas podem passar por um ciclo de aprovações com vários estados de execução condicional. Por exemplo, um relatório de despesas só exige aprovação hierárquica se o colaborador que o inseriu tem um superior

hierárquico, entre outros casos. A definição dos estados de aprovação faz parte do modelo, podendo ser atualizada a qualquer momento, bem como ser feita a adição de novos estados.

No caso dos Adiantamentos de Capital, estes deverão ser inseridos pelos Colaboradores aos quais será entregue o capital. Apesar de ser uma interação mais simples, na qual o utilizador apenas necessita de inserir o capital requerido e a data até à qual o mesmo deverá ser disponibilizado, também passa pelo mecanismo de aprovações descrito acima.

Comuns às duas interações, são os mecanismos de notificações por email. Estes permitem o envio de emails aos aprovadores designados para os Relatórios de Despesa e Adiantamentos de Capital, bem como aos Colaboradores a informar a mudança de estado das suas interações. Este modelo tem também acesso ao motor de contabilidade da plataforma, o que permite obter informação de gestão interessante, sobre a forma de Balancetes, Extratos e Painéis de Controlo. Com este motor é possível, por exemplo, saber quanto é que um utilizador gastou num determinado período, ou qual a distribuição pelos vários tipos de despesas existentes. Este modelo utiliza também o motor de impressões da plataforma, que permite ao utilizador imprimir os seus documentos (Relatórios de Despesa e Adiantamentos).

Por fim, é de realçar que este modelo, devido à sua relevância e complexidade, será utilizado no âmbito desta tese para exemplificar o desenvolvimento dos testes.

2.2.3. Gestão de Encomendas

Este é um modelo simples, desenvolvido pela equipa da modelação com o objetivo de demonstrar a facilidade de criar um novo modelo a partir do zero na plataforma myMIS. Este modelo permite a colocação e satisfação de encomendas, através de duas interações.

Este modelo, em conjunto com a Gestão de Vendas, utiliza um mecanismo de satisfação de compromissos (Commitments). Exemplificando com este modelo, a operação de colocação de uma encomenda gera um compromisso entre a Empresa e o Cliente, em que a primeira se compromete a entregar os artigos ao segundo. O momento de envio dos artigos é também o momento de satisfação da encomenda, podendo esta ser total ou parcial. Este modelo será utilizado no âmbito desta tese para exemplificar o desenvolvimento dos testes.

2.3. Práticas de testes na empresa

Durante o desenvolvimento da plataforma e dos diversos modelos descritos, foi sentida a necessidade de efetuar mais testes que os rotineiros testes de desenvolvimento. Os vários momentos de desenvolvimento da aplicação, nos quais se incluem a passagem do desenvolvimento para o servidor de produção, obrigam a práticas de teste específicas, que vão ser descritas em seguida.

Numa fase inicial, além dos testes de desenvolvimento, foi adotada a prática da validação, aquando do lançamento de novas versões, do cumprimento das tarefas atribuídas aos elementos da equipa. Esta validação era sempre realizada por elementos da equipa não responsáveis pela tarefa em causa, de forma a garantir a imparcialidade dos testes. No entanto, rapidamente se percebeu que esta metodologia de testes, não era de todo a mais adequada por vários motivos: ao focar-se apenas num cenário em concreto, não era tido em conta o funcionamento global da plataforma. Ou seja, mesmo que as alterações efetuadas prejudicassem o funcionamento de outras funcionalidades, isso não seria validado.

De forma a colmatar esta limitação, optou-se pelo desenvolvimento de uma aplicação que permitia efetuar vários testes sobre um modelo em específico. Estes testes permitiam abordar quase todas as partes de um modelo, permitindo a inserção de interações e entidades, bem como a sua edição e aprovação. Uma vantagem inerente a esta aplicação foi a capacidade de efetuar testes de carga sobre a aplicação, tanto em ambiente de desenvolvimento como em ambiente de produção. No entanto, continuam a verificar-se limitações, como a possibilidade de testar apenas um modelo em específico, bem como estar limitado a cenários de teste fixos e testar apenas a camada de API.

O passo seguinte, tomado de forma a restringir a limitação de testes sobre a camada de API, passa pela execução de testes sobre a camada de UI nos momentos de passagem de uma versão de desenvolvimento para produção. De forma a minimizar a interferência com a versão de produtivo, o deploy da aplicação é, nos casos em que são efetuadas alterações mais profundas, efetuado para um servidor de testes, no qual são realizados alguns testes básicos, complementados por testes às alterações efetuadas no modelo ou correções na framework, de forma a verificar a sua eficácia.

Será seguro dizer que a solução atual não é adequada às necessidades da equipa de desenvolvimento e de modelação, por vários motivos, sendo os mais relevantes os seguintes:

- Específica para um modelo;
- Cenários de teste implementados bastante limitados;
- Envolve uma elevada intervenção humana;
- Testes realizados de forma informal, sem a definição de scripts de teste nem a produção de resultados concretos (sucesso/insucesso do teste).

3. Estado da Arte

Neste capítulo será realizada uma descrição das últimas práticas e ferramentas consideradas relevantes para esta tese. Desta forma, o Estado da Arte está organizado em quatro secções: Metodologias de Desenvolvimento de Software, Automatização de Testes, Ferramentas de Testes e, por fim, Metodologias de Desenvolvimento, Manutenção e Execução de testes.

3.1. Metodologias de Desenvolvimento de Software

Atualmente, o estado da arte nesta área inclui, entre outras, três metodologias que vão ser alvo de análise nesta secção: o TDD (Test-Driven Development), o ATDD (Acceptance Test-Driven Development) e o BDD (Behavior Driven Development), sendo os dois últimos baseados no primeiro. É de realçar que as metodologias descritas foram escolhidas para análise visto estarem adaptadas a ambientes de desenvolvimento Agile, sendo este o ambiente sobre o qual a plataforma myMIS é desenvolvida.

3.1.1. Test Driven Development

O TDD pode ser descrito como uma prática de programação que permite planear o desenvolvimento do código tendo em conta determinadas exceções previamente identificadas, permitindo também identificar as falhas do código desenvolvido. Estes objetivos são obtidos através do desenvolvimento de várias unidades de teste que se deverão caracterizar pela sua simplicidade, focando-se no teste de apenas uma determinada condição.

No caso de, após o desenvolvimento, serem identificadas falhas no código, dever-se-á proceder às correções necessárias no mesmo e executar novamente as unidades de teste necessárias até ao teste passar. Em seguida deverá existir um esforço da equipa de desenvolvimento para otimizar o código desenvolvido, reduzindo a replicação de código e

melhorando a performance da aplicação, devendo este passo ser concluído com um novo ciclo de testes de forma a validar se as alterações não afetaram o funcionamento da aplicação. Através da aplicação destas práticas de programação vai-nos ser possível identificar possíveis pontos de falha no código que vai ser desenvolvido, sendo conhecidos os pormenores a ter em conta antes de iniciar a fase de desenvolvimento (Hendrickson, 2008).

O TDD tem obtido grande aceitação por parte da comunidade, sendo assunto de diversas discussões por parte de indivíduos e equipas de desenvolvimento que implementaram ou pretendem implementar as suas práticas. Como resultados destas discussões, são identificados de forma recorrente possíveis pontos fracos e limitações do TDD, que serão descritos em seguida.

Uma crítica recorrente do uso das práticas do TDD prende-se com o fato dos testes serem responsáveis apenas por testar pequenas partes da aplicação, deixando de fora os testes de integração. Isto pode fazer com que, caso não sejam implementadas outras metodologias de testes, não sejam identificados bugs relevantes da aplicação.

São também comuns críticas à forma de desenvolvimento dos testes, nomeadamente à necessidade que existe por vezes de efetuar alterações na arquitetura da solução e/ou no código desenvolvido, de forma a torna-lo testável. Em diversas situações, é possível que o código desenvolvido de forma a ser testável seja mais complexo e difícil de ler (Hansson, 2014).

No entanto, é de realçar que todas estas críticas poderão não ser válidas, caso as práticas do TDD não tenham sido implementadas corretamente ou a aplicação base contenha falhas desconhecidas na sua arquitetura que reduzam a eficiência dos testes.

3.1.2. Behavior Driven Development

O BDD surge com o objetivo de complementar os princípios do TDD, nomeadamente na definição do processo de desenvolvimento. Esta metodologia segue o princípio de que a descrição dos comportamentos da aplicação deve ser realizada através da perspetiva das diversas partes interessadas (*stakeholders*), numa linguagem acessível (Chelimsky, 2010).

O desenvolvimento de software seguindo esta metodologia deve ser iniciado pela comunicação entre as partes interessadas e a equipa de desenvolvimento, sendo esperado que os primeiros descrevam o problema e o que é esperado que a solução encontrada realize. A partir desta descrição, deverão ser identificados um conjunto de comportamentos que deverão fazer parte da solução final, e que deverão estar associados ao problema apresentado pelos clientes (Solís & Wang, 2011). As features determinadas pelos clientes deverão ser transformadas em user stories (caso não sejam apresentadas nesse formato), sendo atribuída uma prioridade a cada uma, de forma a orientar o desenvolvimento da solução e definir o que deverá ser desenvolvido por cada iteração, sendo que no fim da qual deverá ser feita uma entrega ao cliente.

Cada user story identificada pode ser composta por um ou mais comportamentos e cenários, que deverão ser descritos sobre a forma de uma linguagem específica de domínio. Esta descrição permite que toda a equipa de desenvolvimento, bem como as partes interessadas, consigam ver de forma clara quais são os comportamentos esperados para uma determinada feature. Das diversas linguagens existentes, a mais indicada para a implementação dos princípios do BDD, é o Gherkin (Cucumber, 2014). Esta linguagem permite a escrita de testes e de documentação, dependendo a sua sintaxe de palavras-chave (Given, When, Then), e vai ser descrita em maior pormenor em seguida.

O Gherkin permite, além da descrição de cenários de uma feature, a construção de cenários de teste. No Gherkin, um cenário é sempre iniciado pela palavra-chave "Given", tendo esta o propósito de definir as pré-condições que devem estar reunidas antes da interação com o utilizador no sistema. Segue-se o "When", que descreve a interação do utilizador com o sistema e, por fim, o "Then", que descreve o resultado esperado (Cucumber, 2013). Todas estas palavras-chave podem ser repetidas as vezes necessárias, como podemos ver no exemplo seguinte, que descreve o acesso ao formulário de um Colaborador.

```
Given Employees list
Given logged in user
When I edit Employee EM001
Then I see Employee EM001 edition form
Then I don't see Employees list
```

Figura 7 – Teste escrito em Gherkin

Conforme mencionado anteriormente, a descrição de cada cenário pode ser utilizada como um teste de aceitação da funcionalidade. Estes testes deverão ser executados e interpretados de forma automática, sendo através do resultado que o cenário é dado como cumprido ou não. Alguns exemplos notáveis de ferramentas que utilizam esta linguagem específica de domínio e seguem os princípios do BDD são o Cucumber[1] e o SpecFlow[2], sendo o primeiro mais orientado à linguagem de programação Ruby, e o segundo à framework .NET.

3.1.3. Acceptance Test Driven Development

O ATDD segue os princípios do TDD, consistindo no desenvolvimento de testes antes de iniciar a implementação do código, sendo o seu foco principal o cumprimento de determinadas condições identificadas nas User Stories e que se espera que a aplicação cumpra. As User

[1] http://cukes.info/
[2] http://www.specflow.org/

Stories deverão ser escritas no momento de desenho de uma nova funcionalidade, e deverão ser discutidas entre a equipa de desenvolvimento e as diversas partes interessadas.

O desenvolvimento seguindo os princípios do ATDD envolve três entidades diferentes: o cliente (parte interessada), a equipa de desenvolvimento e a equipa de testes. Este envolvimento é essencial, visto permitir que todas as entidades estejam mais próximas do projeto, identificando eventuais questões numa fase inicial do desenvolvimento. Isto faz com que as equipas que optam por esta metodologia de desenvolvimento consigam ganhos de produtividade e qualidade do produto desenvolvido (Pugh, 2010).

As condições de aceitação, definidas pelas três entidades descritas acima, permitem guiar o desenvolvimento da solução. Não é suficiente desenvolver determinada funcionalidade que o cliente precisa, é necessário que esta cumpra com uma ou mais condições de aceitação. Alguns exemplos de condições de "aceitação" são, por exemplo, o tempo necessário para executar determinada tarefa, o feedback fornecido ao utilizador em caso de erro/sucesso, entre outros (Hendrickson, 2008). É de realçar que os testes de aceitação não deverão ser influenciados por alterações na implementação, pois estas não alteram as condições de aceitação definidas antes do início do desenvolvimento.

Visto o ATDD envolver partes interessadas que não estão necessariamente envolvidas na programação da solução, os testes deverão ser escritos numa linguagem que seja percetível por todas as partes (REF). Desta forma, os testes deverão ser escritos numa Business-Readable DSL[3], como o Gherkin, cujos detalhes podem ser vistos no subcapítulo anterior.

3.2. Automatização de testes

No âmbito da execução dos testes, um tema recorrente é a automatização da sua execução. Esta é importante para agilizar o processo, tornando-o mais rápido e fiável. A automatização pode ser feita em dois pontos distintos da aplicação: na interface com o utilizador (UI), ou na camada de API.

3.2.1. Camada de UI

A realização de testes sobre uma interface com o utilizador é de extrema importância, visto esta ser responsável por toda a interação com o mesmo. Normalmente espera-se que uma interface seja intuitiva, rápida e responsiva, de forma a garantir uma boa user experience.

A realização de testes automatizados sobre uma interface nem sempre é fácil, devido à grande quantidade de condições a testar e às dificuldades em simular o comportamento do utilizador de forma automatizada.

[3] http://www.martinfowler.com/bliki/BusinessReadableDSL.html

De forma a permitir os testes de forma simples e intuitiva, existe um padrão, denominado de "Page Object Pattern", que permite a definição dos elementos de uma página e permite simular a interação de um utilizador com os mesmos, através do objeto gerado (Fowler, 2013). O objeto gerado deve disponibilizar uma interface que forneça um acesso simples às propriedades, e garantir que uma aplicação consiga realizar todas as operações que um ser humano, numa determinada página.

Estes objetos permitem simplificar o acesso aos diversos elementos da página, mas não permitem, de forma independente, levar a cabo testes sobre a camada de UI. Para esse efeito é necessária a utilização de ferramentas que permitam escrever e executar os testes, através da interação com os "Page Objects". Exemplos dessas ferramentas podem ser vistos no capítulo 3.1.3.

3.2.2. Camada de API

A realização de testes sobre uma API é bastante menos complexa que os testes sobre a interface com o utilizador, devido ao ambiente ser bastante mais controlado e menos vulnerável ao ambiente externo.

Para efetuar testes sobre uma API, é necessário saber de antemão a assinatura dos serviços a testar, e o resultado esperado para o *input* fornecido. A automatização dos testes deverá ser obtida através da utilização de ferramentas que permitam realizar as chamadas aos serviços com os parâmetros necessários e avaliar se o output corresponde ao esperado.

3.3. Ferramentas de Teste

A escolha das ferramentas de teste a utilizar depende de vários fatores, e deve ser cuidadosamente ponderada no momento de início do desenvolvimento da aplicação e da lógica de testes. Alguns dos parâmetros mais relevantes a ter em conta são o tipo de aplicação a testar (aplicação de consola, aplicação web, entre outras), a metodologia de testes que vai ser seguida e o tipo de testes que se pretende realizar. Outras razões passam pela escolha entre software livre ou pago, entre outros. Neste subcapítulo vão ser analisadas as ferramentas de teste consideradas mais relevantes, tendo em conta o problema apresentado. A análise será organizada em três categorias principais: a execução de testes sobre a interface com o utilizador, sobre a camada de API e, por fim, a interpretação dos testes. Na análise de cada uma destas ferramentas, serão descritos os pontos fortes e fracos de cada uma, bem como escolhidas as ferramentas que compõem a proposta de solução.

3.3.1. Camada de UI

3.3.1.1. Selenium WebDriver

A aplicação Selenium permite, conforme o nome sugere, a automação de ações num navegador de Internet, tendo suporte para diversos navegadores e sistemas operativos[4], sendo possível a sua utilização com várias linguagens de programação[5]. Sendo uma aplicação que permite apenas testar ambientes web, é importante mencioná-la no contexto desta tese, visto a aplicação myMIS ter um cliente web, executado a partir do navegador e sobre o qual deverá ser possível realizar testes.

O Selenium WebDriver permite simular a interação de um utilizador de forma automática. Estes comportamentos são reproduzidos através da utilização dos mecanismos de automação de cada navegador de internet, o que permite uma alta compatibilidade. A utilização dos mecanismos nativos de cada um representa uma grande evolução perante a versão anterior do Selenium (denominada Selenium Remote Control), que reproduzia os comportamentos via Javascript, de forma semelhante para todos os navegadores e sistemas operativos.

Esta ferramenta permite a realização de várias operações básicas, sendo as mais relevantes a interação com as janelas do navegador (abertura, fecho, escrita de url's), bem como a interação com as páginas abertas pelo mesmo, através da simulação da ação do rato e do teclado sobre os vários elementos DOM da página. É também possível a obtenção de valores da página, o que permite validar, por exemplo, se valores computados numa determinada página quando recebem um determinado input estão de acordo com o esperado.

Esta aplicação, ao permitir a reprodução dos comportamentos de um utilizador é interessante para a execução de testes sobre a camada de UI. No entanto, de forma a poder ser utilizada, tem de ser complementada com outras aplicações, que permitam a escrita e interpretação dos testes, bem como a transformação dos mesmos em ações dos utilizadores.

Após a análise da ferramenta Selenium, foi possível concluir que a mesma cumpre com esperado, permitindo a execução de ações sobre os navegadores mais utilizados, da mesma forma que um utilizador faria, sendo possível a manipulação das páginas web de forma fácil e intuitiva. Desta forma, esta ferramenta foi incluída na proposta de solução, destinando-se a executar os testes sobre o cliente de UI da plataforma myMIS.

3.3.1.2. PhantomJS

O PhantomJS é um ambiente que permite o teste de aplicações web de uma forma "Headless", ou seja, sem a necessidade de recorrer a um navegador. Os testes são realizados sobre a componente javascript das páginas recorrendo ao motor de renderização WebKit. Este motor é utilizado por vários navegadores, sendo o mais relevante o Safari da Apple. Foi no entanto

[4] http://www.seleniumhq.org/about/platforms.jsp#browsers
[5] http://www.seleniumhq.org/about/platforms.jsp#programming-languages

utilizado também pelo navegador Opera e Chrome até à sua migração em 2013 para o motor Blink. A utilização de tecnologias comprovadas e adotadas por vários navegadores permite que se consigam reproduzir com o PhantomJS praticamente todas as ações que se podem executar num browser (Beltran, 2013).

A realização de testes com esta ferramenta é realizada através da linha de comandos, sendo esta complementada com uma API de Javascript de forma a facilitar a manipulação dos elementos de uma página, a inserir scripts, aceder e escrever em ficheiros, entre outras características. A sua reduzida dimensão, bem como a não necessidade de instanciar uma janela de um navegador permite que os testes sejam executados com uma rapidez superior a outaras ferramentas de automação, nas quais se inclui o Selenium (Chasins & Phothilimthana, 2014).

Apesar das vantagens evidentes ao nível da performance e do consumo de recursos, esta forma de execução dos testes apresenta maiores dificuldades em desenvolver e manter a integridade dos testes que o Selenium (Chasins & Phothilimthana, 2014), visto este correr sobre instâncias de um determinado browser, não num ambiente genérico. Desta forma, das ferramentas analisadas no âmbito da automatização dos testes, esta ferramenta não foi escolhida em detrimento do Selenium, devido a este último permitir uma simulação mais exata do ambiente apresentado ao utilizador, e devido à maior facilidade em desenvolver os comportamentos a executar.

3.3.2. Camada de API

Por sua vez, o desenvolvimento de testes sobre a API trouxe desafios diferentes, nomeadamente ao nível da reprodução dos comportamentos esperados. Visto estes não serem reproduzidos pelo cliente de UI, existe toda uma lógica que se encontra encapsulada pelo mesmo, e que tem de ser desenvolvida com o auxílio de outras ferramentas, que vão ser descritas em seguida. Conforme descrito anteriormente, já tinha existido algum desenvolvimento prévio de testes sobre a camada de API, tendo sido já realizada uma escolha de ferramentas a utilizar.

As questões identificadas começam pela necessidade de implementar um mecanismo de autenticação de utilizadores. Conforme mencionado anteriormente, a plataforma utiliza o standard OAuth 2.0 para a autenticação e autorização dos utilizadores, sendo essencial a utilização de auxiliares para a gestão dos tokens de acesso. Após a análise das bibliotecas existentes, decidiu-se proceder à utilização da "DotNetOpenAuth"[6], pela sua documentação extensa e pela sua completa integração com a framework .NET, na qual se inclui a disponibilização da aplicação via nuget[7].

[6] http://dotnetopenauth.net/
[7] https://www.nuget.org/packages/DotNetOpenAuth/

Após a autenticação, outra das questões passa pela realização de pedidos das várias operações CRUD. Tendo em conta que a resposta aos pedidos de consulta é devolvida em formato JSON, bem como o envio dos dados nas operações de escrita também ser realizado neste formato, revelou-se necessária a utilização de um auxiliar, que permita a conversão de objetos em JSON e vice-versa. Desta forma, e tendo sempre em conta a integração com a framework .NET, foi escolhida a biblioteca JSON.NET[8]. Por fim, e de forma a auxiliar a construção dos pedidos HTTP à API, foi escolhida a biblioteca RestSharp[9], que permite a execução de pedidos de forma síncrona e assíncrona.

3.3.3. Interpretação de Testes

3.3.3.1. Cucumber

O Cucumber é uma ferramenta que implementa os princípios do processo de desenvolvimento de software BDD, permitindo a descrição dos comportamentos esperados numa linguagem acessível aos vários stakeholders e utilizando a mesma informação como testes, documentação e ajuda ao desenvolvimento.

De forma a cumprir com o especificado, esta ferramenta utiliza uma DSL denominada Gherkin, cuja sintaxe se encontra descrita no capítulo 3.1.3. Esta é escrita e orientada à linguagem Ruby, mas é comum a sua utilização para testar aplicações escritas noutras linguagens de programação, utilizando apenas algumas características da ferramenta, como o interpretador da DSL Gherkin, sendo os testes executados com o auxílio de outras ferramentas.

Após a análise desta ferramenta, foi possível verificar que a mesma poderia ser utilizada na proposta de solução, visto permitir a interpretação dos testes desenvolvidos numa DSL específica. No entanto, a existência de aplicações mais adequadas à framework .NET, fez com que a ferramenta Cucumber não fosse incluída na proposta de solução.

3.3.3.2. Specflow

A ferramenta Specflow permite, tal como o Cucumber, a interpretação de testes escritos na DSL Gherkin, sendo no entanto orientado especificamente à framework .NET. Esta aplicação tem o objetivo de permitir implementar nos projetos .NET uma aproximação aos princípios do "Specification-By-Example" (Adzic, 2014), sendo que os princípios mais relevantes desta abordagem são comuns aos processos de desenvolvimento ATDD e BDD.

Sendo esta ferramenta orientada à framework .NET, tem uma grande integração com o IDE Visual Studio. Isto permite com que a instalação seja efetuada de forma simples e intuitiva

[8] http://james.newtonking.com/json
[9] http://restsharp.org/

através do gestor de pacotes NuGet[10]. Após a instalação, são disponibilizados novos tipos de ficheiros que podem ser adicionados de forma simples à solução.

A escrita dos cenários a testar deve ser realizada em ficheiros .feature, seguindo os princípios da linguagem Gherkin. Após a escrita do teste, devem ser desenvolvidas em C# as funções que irão executar os comportamentos esperados para cada frase do teste, sendo que para a avaliação do resultado do teste (palavra-chave "Then"), deverá ser realizada a avaliação se o resultado é o esperado, devolvendo ao utilizador a informação se o teste sucedeu ou não. Para cada função podem ser utilizadas outras ferramentas que façam sentido, como o Selenium para reproduzir comportamentos no navegador web, ou o Test Explorer do Visual Studio para avaliar as condições de sucesso do teste. É de realçar que, de forma a esta aplicação poder ser utilizada, tem obrigatoriamente de ser complementada com outras que realizem a execução das ações e avaliação do resultado dos testes.

Após a análise desta ferramenta, foi verificado que a mesma cumpre com o pretendido, ao permitir a interpretação dos testes de forma fácil e dinâmica. A sua completa integração com a framework .NET e a facilidade com que a mesma pode ser utilizada com outras ferramentas de testes, fez com que o Specflow fosse incluído na solução final, sendo responsável pela interpretação dos testes desenvolvidos.

3.4. Metodologia de Manutenção e Execução de testes

Após a definição e desenvolvimento da lógica de implementação dos testes, é relevante definir a lógica de builds e execução dos testes associados. Seguindo as lógicas de programação ágil, foi escolhida para análise neste subcapítulo a prática de gestão e deploy de build Continuous Integration, pela sua relevância no panorama atual e adequação à plataforma myMIS.

Continuous Integration é uma prática de gestão e de deploy de builds utilizada por equipas que seguem o padrão de desenvolvimento ágil é o Continuous Integration. Esta pode ser descrita como um conjunto de ações que permitem aumentar a velocidade das entregas de novas versões, através, por exemplo, da realização de builds e de testes de forma automática. Continuous Integration é considerada uma prática relevante no âmbito da metodologia de desenvolvimento de software Extreme Programming (XP).

Num ambiente de desenvolvimento ágil, é comum que os programadores da equipa realizem uma ou mais integrações de código por dia, o que faz com que existam várias versões atualizadas da aplicação por dia. De forma a agilizar a disponibilização de novas versões, o comportamento esperado será que, de uma forma automática, o código seja compilado e sejam realizados os testes adequados, de forma a garantir a integridade da nova versão da aplicação.

[10] https://www.nuget.org/

Desde cada integração de código até à sua inserção na versão de produção, deverá ser seguida uma sequência de passos. Esta inicia-se pela realização de uma alteração na versão atual, que deverá ser identificada de forma automática pela plataforma e despoletar a compilação do código. Caso esta termine com sucesso e existam testes unitários, deverão ser executados neste momento. Após a execução dos testes unitários com sucesso, deverá ser feita uma publicação da aplicação para um servidor de testes e executados os testes de aceitação. Por fim, caso estes decorram com sucesso, deverá ser feita a publicação para o servidor de produção, devendo a equipa de desenvolvimento ser notificada do sucesso da publicação, entre outros pormenores da operação (Stolberg, 2009).

Todo este processo permite tornar o processo de testes acessível a todos os elementos da equipa de desenvolvimento, deixando de haver a necessidade de existir um ou mais elementos alocados de forma permanente ao desenvolvimento de testes. Para a execução destas práticas é necessária a utilização de um repositório, sendo uma boa prática não colocar apenas o código da aplicação, mas também todos os scripts necessários, documentação, bem como tudo o que seja necessário para a manutenção da aplicação. Toda esta informação deve ser partilhada entre os elementos da equipa de desenvolvimento, não estar limitada a algumas máquinas locais.

Será também importante a utilização de uma plataforma que permita a realização de builds e a, execução de testes. Iniciando pela primeira, deverá ser feito o esforço de automatizar o máximo de procedimentos possível. O ideal será que seja possível partir de uma máquina limpa e, com a execução de um comando, conseguir colocar a aplicação pronta a executar (Fowler, Continuous Integration, s.d.). Após o início da build deverá ser efetuado o controlo do estado da mesma de forma periódica até ao fim da mesma, de forma a identificar e corrigir possíveis falhas o mais rápido possível. Deverá ser realizado um esforço de manter o tempo de geração e publicação da build o mais reduzido possível, de forma a causar a mínima entropia no sistema.

4. Proposta de solução

4.1. Características principais da solução

A solução proposta foi desenvolvida de forma a permitir o desenvolvimento da framework e dos modelos interpretados pela mesma de uma forma mais orientada às user stories, e principalmente de uma forma que permita o desenvolvimento e execução de testes a diferentes modelos interpretados pela plataforma com um esforço reduzido. A proposta de solução aqui apresentada permitirá o desenvolvimento de testes a um número limitado de features (identificadas no contexto de user stories) para todos os modelos suportados, podendo os testes ser realizados sobre o cliente de UI ou sobre a camada de API da plataforma myMIS.

Esta proposta foi definida de acordo com os princípios da metodologia de desenvolvimento BDD. Esta metodologia, derivada dos princípios do TDD, foi escolhida por relevar a importância da definição pormenorizada das features a testar em cada unidade de teste, o que origina testes de dimensões mais reduzidas mais fáceis de desenvolver, de manter e de execução mais rápida. Outra vantagem desta abordagem é a utilização de uma DSL para a descrição dos comportamentos pretendidos numa feature. Esta descrição, ao ser escrita numa linguagem de fácil interpretação, pode ser utilizada como documentação acessível tanto à equipa de desenvolvimento como aos stakeholders, existindo também a vantagem de, com pouco esforço, se obterem testes de aceitação.

A escolha da DSL a utilizar para o desenvolvimento dos testes foi o Gherkin. Esta foi escolhida maioritariamente pela vasta documentação disponível, bem como pelas várias aplicações que interpretam esta DSL e implementam os princípios do BDD. Desta forma, a descrição dos cenários e os testes de aceitação serão desenvolvidos nesta linguagem, para posterior interpretação pela framework de testes.

A abordagem escolhida no desenvolvimento desta proposta tem em conta, sempre que possível, a vista do utilizador, sendo a maioria dos testes executados num ambiente em tudo semelhante ao cliente de UI da aplicação. No entanto, alguns dos testes que foram identificados não se adequam totalmente a uma interpretação pela UI, por serem extremamente mais complexos, tanto no processo de escrita dos mesmos como no desenvolvimento da plataforma que efetuará a sua interpretação. De forma a diminuir ao máximo a complexidade (o que garante uma manutenção dos testes mais eficaz), optou-se por efetuar testes não só sobre o cliente de UI, mas também sobre a camada de API, o que se pode traduzir em dividir um único teste em vários testes distintos, mas com uma relação de dependência entre eles. É de realçar que os testes sobre a camada de API só deverão ser utilizados para os casos em que esta abordagem se revele mais adequada que a execução de testes sobre a interface, como por exemplo, para a consulta de informações sobre um

documento persistido, valores das contas do sistema, entre outros. Os testes sobre a API trouxeram outros desafios não aplicáveis aos testes sobre a user interface, que serão descritos em seguida.

A API da plataforma é, conforme descrito no subcapítulo 2.1, uma API RESTful que utiliza o mecanismo de autorização OAuth. No caso dos testes realizados a partir do cliente de UI, todos estes detalhes são irrelevantes, pelo fato de estarmos a efetuar testes sobre a ótica do utilizador. É apenas necessário efetuar o login com um utilizador válido, e toda a restante lógica de autorização e de realização de pedidos é realizada pelo cliente de UI. A realização de testes diretos à API revela-se mais complexa por obrigar à construção e gestão dos pedidos, bem como da lógica de autenticação.

Como parte da proposta de solução, serão também descritos nos subcapítulos seguintes as abordagens pensadas para a execução dos testes de forma automatizada, bem como a sua integração no mecanismo de builds.

4.2. Implementação da solução

A solução encontrada passa pela disponibilização de um módulo que permite a escrita e interpretação de testes sobre diferentes modelos, de forma a verificar se os mesmos são válidos perante a plataforma, bem como se potenciais alterações da mesma não afetaram de forma inesperada a interpretação dos modelos.

O módulo proposto como solução para o problema apresentado foi desenvolvido na linguagem de programação C#, na qual também foi realizado o desenvolvimento da plataforma. Os testes desenvolvidos podem ser executados, conforme mencionado anteriormente, sobre a camada de UI ou sobre a API. Em seguida serão descritos os procedimentos de escrita dos testes, bem como descrita a abordagem seguida para a interpretação dos testes sobre a camada de UI e sobre a API.

4.2.1. Pré-Condições

Os testes desenvolvidos deverão ser executados sempre que se revele adequado, nomeadamente em momentos de colocação de uma nova versão no ambiente de testes ou produtivo, bem como quando são efetuadas alterações ao modelo. De forma a garantir a integridade dos testes, é importante assegurar que os mesmos são sempre executados num ambiente semelhante.

Desta forma, foi definido que os testes devem ser sempre realizados sobre contas geradas para o efeito, não sobre contas de teste já existentes. Este procedimento garante que os testes

são sempre executados sobre a última versão do modelo, sendo que o tenant em causa possuirá apenas os dados necessários para o funcionamento dos testes.

Conforme mencionado anteriormente, poderão existir casos de dependência entre testes. Isto obriga que os mesmos tenham como pré-condição a realização prévia de determinados testes, sempre sobre um tenant criado de novo. Este pormenor é relevante para garantir que os testes possam referir entidades e interações com um determinado identificador. Um exemplo será a necessidade de validar os dados de uma interação que acabou de ser inserida. Através do controle da ordem de execução dos testes e da garantia que estes foram executados sobre um tenant que não continha interações, podemos indicar explicitamente o número da interação sobre a qual queremos efetuar a validação, o que permite a simplificação da escrita do teste.

Por fim, de forma a garantir que o módulo de testes suporta a interpretação de casos de teste de vários modelos distintos, é necessária a adição de algumas configurações prévias. Estas configurações são lidas no momento do arranque da aplicação de testes, e contêm informação específica de cada modelo a testar.

Para cada modelo, é necessária a indicação dos dados do utilizador (username e password) que irá ser utilizado para a operação inicial de Login nos testes sobre as duas camadas. A indicação do tenant a ser utilizado é também necessária, de forma a permitir que, caso um utilizador tenha acesso a mais de uma conta, seja garantido que os testes são executados sobre a conta certa.

O ficheiro de configurações é também composto por informações relevantes de cada interação, de forma a permitir que os testes contenham apenas a informação estritamente necessária. Desta forma, por cada interação é necessária a indicação do processo ao qual ele pertence, sendo esta informação utilizada para permitir o acesso de forma rápida à opção certa no menu lateral, visto este organizar as interações por processo.

Por fim, revelou-se também necessária a organização dos testes existentes por modelo, sendo realizada a distribuição dos mesmos pelo ambiente no qual vão ser executados: API ou UI. Esta organização permite-nos, no momento da execução dos testes, decidir qual a camada sobre o qual vai ser executado, o que se irá traduzir em operações diferentes para o mesmo teste.

4.2.2. Testes sobre UI

A execução dos testes sobre o cliente Web da plataforma myMIS é realizada, conforme mencionado, recorrendo à ferramenta Selenium. Esta permite inicializar uma nova instância de um navegador web, executando as ações necessárias sobre o mesmo de forma automática, como se fossem realizadas por um comum utilizador.

Os testes sobre a user interface utilizam o mecanismo de interpretação dos testes que será descrito no subcapítulo 4.3.3., sendo que cada um dos métodos resulta em uma ou mais ações

a ser executadas sobre o navegador web. Na solução proposta são suportados os browsers mais utilizados no sistema operativo Windows (Chrome, Internet Explorer e Firefox), sendo também suportados os navegadores por defeito dos sistemas operativos de dispositivos móveis Android e iOS. A decisão do navegador sobre o qual os testes devem ser executados deve ser tomada antes da execução dos mesmos, sendo de momento definida num ficheiro de configurações em formato XML.

Visto cada teste funcionar de forma independente, é aberta uma nova instância do navegador escolhido por cada um. Este comportamento permite-nos identificar duas pré-condições de cada teste executado sobre a camada de UI: todos os testes têm de ser iniciados pela operação de login do utilizador que foi configurado e, opcionalmente, caso o mesmo tenha acesso a mais que uma conta deverá ser indicada sobre qual o teste será realizado.

Todas as fases de um teste (Pré-condições, ações a executar e avaliação de resultados) utilizam as mesmas operações básicas de interação com a página web. As operações mais relevantes disponibilizadas pela framework Selenium em uso são as seguintes:

- Abertura de uma nova instância do navegador, através da indicação do URL inicial;
- Interação com elementos DOM da página: clique em botões, seleção de caixas de texto, listas, entre outros;
- Inserção de valores em elementos do tipo input e listas;
- Procura de elementos na página por vários atributos do elemento (p.ex.: ID, nome, classes, entre outros);
- Obtenção do valor dos elementos de uma página;
- Possibilidade de condicionar o teste mediante o estado dos elementos da página: Por exemplo, é possível avaliar se um elemento da página se encontra editável ou visível antes de se prosseguir com o teste, permitindo comportamentos diferentes mediante o resultado.

Com estas operações conseguimos reproduzir os comportamentos necessários para a execução dos testes sobre a camada de UI. Nos métodos que executam as pré-condições e as ações dos utilizadores, são usadas maioritariamente as operações de seleção de elementos para a navegação na aplicação e a inserção de dados. Por sua vez, na avaliação de resultados é realizada uma maior utilização das operações de consulta de valores, de forma a validar se o resultado é o esperado.

4.2.3. Testes sobre API

Por sua vez, os testes sobre a camada de API são desenvolvidos com a utilização de várias ferramentas que permitem a gestão da autenticação, bem como a geração e execução de pedidos a esta camada.

Ao contrário dos testes realizados com o cliente de UI, estes não têm de ser inicializados com a autenticação do utilizador. Isto deve-se ao fato de ser possível executar procedimentos do teste, como a composição de um pedido, sem a necessidade de autenticação. No entanto esta tem obrigatoriamente de ser feita antes de ser executado o primeiro pedido à API.

Os testes sobre esta camada utilizam, tal como os testes sobre a camada de UI, o mecanismo de interpretação descrito no subcapítulo 4.3.3., permitindo-nos este mecanismo que, com o mesmo código, seja possível efetuar chamadas aos serviços com diferentes parâmetros. Após a execução das chamadas à API, os resultados devolvidos pela mesma são interpretados, sendo obtida a informação necessária para a avaliação do sucesso do teste.

4.2.4. Escrita e Interpretação de testes

Nesta proposta de solução do problema apresentado, a escrita dos testes será realizada na DSL Gherkin, sendo utilizada a ferramenta Specflow para a escrita e interpretação dos mesmos.

Iniciando pela escrita dos testes, esta tem de ser realizada em ficheiros com uma formatação específica, de forma a serem interpretados pelo Specflow. Estes ficheiros, com a extensão .feature, têm o seguinte formato:

- Feature: De forma a permitir que estes ficheiros sirvam não só para a realização de testes, mas também como documentação, estes poderão ser iniciados com uma descrição do caso a testar.
- Scenario: Contém uma descrição do cenário específico que está a ser testado, sendo que um caso pode conter vários cenários. Segue-se a escrita do teste, em linguagem Gherkin

Passando para a interpretação dos testes, a orientação da plataforma myMIS a modelos faz com que esta não seja tão linear como em aplicações desenvolvidas de uma forma tradicional. De forma a permitir que os testes para qualquer modelo sejam desenvolvidos de forma ágil e por utilizadores com menores conhecimentos de programação, é esperado que a framework de testes resultante permita a interpretação de scripts de teste de uma forma genérica, através da identificação de padrões nas operações efetuadas nos testes. Este comportamento

da framework de testes é essencial para não ser necessário o desenvolvimento de código específico por cada modelo interpretado pela plataforma myMIS, o que se iria traduzir na replicação de código, bem como na impossibilidade dos testes serem desenvolvidos pelos mesmas pessoas responsáveis pelo desenvolvimento dos modelos.

De forma a permitir os comportamentos descritos acima, verificou-se a necessidade de interpretar as frases de cada teste de forma genérica. Ou seja, todas as frases avaliadas deverão ter uma estrutura base, sendo algumas partes da mesma específicas do modelo em causa, ou representativas do input do utilizador. A ferramenta Specflow, escolhida nesta proposta para a interpretação dos testes, tem em conta estas necessidades, sendo possível desenvolver métodos que interpretem um conjunto de frases que obedeçam a uma determinada expressão regular, conforme podemos ver em seguida:

```
[Given("I open Create form of an (.*) Interaction")]
0 references
public void IOpenAnInteraction(string reportCode)
{
    interactionNode = doc.SelectSingleNode("Models/" + modelToTest +"/Interactions/"+reportCode);
    string processCode = interactionNode.SelectSingleNode("Process").InnerText;
    automation.OpenInteraction(processCode, reportCode);
}
```

Figura 8 – Interpretação de uma condição com o Specflow

O bloco de código acima é um simples método que avalia uma pré-condição de um teste, referindo-se à ação de abrir um formulário de criar uma nova Interação. Um problema que poderia advir da necessidade de testar múltiplos modelos, ou múltiplas interações de um só modelo, seria a necessidade de criar vários blocos de código em tudo semelhantes, nos quais apenas mudaria o identificador da Interação em causa. No entanto, através do uso de expressões regulares, é possível interpretar testes de forma genérica, utilizando o mesmo código para vários cenários de teste. Neste caso, o método descrito acima será capaz de efetuar a abertura de todas as interações que o modelo a ser testado possua, desde que seja indicado corretamente o identificador (código) da mesma, sendo suportados todos os modelos interpretados pela plataforma.
Como podemos verificar, os blocos de código são precedidos por um identificador, que nos permite a descrição da frase que este método avalia. Esta frase pode ser composta por uma expressão regular, sendo que, sempre que exista uma que encaixe no padrão definido, a lógica definida será executada. Esta lógica será, dependendo do teste, a execução de ações na User Interface, ou a execução de, por exemplo, um pedido na API ou de uma ação necessária para a composição desse pedido.

Tendo em conta esta característica da ferramenta Specflow, foi realizada a identificação das frases a suportar para esta proposta inicial (enumeradas no Anexo 1), bem como determinadas características que devem ser cumpridas na sua escrita. Visto a maioria dos testes identificados

nesta proposta inicial simularem a interação do utilizador com o sistema, revelou-se necessário distinguir, de forma clara, os dados que se pretendem inserir e em que propriedade é que essa inserção deve ser feita do resto da frase. Esta distinção deve ser feita destacando o valor que se pretende inserir com o caracter plica (‘), sendo este valor sempre seguido do Código do atributo no qual o mesmo irá ser colocado, como podemos ver em seguida:

```
When I enter 'Xpto002' in Code
```

Figura 9 – Inserção de valor num campo

Esta frase será traduzida na inserção do texto “Xpto002”, no elemento HTML que tenha o ID “Code”. No caso do preenchimento de linhas, esta lógica foi estendida de forma a permitir a adição de uma linha e o preenchimento de vários campos da mesma numa só frase, de forma a reduzir a complexidade do teste:

```
When I enter a line with 'PROD001' Resource,with Quantity '20' and ReceiverAgent 'CUST001'
```

Figura 10 – Inserção de uma nova linha

Neste exemplo, seria introduzida uma nova linha na grelha de uma interação, com os valores “PROD001” para o atributo “Resource”, “20” para “Quantity” e “CUST001” para o atributo “ReceiverAgent”.

A parte final de qualquer teste envolve a avaliação do sucesso do mesmo. Esta avaliação pode tomar várias formas, sendo de momento suportadas a validação se determinada informação do documento tem o valor esperado, bem como a validação da gravação de novas entidades/interações ou alterações às já existentes. Visto a framework myMIS executar todas as operações de escrita de forma assíncrona através da utilização de comandos, revelou-se necessária a implementação de formas de avaliação do resultado dos mesmos.

Durante o desenvolvimento dos mecanismos de avaliação dos comandos para os testes executados sobre o cliente de UI, foram identificados dois caminhos possíveis para obter os resultados esperados: a primeira proposta seria a obtenção do resultado dos comandos através de chamadas a serviços existentes na API, já utilizados pelo cliente de UI para transmitir informação aos utilizadores. Este caminho foi descartado devido à necessidade de realizar chamadas à API para a obtenção de dados já devolvidos para o cliente de UI, sendo possível, com algumas alterações, efetuar a leitura dos mesmos.

Desta forma, e visto já existir toda uma estrutura de avaliação e demonstração do resultado dos comandos aos utilizadores, optou-se pelo segundo caminho possível, que recorre ao uso do cliente de UI para avaliar o resultado dos mesmos. No entanto, rapidamente se verificou que a forma de disposição da informação, apesar de ser fácil de interpretar para o utilizador da aplicação, não permitia a avaliação por parte do Selenium.

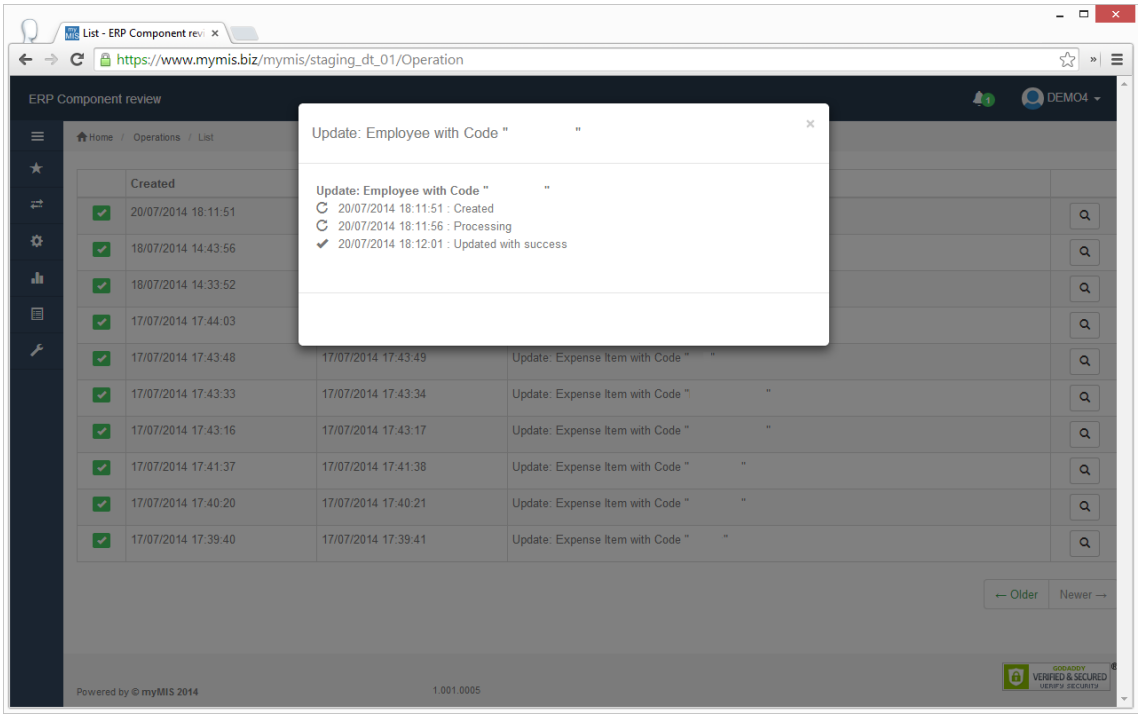

Figura 11 – Estado de um comando

Desta forma, foram realizadas alterações no cliente de UI da plataforma myMIS para este permitir o acesso fácil ao estado do último comando gerado. Esta informação encontra-se disponível num elemento invisível da página, e pode ser consultado mesmo que seja feita a navegação para outas páginas da aplicação. Com esta informação, conseguimos verificar se o comando foi processado com sucesso ou com erro, sendo o resultado do teste inferido do resultado do comando.

4.2.5. Proposta de Deploy e execução dos testes

Neste subcapítulo será realizada a descrição da proposta de execução dos testes desenvolvidos num ambiente de utilização real. A plataforma myMIS, devido à sua orientação a modelos, pode ser descrita como tendo duas frentes de desenvolvimento: a framework e os modelos. Conforme mencionado anteriormente, qualquer incremento na aplicação, seja em forma de modelo ou código, deverá ser alvo de testes adequados. Desta forma, revelou-se necessária a definição de estratégias de execução dos testes para estas duas situações, que serão descritos em seguida.

Iniciando pelos procedimentos de teste a executar na ocorrência de alterações no código da plataforma, a proposta apresentada no âmbito desta tese segue as práticas definidas pela metodologia de desenvolvimento Continuous Integration, descrita no subcapítulo 3.4.1. Esta metodologia defende a realização de múltiplas contribuições dos programadores por dia para

a versão de produção, sendo a versão atualizada várias vezes ao dia sendo que, de forma a conseguirmos aplicar esta prática, o mecanismo de geração de builds deverá ser o mais automatizado possível.

No processo de automatização das builds têm de ser incluídos os procedimentos necessários para a execução dos testes disponíveis. Estes procedimentos deverão incluir a instanciação das contas necessárias à execução dos testes, bem como o deploy da aplicação para um ambiente de staging, em tudo semelhante ao de produção, onde todos os testes serão executados de forma automática. O resultado dos testes, em caso de falha, deverá ser transmitido ao utilizador, sendo todo o processo anulado em caso de falha de um teste. Caso todos os testes sejam executados com sucesso, deverá ser realizada a troca da versão em produção, com a transmissão dos resultados dos testes e de informações relevantes aos utilizadores.

Por sua vez, a execução dos testes durante o desenvolvimento dos modelos adota uma lógica diferente. Enquanto que as alterações na framework exigem a execução de todos os testes disponíveis, por uma questão de coerência do funcionamento da framework e dos modelos, as alterações efetuadas num modelo específico não afetam a integridade dos restantes modelos existentes. Desta forma, a proposta aqui apresentada passa pela definição de um mecanismo que deverá ser despoletado pelo modelador sempre que efetua alterações num ou em mais modelos. A indicação da alteração de um ou mais modelos deverá iniciar a instanciação de novos tenants, um por cada modelo a testar. Em seguida, os testes, que estarão relacionados com um modelo em específico, serão executados sobre os novos tenants, sendo os resultados fornecidos ao modelador.

5. Testes a executar

Os testes a executar sobre os modelos interpretados pela plataforma myMIS deverão ser desenvolvidos de acordo com as *features* identificadas, sendo cada teste bastante específico e responsável apenas por testar uma parte reduzida do modelo.

Para efetuar a avaliação da solução proposta, foi identificado um conjunto de features com maior relevância nos modelos da plataforma myMIS, tendo sido desenvolvido um teste para cada uma das features, permitindo este a validação do cumprimento da mesma.

Um dos requisitos mais importantes da solução apresentada é a necessidade da mesma permitir o desenvolvimento de testes sobre todos os modelos da plataforma myMIS, sem a necessidade de desenvolvimento de código específico. De forma a conseguir validar se este requisito é cumprido pela solução proposta, os testes vão ser realizados sobre dois modelos distintos, permitindo assim demonstrar a flexibilidade da plataforma de testes. Para conseguirmos fazer esta verificação, as features escolhidas entre os dois modelos possuem várias semelhanças, focando-se nas mesmas operações.

Iniciando pela descrição do primeiro modelo utilizado (Gestão de Despesas), este tem como objetivo permitir que uma empresa faça a gestão das despesas inseridas pelos seus funcionários, bem como permitir que sejam feitos adiantamentos de capital aos mesmos.

Neste modelo, toda a informação das despesas é inserida pelos colaboradores que as efetuaram sobre a forma de Relatórios de Despesas, sendo que um relatório pode conter várias despesas realizadas em momentos diferentes. Cada relatório passa por várias fases de aprovação e validação realizadas por outros utilizadores do sistema que possuem privilégios para tal. A cada inserção ou alteração no ciclo de aprovações são despoletadas notificações de email para as entidades envolvidas (colaborador e aprovador seguinte).

Este modelo caracteriza-se também por utilizar o motor de impressões da plataforma myMIS, bem como por permitir a integração dos documentos gerados em outros sistemas. Durante as diversas iterações de desenvolvimento do modelo foram identificadas diversas *features* que necessitam de ser testadas, tendo sido algumas delas escolhidas como casos de teste no desenvolvimento da framework de testes.

No caso do segundo modelo (Gestão de Encomendas), este tem como objetivo permitir que uma empresa efetue a gestão das encomendas colocadas pelos clientes, sendo possível registar tanto a operação de colocar uma nova encomenda bem como de fechar a mesma, através da criação de uma guia de remessa.

5.1. Gestão de Despesas: Cenários de Teste

No âmbito dos testes ao modelo de Gestão de Despesas, foram escolhidas 6 *features* a testar. Estas abrangem diferentes áreas da aplicação, desde a introdução de novas entidades até à inserção e aprovação de Interações.

- Feature 1: Como colaborador, necessito de registar as minhas despesas

A primeira *feature,* apesar da sua aparente simplicidade, é o ponto de partida para todas as despesas inseridas no sistema e responsável por despoletar diversas ações que irão resultar na intervenção de vários utilizadores. Tendo em conta que esta ação é sempre executada por um utilizador através do cliente de UI, o teste resultante será executado sobre este cliente.

Para este teste ser realizado o modelo tem de cumprir com as seguintes pré-condições:

- Tem de existir um perfil de segurança que permita a inserção de Relatórios de Despesa;

- Tem de existir um utilizador associado a um Colaborador e com um perfil de segurança adequado.

Estando estas pré-condições asseguradas, o teste pode ser desenvolvido e executado. A definição do mesmo em linguagem Gherkin é a seguinte:

```
Given I am in the main page
And I open Create form of an ExpenseReport Interaction
When I enter a line with 'TAXI' Resource,with ExpenseAmount '20' and DateOccurred '19-05-2014'
When I enter a line with 'Comboio' Resource,with ExpenseAmount '15' and DateOccurred '19-05-2014'
When I press the button Submit For Approval
Then the Document should be saved
```

Figura 12 – Gestão de despesas: Feature 1

Efetuando a decomposição do teste por linhas, facilmente conseguimos dividir as diferentes fases do teste: pré-condições, ações do utilizador e avaliação da condição. Começando pela primeira linha, conseguimos verificar que a mesma é autodescritiva, o que evidência das vantagens da DSL Gherkin. Nesta é realizada a abertura de uma nova instância do navegador web escolhido, sendo realizada a navegação para a página inicial da aplicação. Caso não exista uma sessão válida, é também realizado o login do utilizador, sendo os dados lidos do ficheiro de configurações que acompanha a aplicação.

Por sua vez, na segunda linha é realizada a colocação do utilizador na página de inserção de uma nova interação. De forma a conseguirmos reproduzir este comportamento, é necessária a indicação da operação a efetuar, podendo esta ser uma criação, edição ou aprovação. É também necessário indicar qual o tipo de entidade que pretendemos inserir (neste caso, "ExpenseReport"), bem como a definição se a mesma é uma simples entidade ou uma iteração. Com estes dados é possível escolher no menu lateral a opção pretendida.

Com a execução destas duas primeiras linhas, encontramo-nos prontos -a inserir os dados da interação, reproduzindo a interação de um utilizador com a página. Neste caso, nas linhas 3 e 4 vamos proceder à inserção de duas novas linhas no documento, com os dados indicados. A identificação dos campos deverá ser indicada em conjunto com o valor a inserir nos mesmos, sendo essencial o valor a inserir encontrar-se delimitado pelo caracter plica. Na figura 13 podemos ver um exemplo de um relatório preenchido de forma automática, através da interpretação deste teste.

Por fim, na quinta linha é realizada a operação de submeter o documento para aprovação, através do clique de um botão, sendo o botão a carregar inferido através da informação colocada na frase. Na sexta linha é realizada a avaliação do sucesso da operação de submissão para aprovação, através da avaliação do resultado do comando gerado sendo que, caso o comando tenha sido terminado com sucesso, o teste é passado.

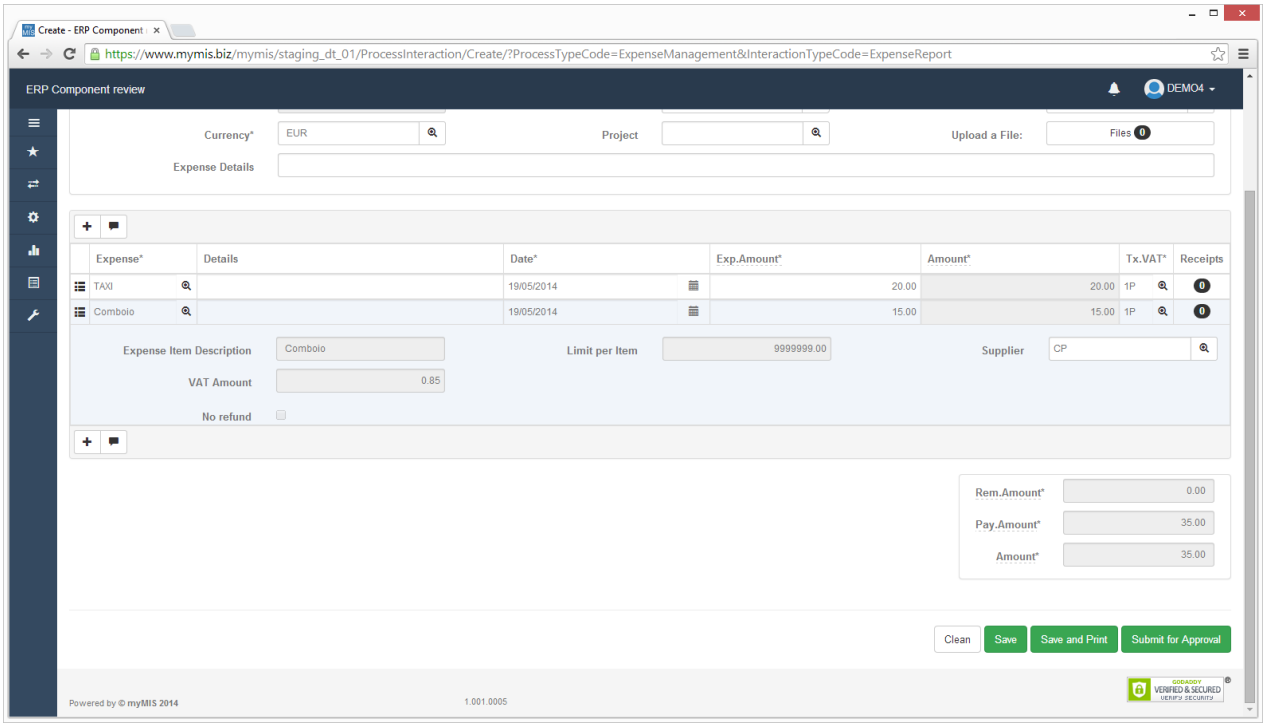

Figura 13 – Preenchimento de uma Interação

De forma a permitir a validação da integridade dos testes, foi incluído mais um teste para esta feature, que tenta efetuar a inserção de dados não válidos. Neste caso, vai ser efetuada a inserção de um tipo de despesa não existente numa das linhas do relatório de despesas, como podemos verificar na imagem seguinte:

```
Given I am in the main page
And I open Create form of an ExpenseReport Interaction
When I enter a line with 'TX' Resource,with ExpenseAmount '20' and DateOccurred '19-05-2014'
When I press the button Submit For Approval
Then the Document should be saved
```

Figura 14 – Gestão de Despesas: Teste inválido à Feature 1

É esperado que este teste não termine com sucesso, sendo identificada uma falha que deverá ser validada de forma a ser encontrada a origem do erro.

-Feature 2: Como aprovador, devo conseguir aprovar uma despesa para pagamento

Esta *feature* representa o ponto final de um Relatório de despesas dentro da plataforma myMIS. Para o modelo alvo de testes, quando o relatório chega ao estado de "validação para pagamento", caso seja considerado válido, o seu estado deve ser alterado, ficando o relatório num estado final (concluído). Tal como a feature 1, esta é executada pelos utilizadores no cliente de UI da plataforma myMIS, sendo esse o ambiente utilizado para a execução do teste.

Para este teste ser realizado o modelo tem de cumprir com as seguintes pré-condições:

- Tem de existir um perfil de segurança que permita a validação para pagamento de Relatórios de Despesa;

- Tem de existir um utilizador com um perfil de segurança adequado e com relatórios de despesa para validar.

Estando estas pré-condições asseguradas, o teste pode ser desenvolvido e executado. A definição do mesmo em linguagem Gherkin é a seguinte:

```
Given I am in the main page
And I have an ExpenseReport Interaction in approval state 'FinanceApprove'
When I change its status to Approved
When I add a Approval Note saying 'Refund to Employee'
When I press the button Save
Then the Document should be saved
```

Figura 15 – Gestão de Despesas: Feature 2

Tal como no primeiro cenário de testes descrito, o momento inicial é o Login do utilizador, sendo na segunda linha efetuada a operação de abertura de um relatório de despesas que se encontre a aguardar aprovação. De forma a ser possível a abertura da interação, é necessária a indicação do seu tipo e do estado de aprovação em que a mesma tem de se encontrar, bem como se pretendemos manipular uma entidade simples ou uma interação. Com estes dados, é possível aceder à listagem de interações do tipo relatório de despesa que se encontrem a aguardar aprovação, sendo realizada a abertura para aprovação da primeira da lista.

Na terceira linha já nos encontramos no contexto da interação, sendo efetuada a alteração do valor dos campos referentes à aprovação para "Aprovado", sendo na quarta linha adicionada uma nota (entre plicas) ao campo de notas de aprovação. Por sua vez, na quinta linha é efetuada a gravação do documento através do clique no botão, sendo de realçar que a lógica de execução desta ação é a mesma já descrita na feature 1.

Por fim, é avaliada se a gravação da aprovação decorreu com sucesso, sendo novamente reutilizada a lógica de execução desta ação é descrita na feature 1 para a avaliação do comando. Caso o comando seja terminado com sucesso, o teste é passado.

-Feature 3: Como aprovador, devo poder alterar a taxa de câmbio

Esta *feature* foi colocada pela necessidade de um aprovador, num determinado momento de aprovação, ter a liberdade de editar o valor de determinados campos, para adicionar ou corrigir a informação inserida aquando da submissão para aprovação do Relatório de Despesas. Deve ser tido em conta que, por defeito, em todos os momentos de aprovação, as interações encontram-se num estado de leitura apenas, não sendo possível editar a informação. De forma a conseguir efetuar a edição, têm de ser definidos no modelo os campos que podem ser editados.

O teste resultante desta feature será executado sobre o cliente de UI, visto este ser o ambiente no qual o utilizador irá executar esta operação.

Para este teste ser realizado o modelo tem de cumprir com as seguintes pré-condições:

- Tem de existir um perfil de segurança que permita a validação para pagamento de Relatórios de Despesa;

- Tem de existir um utilizador com um perfil de segurança adequado e com relatórios de despesa para validar.

Estando estas pré-condições asseguradas, o teste pode ser desenvolvido e executado. A definição do mesmo em linguagem Gherkin é a seguinte:

```
Given I am in the main page
And I have an ExpenseReport Interaction in approval state 'FinanceApprove'
When I enter '2.00000' in Rate
Then Rate value should be '2.00000'
```

Figura 16 – Gestão de Despesas: Feature 3

O teste é, tal como os anteriores, iniciado pela operação de Login, sendo na segunda linha efetuada a abertura de uma interação do tipo relatório de despesas no estado de aprovação para pagamento, recorrendo à lógica descrita na feature 2.

Em seguida, na linha três, é realizada a manipulação do campo “Rate”, presente no cabeçalho do documento, sendo inserido o valor que se encontra dentro de plicas. Por fim, na linha 4 é obtido o valor do campo, de forma a verificar se o mesmo foi alterado com sucesso.

-Feature 4: Como colaborador, não devo conseguir efetuar reversões de aprovações ao meu relatório de despesas

Esta *feature* prende-se com a necessidade de limitar o acesso a determinadas áreas ou funcionalidades da aplicação, dependendo do perfil de segurança que lhe foi atribuído. Neste caso em específico, pretendemos limitar o acesso de um utilizador com um perfil básico a efetuar reversões aprovações efetuadas por outros utilizadores. O teste resultante desta feature, será executado sobre o cliente de UI.

Para este teste ser realizado o modelo tem de cumprir com as seguintes pré-condições:

- Tem de existir um perfil de segurança básico que limite o acesso do utilizador ao seu perfil de colaborador;

- Tem de existir um utilizador com um perfil de segurança básico e um colaborador associado.

- Tem de existir um relatório de despesas que já tenha passado por uma etapa de aprovação

Estando estas pré-condições asseguradas, o teste pode ser desenvolvido e executado. A definição do mesmo em linguagem Gherkin é a seguinte:

```
Given I am in the main page
And I have a ExpenseReport Interaction in approval state 'Completed'
When I try to revert the approval
Then the operation should fail
```

Figura 17 – Gestão de Despesas: Feature 4

Este teste, tal como os anteriores, é iniciado pela operação de abertura de uma nova instância do browser, sendo efetuado logo em seguida o Login do utilizador configurado. Em seguida, na segunda linha, é efetuada a abertura da listagem de relatórios de despesa que se encontrem no estado completo, sendo aberto para edição o primeiro relatório disponível.

Em seguida, na terceira linha, é realizada a operação de reversão do documento, através do clique do botão de reversão presente na interação. Através desta operação, é lançado um comando, sendo esperado que o mesmo seja concluído com erro, visto a operação pedida não ser permitida. Na quarta linha é avaliado se o comando foi concluído com erro, sendo o teste passado com sucesso caso isso se verifique.

-Feature 5: Como gestor da conta, devo conseguir inserir um novo fornecedor

Esta *feature* está relacionada com a gestão das entidades existentes no modelo em avaliação. Para o modelo de gestão de despesas, a inserção, edição ou inativação de entidades só pode ser realizada pelo gestor da conta. Desta forma, este teste pretende avaliar se o mecanismo de permissões permite adicionar uma nova entidade, bem como validar o funcionamento do processo de adição. O teste resultante desta feature será executado sobre o cliente de UI.

Para este teste ser realizado o modelo tem de cumprir com as seguintes pré-condições:

- Tem de existir um perfil de segurança que permita gerir entidades do tipo "Fornecedor";

- Tem de existir um utilizador com um perfil de segurança adequado e um colaborador associado.

Estando estas pré-condições asseguradas, o teste pode ser desenvolvido e executado. A definição do mesmo em linguagem Gherkin é a seguinte:

```
Given I am in the main page
And I open Create form of an Supplier Entity
When I enter 'XPTO001' in Code
When I enter 'Xpto 001' in Name
When I press the button Submit
Then the Entity should be saved
```

Figura 18 – Gestão de Despesas: Feature 5

Este teste, tal como os anteriores, é iniciado pela operação de abertura de uma nova instância do navegador escolhido, seguindo-se o Login do utilizador configurado. Em seguida, na segunda linha, é efetuada a abertura do formulário de criação de uma entidade do tipo "Supplier", através da abertura do menu lateral, sendo escolhida a opção correta.

Em seguida, com o formulário já aberto, são executadas as linhas três e quatro do teste. Estas são responsáveis pela inserção dos dados da entidade, através da atualização dos valores dos diversos campos que compõem o formulário. Como no momento de inserção de uma interação, a indicação dos valores a inserir deve ser feita dentro de plicas, sendo o código do campo colocado no fim da frase a avaliar.

Por fim, este teste é concluído com o pressionar do botão indicado, sendo gerado um comando de gravação dos dados inseridos. Caso o comando seja terminado com sucesso, o teste é passado com sucesso.

De forma a validar se o módulo de testes identifica possíveis erros no modelo e/ou na plataforma, foi desenvolvido um segundo teste sobre esta feature, cujo propósito é verificar se, nos casos em que o teste contenha um erro, se o mesmo é identificado. Desta forma, foi realizado um teste em tudo semelhante ao presente na figura anterior, mas que refere um tipo de entidade não existente, como podemos verificar na imagem seguinte:

```
Given I am in the main page
And I open Create form of an InvSupplier Entity
When I enter 'XPTO001' in Code
When I enter 'Xpto 001' in Name
When I press the button Submit
Then the Entity should be saved
```

Figura 19 – Gestão de Despesas: Teste inválido sobre Feature 5

É esperado que este teste não termine com sucesso, sendo originada uma falha que deverá ser investigada, de forma a ser descoberta a sua causa.

-Feature 6: Como utilizador, devo conseguir validar se a informação inserida numa Interação se encontra correta

Esta *feature* tem como objetivo a validação da integridade dos dados inseridos. Numa interação, muitos dos dados são calculados automaticamente recorrendo à informação que o utilizador insere. Desta forma, revela-se útil validar se a gravação dos dados inseridos pelo utilizador e os calculados se encontram de acordo com o esperado. De forma a validar a integridade dos dados, foi desenvolvido um teste que valida se qualquer utilizador que tenha efetuado login no sistema consegue validar os dados de uma entidade, de forma a verificar a sua integridade. Ao contrário dos testes anteriores, este é executado diretamente sobre a camada de API, visto o acesso aos dados pretendidos ser bastante mais fácil do que através do cliente de UI.

Para este teste ser realizado o modelo tem de cumprir com as seguintes pré-condições:

- Tem de existir um perfil de segurança que permita a edição de Interações do tipo “ExpenseReport”;

- Tem de existir uma Interação do tipo “ExpenseReport” com um determinado identificador;

- Tem de existir um utilizador com acesso à conta que contém o modelo de Gestão de Despesas e um perfil de segurança adequado.

Estando estas pré-condições asseguradas, o teste pode ser desenvolvido e executado. A definição do mesmo em linguagem Gherkin é a seguinte:

```
Given I am logged in
And I have an ExpenseReport Interaction with ID '12345'
When I get its Currency
Then Currency value should be 'EUR'
```

Figura 20 – Gestão de Despesas: Feature 6

Este teste, apesar de ser executado sobre a camada de API, é iniciado da mesma forma dos testes ao cliente de UI. No entanto, como estamos perante um ambiente diferente, a avaliação da primeira frase traduz-se numa ação distinta da executada nos testes descritos até agora, procedendo-se ao login do utilizador configurado através do uso de pedidos HTTP.

Por sua vez, na segunda linha, é realizada a obtenção da interação do tipo "ExpenseReport", com o ID que se encontra dentro de plicas. A obtenção do relatório de despesas é realizada através de um pedido GET à API, composto pela informação inserida na linha do teste. O resultado deste pedido são os dados da interação, devolvidos como uma string em formato JSON, sendo esta armazenada para posterior utilização.

Por fim, na terceira linha é obtido o valor de uma propriedade do relatório de despesas, sendo na quarta linha efetuada a validação se o valor se encontra dentro do esperado. Caso esteja, o teste é concluído com sucesso.

5.2. Gestão de Encomendas: Cenários de teste

No âmbito dos testes ao módulo de gestão de encomendas, e de forma a validar a capacidade da framework de testes em testar múltiplos modelos, foram identificadas duas features que, além de relevantes para o bom funcionamento do modelo, se enquadram nos casos previstos nesta proposta inicial da framework de testes.

- **Feature 1:** Como utilizador do sistema, necessito de registar novas encomendas

O registo de uma encomenda é uma operação essencial no modelo de gestão de encomendas, sendo o passo inicial de um workflow que termina com a entrega da encomenda ao cliente. Esta feature, identificada numa fase inicial do desenvolvimento do modelo, deverá resultar no desenvolvimento de um teste que valida se qualquer utilizador que tenha efetuado login no sistema consegue efetuar a inserção de encomendas, permitindo despistar possíveis erros na

modelação da interação, bem como limitações da framework. O teste resultante é executado sobre o cliente de UI, visto efetuar a representação de uma ação que os utilizadores executam pela user interface.

É de realçar que este o teste resultante desta feature é focado na criação de uma interação, tal como o teste resultante da feature 1 do modelo de Gestão de Despesas. Esta identificação de features similares entre os dois modelos foi intencional, visto permitir testar se a solução encontrada consegue executar testes de vários modelos diferentes que se focam nas mesmas funcionalidades.

Para este teste ser realizado o modelo tem de cumprir com as seguintes pré-condições:

- Tem de existir um perfil de segurança que permita a inserção de Encomendas;

- Tem de existir um utilizador com acesso à conta que contém o modelo de Gestão de Despesas e um perfil de segurança adequado.

Estando estas pré-condições asseguradas, o teste pode ser desenvolvido e executado. A definição do mesmo em linguagem Gherkin é a seguinte:

```
Given I am in the main page
And I open Create form of an Order Interaction
When I enter 'Company' in CompanyCode
When I enter a line with 'PROD001' Resource,with Quantity '20' and ReceiverAgent 'CUST001'
When I press the button Save
Then the Document should be saved
```

Figura 21 – Gestão de Encomendas: Feature 1

Na definição do teste acima, podemos verificar que, tal como na Feature 1 do modelo de gestão de despesas, as duas primeiras linhas referem-se à avaliação das pré-condições, nomeadamente o login do utilizador e o acesso ao formulário de criação da interação de Encomendas.

Por sua vez, na terceira linha é realizada a manipulação do campo “CompanyCode” do cabeçalho do documento, sendo inserido o valor que se encontra dentro de plicas. Em seguida, na quarta linha do teste, é realizada a adição de uma nova linha na grelha com os valores indicados.

Por fim, as linhas 5 e 6 referem-se à gravação do documento, através do clique do botão “Save”, sendo em seguida avaliado o resultado do comando. O teste é concluído com sucesso mediante o estado final do comando.

-Feature 2: Como utilizador, devo conseguir inserir um novo Cliente

Esta *feature*, relacionada com a gestão de entidades de uma conta, é essencial para o funcionamento das interações do modelo, que dependem sempre da existência de Clientes no sistema. De forma a garantir o funcionamento do modelo, foi desenvolvido um teste que valida se qualquer utilizador que tenha efetuado login no sistema consegue introduzir novos clientes, permitindo despistar erros na modelação ou na plataforma. É de realçar que esta feature é similar à Feature 5 do modelo de Gestão de Despesas, visto as duas permitirem a inserção de novas entidades. O teste resultante será executado sobre a camada de UI.

Para este teste ser realizado o modelo tem de cumprir com as seguintes pré-condições:

- Tem de existir um perfil de segurança que permita gerir entidades do tipo "Cliente";

- Tem de existir um utilizador com acesso à conta que contém o modelo de Gestão de Despesas e um perfil de segurança adequado.

Estando estas pré-condições asseguradas, o teste pode ser desenvolvido e executado. A definição do mesmo em linguagem Gherkin é a seguinte:

```
Given I am in the main page
And I open Create form of an Customer Entity
When I enter 'XPTO001' in Code
When I enter 'Xpto 001' in Name
When I press the button Submit
Then the Entity should be saved
```

Figura 22 – Gestão de Encomendas: Feature 2

Este teste é iniciado pela operação de Login, seguindo-se a abertura da página de criação de um novo Cliente. Em seguida são preenchidos os campos obrigatórios e efetuada a gravação da informação. Por fim, é feita a validação do resultado do comando, sendo o teste concluído com sucesso caso se verifique que o comando foi completo.

-Feature 3: Como utilizador, devo conseguir validar se a informação inserida numa Entidade se encontra correta

Esta *feature* tem como objetivo a validação da integridade dos dados inseridos. Numa entidade, é possível que determinados dados sejam calculados automaticamente recorrendo à informação que o utilizador insere. Desta forma, revela-se útil validar se a gravação dos dados inseridos pelo utilizador e os calculados se encontra de acordo com o esperado. De forma a validar a integridade dos dados, foi desenvolvido um teste que valida se qualquer utilizador que tenha efetuado login no sistema consegue validar os dados de uma entidade, de forma a

verificar a sua integridade. Esta feature pode ser descrita como equivalente à Feature 6 do modelo de Gestão de Despesas, sendo o teste gerado também executado sobre a camada de API.

Para este teste ser realizado o modelo tem de cumprir com as seguintes pré-condições:

- Tem de existir um perfil de segurança que permita gerir entidades do tipo “Cliente”;

- Tem de existir uma entidade do tipo “Cliente” com o código “XPTO001”;

- Tem de existir um utilizador com acesso à conta que contém o modelo de Gestão de Despesas e um perfil de segurança adequado.

Estando estas pré-condições asseguradas, o teste pode ser desenvolvido e executado. A definição do mesmo em linguagem Gherkin é a seguinte:

```
Given I am logged in
And I have a Client with Code 'XPTO001'
When I get its Name
Then Name value should be 'Xpto 001'
```

Figura 23 – Gestão de Encomendas: Feature 3

Este teste é conforme mencionado anteriormente, semelhante ao teste desenvolvido para a Feature 6 do módulo de gestão de despesas, sendo as únicas diferenças relativas à informação específica do modelo, como o tipo de Entidade e os campos manipulados. Desta forma, para este teste ser executado com sucesso o valor persistido deverá ser semelhante ao indicado na quarta linha do teste.

6. Resultados dos testes

Neste capítulo será realizada a descrição dos resultados dos testes efetuados à solução proposta. O conjunto de testes executado foi o descrito no capítulo anterior, sendo o ambiente utilizado para a execução dos mesmos a máquina local. No entanto, os tenants com os dois modelos utilizados já se encontram no servidor de produção.

É de realçar que o ambiente de execução dos testes não é o mais apropriado, visto ainda não ser representativo de uma situação real. Conforme descrito na proposta de solução, o ideal seria que a execução fosse realizada num servidor de testes, permitindo que os mesmos fossem executados quando uma nova versão da plataforma fosse colocada, ou a pedido quando fossem efetuadas alterações num ou mais modelos. Apesar da execução na máquina local não ser a mais indicada, já nos permitiu avaliar se a aplicação desenvolvida para a execução dos testes cumpre com o esperado.

É esperado que os testes realizados permitam concluir se a aplicação desenvolvida cumpra com determinados requisitos. Os requisitos identificados para uma primeira versão do módulo de testes passam por este permitir a execução de testes sobre a camada de API e sobre o cliente de UI. Cumpridos estes dois requisitos, é esperado que o módulo desenvolvido permita a execução de testes sobre todos os modelos válidos perante a plataforma myMIS, sem a necessidade de desenvolver código adicional.

6.1. Camada de UI

A execução dos testes descritos sobre o cliente de UI permitiu verificar que a solução proposta consegue reproduzir as ações dos utilizadores com sucesso nos navegadores com maior quota de mercado (Internet Explorer, Firefox e Chrome).

Conforme descrito no capítulo 5, os testes executados foram desenvolvidos para dois modelos distintos, o modelo de Gestão de Encomendas e o de Gestão de Despesas sendo que, alguns dos testes entre os dois modelos executam tarefas semelhantes, como a inserção de uma interação ou de uma entidade. A escolha de features que originam testes semelhantes foi propositada, visto permitir-nos validar se a solução apresentada permite a execução de testes a diferentes modelos sem a necessidade de adicionar código específico.

Através da execução dos testes desenvolvidos para as *features* 1 e 5 do modelo de Gestão de despesas e dos testes para as *features* 1 e 2 do modelo de Gestão de encomendas, conseguimos validar que o módulo de testes proposto permite a interpretação de testes entre *features* semelhantes de modelos distintos.

Durante os testes a esta camada foi avaliado não só o sucesso dos testes, mas também outros critérios como o desempenho, eficácia e a reprodução contínua dos mesmos. Foi possível verificar, em pouco tempo, que os testes, dependendo da sua natureza, podem ser reproduzidos continuamente sobre o mesmo tenant sem a ocorrência de perdas de performance significativas. De forma a avaliar a eficácia dos testes, foram realizados dois procedimentos distintos: a escrita e execução de testes inválidos e alterações ao modelo. Para o caso do primeiro procedimento, foi feita a introdução de testes às features 1 e 5 do modelo de Gestão de Encomendas com informação inválida para o modelo em causa, de forma a validar se a solução apresentada conseguia identificar a existência de um erro. Após a execução dos mesmos foi possível validar que os testes não chegaram ao fim com sucesso, tendo sido identificado um erro que teria de ser investigado pela equipa de desenvolvimento. No caso da alteração do modelo de forma a tornar os testes existentes inválidos, verificou-se o mesmo comportamento.

Por fim, a validação da performance permitiu identificar algumas limitações, tendo sido registada uma média de execução elevada por cada teste. Como exemplo, no caso concreto do teste à feature 1, na qual é realizada a inserção de uma interação, verificou-se uma média de execução do teste de 50 segundos. Esta média elevada é justificada por vários fatores, nomeadamente a dependência dos testes de uma conexão à internet e o fato de a gravação de interações na plataforma myMIS ser realizada de forma assíncrona, sendo que este último fator contribui para alguma disparidade na medição dos tempos, visto que quanto maior a carga sobre o sistema, maior o tempo de gravação do documento.

Na tabela seguinte são enumerados os tempos obtidos (em segundos) na execução dos testes desenvolvidos para cada feature. Os tempos medidos foram obtidos após 10 execuções de cada teste no navegador Chrome.

Tabela 1 – Tempos de execução no UI

Modelo	***Feature***	**Tempo mínimo**	**Média**	**Tempo Máximo**
Despesas	1	47.7	50.2	52.9
Despesas	2	59.5	65.1	68.8
Despesas	3	19.6	20.4	20.9
Despesas	4	23.2	25.9	27.3
Despesas	5	20.9	21.9	22.5
Encomendas	1	59.2	63.8	66.7
Encomendas	2	20.8	22.3	23.3

6.2. Camada de API

Por sua vez, a execução dos testes sobre a camada de API permitiu verificar que a solução proposta consegue reproduzir as ações definidas em cada passo dos scripts de teste. Foi também possível verificar que a solução proposta para os testes sobre a camada de API permite a execução de testes sobre vários modelos distintos. Através da execução dos testes desenvolvidos para as *features* 1 e 5 do modelo de Gestão de despesas e dos testes para as *features* 1 e 2 do modelo de Gestão de encomendas, conseguimos validar que o módulo de testes proposto permite a interpretação de testes entre *features* semelhantes de modelos distintos.

Durante os testes a esta camada foi, tal como para a camada de UI, avaliado o sucesso dos testes e também outros critérios como o desempenho, eficácia e a reprodução contínua dos mesmos. Foi possível verificar, em pouco tempo, que os testes podem ser reproduzidos continuamente sobre o mesmo tenant sem a ocorrência de quaisquer perdas de performance. Para a avaliação da eficácia dos testes, foram realizadas alterações aos mesmos, que se traduziram em chamadas aos serviços da camada de API com parâmetros inválidos. Estes testes traduziram-se na identificação de erros, conforme esperado.

Durante os testes à camada de API foi feito o registo de tempos, que permitiu validar que os mesmos aparentam ser mais rápidos do que os testes realizados sobre a camada de UI. Este fato é justificável pela diferença da natureza dos testes realizados. Enquanto que na camada da UI os testes são maioritariamente inserções de dados, que passam por um mecanismo assíncrono, os testes à API caracterizam-se por serem exclusivamente consultas de informação realizadas de forma síncrona, um processo bastante mais rápido. Foi também possível verificar que existe uma menor discrepância de tempos entre as várias execuções do teste realizadas, pelos motivos descritos anteriormente.

Tabela 2 – Tempos de execução na API

Modelo	***Feature***	**Tempo mínimo**	**Média**	**Tempo Máximo**
Despesas	6	12.1	12.2	12.4
Encomendas	3	12.2	12.4	12.7

7. Conclusões

Neste capítulo será realizada a conclusão desta tese, através da descrição do trabalho realizado e dos objetivos cumpridos. Será também realizada a avaliação pessoal do trabalho realizado, bem como a descrição de trabalho a realizar no futuro.

7.1. Trabalho e Objetivos Realizados

Neste subcapítulo será descrito o trabalho realizado no âmbito desta tese no âmbito dos objetivos enumerados inicialmente. Será também avaliado o grau de cumprimento dos objetivos definidos.

O primeiro objetivo passava pela identificação da metodologia de desenvolvimento de software mais adequada para a plataforma myMIS. O estudo realizado teve sempre em conta a importância da inclusão de testes no ciclo de desenvolvimento, tendo sido escolhida a metodologia BDD pela sua definição clara das unidades de teste, bem como pelo envolvimento das várias partes interessadas no desenvolvimento, através da utilização de uma DSL percetível. Desta forma, consideramos que este objetivo foi cumprido com sucesso.

O objetivo seguinte passava pela elaboração de uma proposta de um módulo de testes que permitisse a validação da integridade da plataforma após a realização de alterações, bem como a validade dos modelos perante a plataforma após serem realizadas alterações nos mesmos. O módulo de testes resultante da investigação efetuada permite a realização dos mesmos recorrendo a uma DSL de fácil leitura (Gherkin), podendo os mesmos ser executados sobre a camada de UI ou sobre a API. Esta divisão dos testes sobre as duas camadas revelou-se necessário, de forma a manter a dimensão dos testes reduzida e, consequentemente, de fácil desenvolvimento e interpretação. Os testes efetuados sobre este módulo permitiram validar que o mesmo vai de encontro ao objetivo definido, sendo possível a execução de vários testes

sobre diferentes modelos com a utilização do mesmo código. Desta forma, pode ser considerado que este objetivo foi cumprido.

Por fim, o último objetivo recaia sobre a definição da forma de integração das práticas de testes no ciclo de desenvolvimento de software. Após identificar que a metodologia de desenvolvimento Continuous Integration era a mais adequada para a equipa, devido às metodologias ágeis seguidas, foram definidas algumas práticas a aplicar de forma a integrar o desenvolvimento e a execução de testes no dia-a-dia, tal como a automatização da execução dos testes, e a necessidade de executar os mesmos num ambiente de testes em tudo semelhante ao ambiente de produção. No entanto, é importante realçar que não foi possível, no âmbito desta tese, testar a validade da aplicação destas práticas. Desta forma, este objetivo não estará cumprido na sua totalidade.

7.2. Trabalho Futuro

Após os testes sobre o trabalho realizado, foram identificadas algumas questões que poderão dar origem a trabalho futuro, que permitirá uma maior adequação da solução encontrada, e uma maior integração nos ciclos de desenvolvimento. As questões são as seguintes:

- Interpretação de mais funcionalidades do modelo: Conforme descrito nesta tese, a proposta apresentada permite apenas a realização de testes sobre as operações básicas efetuadas entre interações e entidades. Deverá ser adicionado suporte para outras funcionalidades da framework myMIS, estando já identificadas as seguintes áreas:
    - Validação da integridade do mecanismo de contabilidade da plataforma myMIS;
    - Validação da integração de dados em sistemas externos;

- Realização de testes sobre as metodologias propostas de automatização de execução dos testes: Conforme descrito no subcapítulo 6.1. não foi possível, no âmbito desta tese, avaliar a integração do módulo de testes no ciclo de desenvolvimento de software e dos modelos. Desta forma, será necessária a avaliação das mesmas, de forma a permitir adequá-las às necessidades da plataforma myMIS.

8. Referências

Adzic, G. (08 de 2014). *Specification by Example*. Obtido de Specification by Example: http://www.specificationbyexample.com/

AT. (Agosto de 2014). *SAF-T PT (Standard Audit File for Tax purposes) - Versão Portuguesa*. Obtido de Portal das Finanças: http://info.portaldasfinancas.gov.pt/pt/apoio_contribuinte/NEWS_SAF-T_PT.htm

Beltran, A. (2013). *Getting Started with PhantomJS.* Packt Publishing.

Bezemer, C.-P., & Zaidman, A. (2010). *Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare?*

Chasins, S., & Phothilimthana, P. M. (22 de 01 de 2014). A Framework for Parallelizing Large-Scale, DOM-Interacting Web Experiments.

Chelimsky, D. (2010). *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends.* The Pragmatic Programmers.

Cucumber. (2013). *Given When Then*. Obtido em Março de 2014, de GitHub: https://github.com/cucumber/cucumber/wiki/Given-When-Then

Cucumber. (2014). *Gherkin*. Obtido em Fevereiro de 2014, de Github: https://github.com/cucumber/cucumber/wiki/Gherkin

Fowler, M. (15 de Dezembro de 2008). *BusinessReadableDSL*. Obtido de Martin Fowler: http://www.martinfowler.com/bliki/BusinessReadableDSL.html

Fowler, M. (10 de Setembro de 2013). *PageObject*. Obtido em Fevereiro de 2014, de Martin Fowler: http://martinfowler.com/bliki/PageObject.html

Fowler, M. (s.d.). *Continuous Integration*. Obtido de Martin Fowler: http://www.martinfowler.com/articles/continuousIntegration.html

France, R., & Rumpe, B. (2007). *Model-driven Development of Complex Software: A Research Roadmap.*

Hansson, D. H. (2014). Keynote: Writing Software. *RailsConf 2014.* Chicago.

Hendrickson, E. (2008). Driving Development with Tests: ATDD and TDD., (p. 9).

Pugh, K. (2010). *Lean-Agile Acceptance Test-Driven-Development.* Pearson Education.

REA Technology - Technology That Understands Your Business. (2007). Obtido em 11 de 2013, de REA Technology: http://reatechnology.com/what-is-rea.html

Selic, B. (2003). The Pragmatics of Model-Driven Development. *IEEE Software*.

Solís, C., & Wang, X. (2011). A Study of the Characteristics of Behaviour Driven Development. *37th EUROMICRO Conference on Software Engineering and Advanced Applications*, (p. 5). Oulu, Finland.

Stolberg, S. (2009). Enabling Agile Testing Through Continuous Integration . *2009 Agile Conference*, (p. 6).

Yoder, J. W., & Johnson, R. (2002). The Adaptive Object-Model Architectural Style.

Anexo 1: Frases suportadas

Neste anexo vão ser enumeradas as frases suportadas nesta primeira proposta de solução. Cada frase será acompanhada do ambiente de execução no qual este é executado (UI e/ou API) e descrição da ação executada como resultado da avaliação da mesma. É de realçar que as partes dinâmicas da frase, que podem sofrer alterações mediante o modelo ou a operação executada estão identificadas pelos carateres "(.*)".

1. Pré-condições (clausulas Given)

- Given I am in the main page
  - Ambiente de Execução: UI
  - Descrição: Esta frase resulta na abertura de uma nova instância do navegador que se encontra configurado. Após a abertura da nova janela, é realizada a operação de Login com os dados definidos para o modelo em causa.

- Given I am logged in
  - Ambiente de Execução: API
  - Descrição: Esta frase resulta no Login do utilizador configurado na plataforma myMIS.

- Given I open (.*) form of an (.*) Interaction
  - Ambiente de Execução: UI
  - Descrição: Esta frase resulta na abertura de um formulário de uma interação. Na primeira parte dinâmica da frase deverá ser indicado o

tipo de formulário que se pretende abrir, sendo atualmente apenas possível abrir formulários do tipo “Create”. Na segunda parte deve ser indicado o código do tipo de interação.

- Given I open (.*) form of an (.*) Entity
  - Ambiente de Execução: UI
  - Descrição: Esta frase resulta na abertura de um formulário de uma entidade. Na primeira parte dinâmica da frase deverá ser indicado o tipo de formulário que se pretende abrir, sendo atualmente apenas possível abrir formulários do tipo “Create”. Na segunda parte deve ser indicado o código do tipo de entidade.

- Given I open list (.*) of (.*) Interaction
  - Ambiente de Execução: UI
  - Descrição: Esta frase resulta na abertura de uma listagem de uma interação. Na primeira parte dinâmica da frase deverá ser indicado o código da listagem que se pretende abrir. Na segunda parte deve ser indicado o código do tipo de entidade.
- Given I have an (.*) Interaction to (.*)
  - Ambiente de Execução: UI
  - Descrição: Esta frase resulta na abertura de uma interação para edição. Na primeira parte dinâmica da frase deverá ser indicado o código do tipo de interação que se pretende aprovar. Na segunda parte deve ser indicada a operação em causa (atualmente só é suportada a edição).

- Given I have an (.*) Interaction with ID (.*)
  - Ambiente de Execução: API
  - Descrição: Esta frase resulta na obtenção de uma interação, através da chamada dos serviços da API. Na primeira parte dinâmica da frase deverá ser indicado o código do tipo de interação que se pretende aprovar. Na segunda parte deve ser indicado, dentro de plicas, o ID da interação a obter.

- Given I have a (.*) Interaction in approval state (.*)
  - Ambiente de Execução: UI
  - Descrição: Esta frase resulta na abertura de uma interação num determinado estado de aprovação para edição. Na primeira parte dinâmica da frase deverá ser indicado o código do tipo de interação que se pretende aprovar. Na segunda parte deve ser indicado qual o

estado de aprovação em que a interação deve estar (exemplo:”ManagerApproval”).

2. Ações/input do utilizador (clausulas When)

- When I change its status to (.*)
  - Ambiente de Execução: UI
  - Descrição: Esta frase resulta na alteração do estado de aprovação de uma entidade ou interação. De forma a conseguir efetuar a alteração de estado, no momento de avaliação desta frase devemos estar posicionados no formulário de aprovação da entidade/interação. Na parte dinâmica da frase deverá ser indicado qual o estado que se pretende colocar, sendo os únicos valores possíveis a aprovação (“Approve”) ou a rejeição (“Rejected”). No entanto, caso a plataforma passe a suportar novos estados, não será necessária a alteração do módulo de testes.

- When I add a Approval Note saying (.*)
  - Ambiente de Execução: UI
  - Descrição: Esta frase resulta na adição de uma nota de aprovação à entidade/interação. De forma a conseguir efetuar esta adição, no momento de avaliação desta frase devemos estar posicionados no formulário de aprovação da entidade/interação. Na parte dinâmica da frase deverá ser indicado qual o texto de aprovação/rejeição a inserir.

- When I enter (.*) in (.*)
  - Ambiente de Execução: UI
  - Descrição: Esta frase resulta na inserção de um valor num campo do formulário, sendo que, no caso de uma interação, o campo terá de se encontrar no cabeçalho. De forma a conseguir interpretar esta frase com sucesso, no momento de avaliação desta frase devemos estar posicionados no formulário de inserção, edição ou aprovação de uma entidade/interação. Na primeira parte dinâmica da frase deverá ser indicado qual o valor a inserir (entre plicas), sendo na segunda parte definido o código do elemento no qual o valor vai ser inserido.

- When I enter a line with (.*)
  - Ambiente de Execução: UI

- Descrição: Esta frase resulta na inserção de uma nova linha no formulário de uma interação com valores nos diversos campos indicados. Na parte dinâmica da frase deverão ser indicados os valores a inserir nos diversos atributos da mesma. Os valores deverão ser inseridos dentro de plicas, sendo os campos mencionados em seguida (exemplo 'X001' in Resource).

- When I press the button (.*)
  - Ambiente de Execução: UI
  - Descrição: Esta frase resulta no pressionar de um botão existente no ecrã atual da aplicação. A identificação do botão em causa deverá ser realizada na parte dinâmica da frase

- When I try to revert the approval
  - Ambiente de Execução: UI
  - Descrição: Esta frase resulta no lançamento de um comando que irá fazer a reversão da última aprovação efetuada sobre uma interação ou entidade.

- When I get its (.*)
  - Ambiente de Execução: API
  - Descrição: Esta frase resulta na obtenção de um valor da entidade ou interação em causa. O comportamento é diferente mediante a execução do teste seja na camada de API ou na camada de UI, sendo que, no primeiro caso é realizada a leitura do objeto JSON que representa a entidade/interação e no segundo caso a obtenção do valor é realizada a partir da página web. O código da propriedade deve ser indicado na parte dinâmica da frase.

3. Avaliação do resultado (clausulas Then)

- Then (.*) value should be (.*)
  - Ambiente de Execução: UI/API
  - Descrição: Esta frase resulta na avaliação do valor de um determinado campo de uma entidade ou do cabeçalho de uma interação. Esta frase possui duas partes dinâmicas, sendo a primeira a indicação do campo cujo valor pretendemos obter, e a segunda o valor esperado, dentro de plicas.

- Then the (.*) should (.*)
  - Ambiente de Execução: UI/API

- Descrição: Esta frase resulta na avaliação do estado de um comando. Esta frase possui duas partes dinâmicas, sendo a primeira a indicação do tipo de entidade ou documento (irrelevante na implementação atual), sendo na segunda parte definido o valor do comando. Atualmente é realizada a validação se o comando terminou ou não com sucesso.