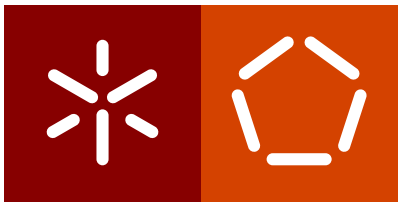
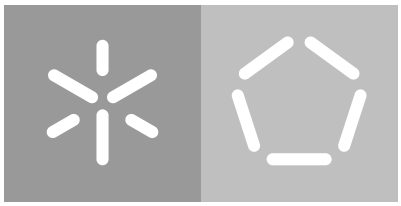


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nelson Arieira Parente

Sistema de Apoio à Decisão baseado em regras para Pricing

October 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nelson Arieira Parente

Sistema de Apoio à Decisão baseado em regras para Pricing

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Professor Doutor José Manuel Ferreira Machado

October 2019

AGRADECIMENTOS

Antes de mais gostaria de agradecer e dedicar este projeto de investigação aos meus pais por todo o apoio dado ao longo destes anos, por me terem transmitido os valores segundo os quais me guio e acredito. Sem o seu apoio não teria sido possível terminar este ciclo não podendo estar mais orgulhoso e agradecido a ambos.

Agradeço particularmente ao Professor Doutor José Machado, meu orientador, por toda a ajuda e acompanhamento prestado ao longo desta caminhada.

À Ana por todo o encorajamento, apoio e carinho dado neste último ano que tornaram esta jornada possível.

Finalmente gostaria de agradecer a todos os meus colegas de trabalho, que também me acompanharam nesta fase importante, a todos um muito obrigado.

RESUMO

Desde os primórdios dos tempos que o ser humano procura optimizar e automatizar todos os processos que se apresentam como morosos e repetitivos. O processo de *pricing* de produtos nos retalhistas é um sistema complexo e que consome tempo ao processo interno dos mesmos. Este projeto de investigação tem como objectivo o desenvolvimento de uma plataforma que seja capaz de optimizar e automatizar o processo de *pricing* de produtos, tendo como base um modelo baseado em múltiplas regras.

Com o aumento do volume de vendas online, surgem, no âmbito do processo de *pricing*, diversos problemas relativos aos processos associados à expansão de uma plataforma de *E-commerce*. Aparece assim, a necessidade de um sistema capaz de responder a estes problemas e auxiliar na eficácia, rapidez e tomada de decisão de quem lida diariamente com esta questão complexa. Assim, propõe-se o desenvolvimento de um sistema que deverá ser capaz de auxiliar a tomada de decisão na definição do preço de venda dos produtos comercializados em qualquer plataforma de *E-commerce*.

A solução que é desenvolvida ao longo deste projeto de investigação, terá que ser passível de gerir, em tempo real, o processo de *pricing* de um retalhista, bem como auxiliar na decisão da definição de preços. O foco deste projeto de investigação será dado a sistemas de apoio à decisão orientados a modelos, uma vez que é de extrema importância a versatilidade e a adaptação do sistema a múltiplos contextos e variáveis.

Como tal, e de forma a responder às questões de investigação que orientam este projeto de investigação, estrutura-se o conteúdo em quatro capítulos fundamentais: o Estado da Arte, a Metodologia de investigação e Ferramentas de desenvolvimento, o Desenvolvimento do trabalho e as Conclusões.

Durante o capítulo dedicado ao Estado da Arte abordam-se definições e conceitos essenciais ao capítulo de Desenvolvimento deste projeto, tal como o conceito de Sistemas de Apoio à Decisão, a definição do conceito de Motores de Regras e de Algoritmos de Inferência.

Para estruturar a forma como se irá conduzir este projeto de investigação, no capítulo de Metodologia e Ferramentas de Desenvolvimento, apresenta-se a metodologia de investigação e as ferramentas de desenvolvimento aplicadas neste estudo, tal como o ambiente no qual a solução final foi desenvolvida.

O capítulo de Desenvolvimento define-se pela exposição da investigação e lógica aplicada no desenvolvimento de um Sistema de apoio à Decisão para o Processo de *pricing*. Por último, o capítulo em que se expõem as conclusões deste projeto de investigação, tem como objectivo analisar os princípios teóricos que servem de base a provas de conceito, seguindo-

se pela exposição da análise SWOT. O desenvolvimento desta análise é enquadrado na metodologia de investigação definida inicialmente, *Design Research*, avaliando a solução desenvolvida de modo a perceber se os requisitos iniciais foram cumpridos.

Assim, e de forma a concluir esta investigação, e relacionar todos os conceitos abordados e tecnologias utilizadas, as questões de investigação são respondidas de forma a expor a viabilidade da solução apresentada.

Palavras-Chave: E-commerce, Sistemas de Apoio à Decisão, Arquitetura de Software, Motores de Regras, Padrões de Desenho

ABSTRACT

Since the dawn of time, human beings have sought to optimize and automate all processes that are slow and repetitive. The product pricing process for retailers is a complex system that consumes time for their internal process. This research project aims to develop a platform that is capable of optimizing and automating the product pricing process, based on a model based on multiple rules.

With the increase in online sales volume, several pricing problems arise as part of the pricing process associated with the expansion of an E-commerce platform. Thus arises the need for a system capable of responding to these problems and assisting in the effectiveness, speed and decision making of those who deal with this complex issue on a daily basis. Thus, it is proposed to develop a system that should be able to assist decision making in defining the selling price of products sold on any E-commerce platform.

The solution that is developed throughout this research project will have to be able to manage, in real time, the pricing process of a retailer, as well as assist in deciding pricing. The focus of this research project will be on model-driven decision support systems, as the versatility and adaptation of the system to multiple contexts and variables is of utmost importance.

As such, and in order to answer the research questions that guide this research project, the content is structured into four fundamental chapters: State of the Art, Research Methodology and Development Tools, Work Development and Conclusions.

The State of the Art chapter addresses definitions and concepts that are essential to the Development chapter of this project, such as the concept of Decision Support Systems, the definition of the concept of Rule Engines and Inference Algorithms.

To structure how this research project will be conducted, in the Methodology and Development Tools chapter, we present the research methodology and development tools applied in this study, as well as the environment in which the final solution was developed.

The Development chapter is defined by the research and logic applied in the development of a Decision Support System for the Pricing Process. Finally, the chapter in which the conclusions of this research project are presented aims to analyze the theoretical principles that underlie proofs of concept, followed by the presentation of a SWOT analysis. The development of this analysis is framed in the initially defined research methodology, Design Research, evaluating the developed solution in order to understand if the initial requirements were met.

Thus, in order to conclude this research, and relate all the concepts covered and technologies used, the research questions are answered in order to expose the viability of the presented solution.

Keywords: E-commerce, Decision support systems, Software Architecture, Rule engines, Design patterns

CONTEÚDO

LISTA DE FIGURAS

1

INTRODUÇÃO

1.1 ENQUADRAMENTO

Actualmente, o acesso à Internet assume uma dimensão global e a dependência desta ferramenta para nos auxiliar e apoiar em contextos tanto pessoais como profissionais é, significativamente, crescente. Com o objectivo de facilitar, simplificar e acelerar os processos que, até à disseminação da utilização da Internet, eram complexos, morosos e exigentes surgem, com o desenvolvimento exponencial desta ferramenta, vários sistemas de auxílio à pesquisa, conhecimento e tomada de decisão. Neste contexto, e como forma de acelerar um processo essencial do quotidiano e das organizações, surge uma nova forma de fazer compras.

Através de qualquer dispositivo conectado à Internet, é possível, actualmente, escolher, encomendar, pagar e receber qualquer produto presente numa plataforma de venda *online* de uma forma fácil, simples e eficaz. A este processo dá-se o nome de *E-commerce*, ou seja, a compra e venda de produtos e serviços através de uma plataforma na Internet (1). Hoje, com o aumento da utilização das plataformas *E-commerce*, surge, consequentemente, um aumento da oferta destas plataformas e a necessidade do melhoramento da rapidez e eficácia no serviço como factores de diferenciação pois, o principal objectivo de uma plataforma *E-commerce* é o aumento contínuo do volume de vendas. Para tal, é necessária a potencialização destas plataformas como resposta a um mercado cada vez mais vasto e, concorrencialmente, mais complexo.

Uma das formas das plataformas de *E-commerce* conseguirem responder aos desejos dos utilizadores, tendencialmente mais exigentes, é aumentar a gama de produtos online. Actualmente, considerando o vasto número de produtos que uma plataforma *E-commerce* poderá conter, o processo de definição do preço de venda de cada um deles e das suas variantes é um processo intrincado e de extrema importância que acarreta diversos problemas. A necessidade de uma correta definição do preço dos produtos é essencial sendo que a má execução deste processo poderá levar à insatisfação dos consumidores e, consequentemente, ao insucesso do negócio.

A definição e priorização dos custos associados a cada produto, nomeadamente o custo de produção, o custo de envio e/ou custo de aquisição de materiais de produção, exige que, para cada produto, seja criado um processo de *pricing* orientado e específico às suas características. Este método exige, da parte das organizações, um processo complexo de definição do preço de venda para cada um destes produtos e as suas possíveis variantes (por mercado, por exemplo). Este processo de definição do preço de venda de um produto chama-se *Pricing* (2).

Com o aumento do volume de vendas online, surgem, no âmbito do processo de *pricing*, diversos problemas relativos aos processos associados à expansão de uma plataforma de *E-commerce*. Aparece assim, a necessidade de um sistema capaz de responder a estes problemas e auxiliar na eficácia, rapidez e tomada de decisão de quem lida diariamente com esta questão complexa. Assim, propõe-se o desenvolvimento de um sistema que deverá ser capaz de auxiliar a tomada de decisão na definição do preço de venda dos produtos comercializados em qualquer plataforma de *E-commerce*. Para isto, é necessário o conhecimento do processo e da forma como é implementado, para que seja possível efectuar a modelação do problema e apresentar uma possível solução para o mesmo.

1.2 OBJETIVOS

O tema desta investigação surge como resposta a uma necessidade actual e contextualizada no âmbito das plataformas de *E-commerce*: o processo de definição de preço e das suas variantes. Como forma de orientar o desenvolvimento desta investigação, definiram-se quatro questões de investigação que surgem como pilares estruturais deste estudo:

- Q1: Quais são as vantagens da centralização do processo de *pricing* num único sistema?
- Q2: Quais são as principais características e funcionalidades que o sistema deve fornecer e quais as que fornece?
- Q3: Quais são as vantagens e desvantagens da aplicação de regras de *pricing* dinamicamente?
- Q4: De que forma um sistema de apoio à decisão pode melhorar o processo de *pricing* de um retalhista?

Para auxiliar na resposta às questões colocadas acima, e para orientar esta investigação de forma clara e objectiva, definiram-se os seguintes objectivos:

- O1: definir e modelar o processo de *pricing* (incluindo os modelos de produto e regras de inferência) para que se definam os preços dos produtos incluindo os diversos custos envolvidos no processo de produção, aquisição e expedição dos mesmos;

- O2: desenvolver um Sistema de Apoio à Decisão cujas principais funcionalidades residem na capacidade de criação, manipulação de modelos e regras de *pricing* bem como a capacidade de inferência sobre os dados;
- O3: perceber de que forma se poderá integrar este sistema numa plataforma de *E-commerce*.

1.3 ESTRUTURA DO DOCUMENTO

Para organizar e apresentar esta investigação de forma clara e contextualizada, inicia-se esta exposição com uma Introdução ao tema onde é apresentada uma vista geral sobre a pesquisa, a necessidade e os objectivos deste trabalho. Em seguida, elabora-se o Estado da Arte, onde se expõe uma pesquisa contextualizada sobre os trabalhos existentes sobre a temática aqui abordada, os processos de *Pricing*, e o estado actual destas investigações. Abordam-se, aqui, definições e conceitos essenciais à parte que se segue, o Desenvolvimento, tal como o conceito de Sistemas de Apoio à Decisão, o que são Motores de Regras e Algoritmos de Inferência. O Desenvolvimento, nesta investigação, define-se pela exposição da investigação e lógica aplicada ao desenvolvimento de um Sistema de apoio à Decisão para o Processo de *pricing*. Por último, apresenta-se a forma como este sistema pode ser integrado numa plataforma de *E-commerce* e as conclusões que advieram deste estudo.

2

ESTADO DA ARTE

2.1 SISTEMAS DE APOIO À DECISÃO

Um sistema de apoio à decisão, tal como o nome sugere, é um método que objectiva auxiliar no processo de deliberação sobre uma dada ação, considerando o estado e as características do ambiente no momento desta tomada de decisão (3). Isto é, a necessidade de recorrer a sistemas de apoio à decisão é proporcional à complexidade do problema ao qual se pretende reagir. O propósito de um sistema de apoio à decisão é diminuir a incerteza e, consequentemente, aumentar a segurança relativamente ao impacto que uma decisão pode desencadear (4). Isto significa que, para além de ser criado um contexto em que o impacto das decisões é calculado e antevisto, também se racionalizam processos que, muito frequentemente, perturbam e atrasam a produtividade dos negócios. Durante este estudo, o foco será dado a sistemas de apoio à decisão orientados a modelos, uma vez que é de extrema importância a versatilidade e a adaptação do sistema a múltiplos contextos e variáveis.

Como peças centrais dos sistemas de apoio à decisão orientados a modelos existem os próprios modelos e as regras de inferência. Os modelos conferem ao processo uma das mais importantes funcionalidades de um sistema de apoio a decisão: a possibilidade de simular, sem qualquer tipo de consequências, o impacto resultante de uma decisão. Isto permite que seja possível analisar, de uma forma aprofundada, não só o impacto da decisão mas, também, concede a possibilidade de fazer comparações entre todas as opções garantindo que seja possível tomar a decisão mais acertada (4).

Desta forma, é possível, através de uma interface simples e intuitiva conferir ao utilizador não-técnico a capacidade de manipular repetidamente diferentes contextos, abranger situações impossíveis de simular de outra forma e analisar os diferentes resultados com a finalidade de tomar a melhor decisão possível (5).

2.2 PROCESSO DE PRICING

O processo de *Pricing* é considerado como um processo extremamente relevante e poderoso na indústria do retalho (6), sendo este definido como um método complexo que procura

retratar a relação de oferta e procura. Isto é, definir o preço de um produto e as suas oscilações consoante o mercado e o contexto no qual se encontra inserido. Uma má decisão no âmbito da definição do preço de venda de um produto pode refletir-se em perca de vendas para um retalhista (7).

O tempo e o esforço exigidos, por parte da indústria do retalho, na definição dos preços dos produtos, considerando o enorme leque de produtos e as oscilações na oferta e na procura, representa um exercício complexo de ponderação e análise. Para além da definição do preço dos produtos, atendendo a custos de materiais e de produção, os recursos humanos e as despesas de transporte, os retalhistas necessitam de analisar, também, os preços praticados pela concorrência pois estes representam um grande impacto no processo de *Pricing* (8).

Assim, não é possível olhar para a definição dos preços dos produtos apenas ao nível do custo da produção por artigo mas também considerar o contexto e a situação do mercado concorrencial. É necessário optimizar este processo de definição de preço tendo em consideração todos os factores inerentes à produção dos artigos e os factores externos respeitantes a cada contexto. Desta forma, ao otimizar-se o processo de *Pricing*, será possível aos retalhistas aproveitar o potencial deste método de modo a retirar o máximo de lucro possível em cada venda (9).

O *Processo de Pricing* estrutura-se em 5 diferentes fases, ilustradas na figura 1 (9). No entanto, este processo deve ser adaptado às necessidades de cada área de negócio e às necessidades e contextos de cada empresa. Assim, o processo de *Pricing* apresenta-se como um método flexível e adequável aos contextos em que é aplicado.

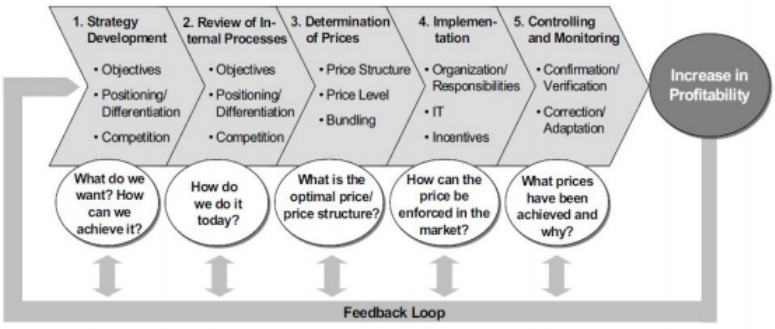


Figura 1: Fases do Processo de Pricing (10)

A primeira fase do processo de *Pricing* é a definição estratégica. Ou seja, durante esta fase deverá fazer-se um levantamento de requisitos e uma clara definição das métricas relativas aos objectivos definidos pelo negócio, nomeadamente lucro, volume, nível de preços, participação no mercado e posicionamento. A fase seguinte é uma fase de reconhecimento. Ou seja, esta etapa tem como objectivo analisar o *Processo de Pricing* existente na empresa

para que se perceba o processo actual e se definam os passos a eliminar, a manter e a melhorar. Após esta fase de reconhecimento do processo implementado, é necessária a definição dos níveis de preço que irão estar incluídos no processo de decisão de definição do preço final. Esta fase é desenvolvida através de sub-processos claros relativos à estrutura, nível e diferenciação de preços. Na penúltima fase, é definido de que forma será organizado o *Processo de Pricing* e quem será responsável pelo mesmo. Por fim, é desenvolvido o método de controlo e monitorização do processo de forma a sustentar a implementação do mesmo.

2.3 FACTORES EXTERNOS DE INFLUÊNCIA DE PREÇO

A definição dos preços de venda é dado através da implementação de uma política de preços que tem em consideração o valor de aquisição dos produtos, procura, custos, mercado e posicionamento do retalhista no mercado. Isto é, com o objectivo de adquirir vantagem competitiva, a política de definição de preços deve considerar, também, as imposições e o comportamento do mercado externo, dos consumidores e da concorrência.

Um sistema de apoio à decisão, baseado em regras, tem como objectivo conferir às organizações uma forma de poder reagir rapidamente a oscilações de mercado, a conseguir competir com a concorrência e aumentar a liquidez final. A principal vantagem da implementação de um sistema de apoio à decisão baseado em regras é a possibilidade de modelar as regras e as estratégias de negócio, facilmente adaptando o ambiente de forma a reagir a oscilações externas (11).

Desta forma, é possível identificar alguns conceitos que têm influência na definição dos preços e que podem por sua vez ser modelados através de regras dentro deste sistema.

Sensibilidade do preço

A sensibilidade do preço perante a procura mede a influência que a oscilação do preço irá ter sobre a procura do mesmo. Este conceito assume que, se todos os factores que poderão ter influência sobre um preço e a procura permanecerem constantes, a quantidade de unidades vendidas irá diminuir quando o preço aumenta e vice-versa (12).

A elasticidade do preço reflecte a sensibilidade do público-alvo face a oscilações nos preços. Ou seja, a procura é considerada elástica quando o consumidor apresenta alterações no comportamento face a variações do preço: uma leve oscilação no preço resulta numa variação grande no volume de vendas(31). Este comportamento manifesta-se normalmente em produtos que são identificados pelo consumidor como supérfluos (13).

A procura é considerada inelástica quando uma oscilação considerável no preço não provoca alterações relevantes na sua quantidade vendida, ou seja, na procura. Produtos considerados como essenciais pelo consumidor têm geralmente procura inelástica (12).

No conceito que aqui se expõe, a sensibilidade do preço, e considerando-se o cariz financeiro e a elasticidade da procura, concluímos que o aumento do preço de venda de um produto tem um efeito negativo em produtos considerados supérfluos pelo consumidor, mas observa-se o comportamento oposto caso se trate de um produto considerado como essencial.

Promoções

Até agora, na indústria do retalho, acreditava-se que o volume de vendas de um determinado produto não apresenta relação com o seu histórico de preços. Porém, estudos recentes comprovam que o volume de vendas de um determinado produto é directamente afectado pelo seu histórico e pelos valores de desconto aplicados no passado (14).

Estudos com base em pesquisa sobre o comportamento do consumidor demonstram que os consumidores avaliam os preços dos produtos tendo em conta referências internas (perceção individual do valor do produto) e valores de referência. O preço de referência, para um dado consumidor, é directamente influenciado pelo histórico dos preços passados, das promoções aplicadas a esse produto e pelo tipo de retalhista que o comercializa. Isto leva a que as promoções nos preços tenham impacto nos preços de referência e na expectativa do preço de venda aos olhos do consumidor (14).

Venda por grosso e descontos

Há alguns tipos de negócios comerciais, entre fornecedores e retalhistas, que também têm impacto no preço de venda dos produtos como, por exemplo, as vendas por grosso. Uma venda por grosso é uma venda na qual o fornecedor vende um produto em grande quantidade. Este tipo de negócio comercial tem impacto nos preços do retalhista pois, a venda em quantidades bastante grandes, leva a reduções de preço por parte do grossista e, consequentemente, descontos por parte dos retalhistas. Este tipo de descontos, por parte dos retalhistas, tem um impacto maior sobre o consumidor do que regulares mudanças de preço. Isto leva a que os retalhistas precisem de tomar, simultaneamente, decisões sobre os preços e as acções promocionais (14).

Concorrência

Segundo Moorthy (2004), um dos factores mais importantes a considerar na definição dos preços de venda é o impacto da concorrência. Os dois aspectos mais relevantes para os retalhistas no que concerne à concorrência são o preço de venda nos retalhistas concorrentes, e a sensibilidade da procura no que se refere aos preços praticados na concorrência (14).

A melhor forma de investigar a concorrência é fazer uma análise de mercado concorrencial baseada no histórico de preços. A partir desta análise, é possível definir-se preços

competitivos pois consideram-se as tendências comportamentais passadas e as variações de preço históricas da concorrência (15).

Para além dos factores acima mencionados, existem outros factores que podem, também, ser considerados por parte dos retalhistas. Como, por exemplo, os limites inferiores, superiores e as margens definidas pela concorrência. Ao considerar estes elementos, é possível estabelecer métricas do seguinte género: definir um preço 5% inferior relativamente a um concorrente especifico num dado produto (14).

O posicionamento de mercado é um dos elementos cruciais a ter em consideração na definição de preços. Para além de posicionarem a marca e o produto no mercado e na mente dos consumidores, em termos de relação qualidade/preço, é, também, um factor essencial para a diferenciação relativamente à concorrência (16). O posicionamento é definido pela marca, ou pelo retalhista, e é necessário que corresponda à identidade, missão e valores da mesma. Como tal, e sendo um dos elementos essenciais para o posicionamento da marca (tome-se como exemplo o posicionamento de luxo, premium, ou mass market, que são posicionamentos definidos pela qualidade e pelo preço dos produtos), é necessário definir uma estratégia adequada ao posicionamento da marca para que esta se estabeleça no mercado e se distinga da concorrência (17).

Preços psicológicos

Diversos estudos relativos ao comportamento do consumidor mostram que existem fenómenos psicológicos que interferem no processo de aquisição de produtos e serviços. Um destes fenómenos é a procura pelos preços baixos (14).

O preço psicológico é um conceito desenvolvido através da definição de preços terminados com algarismos ímpares (normalmente 9). A definição destes preços leva a que o consumidor acredite que está a pagar um valor inferior aquele que realmente paga. Por exemplo, um preço marcado a 5,99 em vez de 6, é visto pelo consumidor como sendo um valor mais perto dos 5 do que dos 6. Este é um fenómeno psicológico chamado de "Gestal". Este fenómeno, mais conhecido como, Gestaltismo, é uma teoria psicológica que considera estes fenómenos como um conjunto autónomo, indivisível e articulado na sua configuração, organização e lei internas (18).

2.4 ENGENHARIA DE SOFTWARE

A Engenharia de Software busca estruturar de forma racional e científica a criação do processo, colecção de métodos e ferramentas relativas à conceção sistemática de sistemas de software. Sendo esta área de conhecimento de Computação responsável, também, pela manutenção deste software ao longo do tempo. Um sistema de software é definido como o resultado de uma aplicação sistemática, disciplinada e quantificável do processo de de-

senvolvimento de software capaz de responder e cumprir os requisitos impostos. Sendo, também, capaz de agregar novos requisitos através de novos desenvolvimentos (19).

A conceção de sistemas de software é realizada por engenheiros que têm como principal objectivo construir e suportar sistemas de software com a máxima qualidade possível no menor período de tempo e custo. A metodologia adoptada na conceção de sistemas de software pode variar, sendo, normalmente, usada aquela que melhor se adapta à equipa envolvida no processo. Considerando que uma das principais prioridades já referidas é a redução do tempo e o custo de conceção, sem por isso diminuir a qualidade do sistema final, são necessários processos de Engenharia de Software para atingir esta finalidade. Actualmente, a utilização de sistemas de software é uma prática universal, seja de forma directa ou indirecta.

Apesar desta clara universalidade na criação e na utilização de sistemas de software, a definição destes sistemas continua a ser um processo iterativo. No entanto, existem determinados padrões que já foram implementados com sucesso e que devem ser aplicados de modo a aumentar as probabilidades de sucesso de um sistema de software.

Segundo Lehman, a mudança é uma característica essencial no desenvolvimento de software e os sistemas de software necessitam de responder à evolução dos requisitos, plataformas e outras pressões de ambiente (20). A esta dinâmica, em que os sistemas têm de ser capazes de evoluir ou ficam em risco de uma morte prematura (21), chama-se de Lei da Evolução de Software de Lehman.

O conceito de evolução e de manutenção são termos historicamente reconhecidos na área do desenvolvimento de software, mas apenas recentemente adaptados pela comunidade. Quanto ao conceito de evolução, entenda-se conceito como sendo sinónimo de processo, é uma área de estudo relativamente recente. Esta recente adaptação da comunidade aos processos de evolução leva a que existam ainda alguns problemas, nomeadamente a mudança de foco no estudo e a definição ambígua de conceitos. No entanto, e considerando a Lei de Evolução de Software de Lehman, é necessário que o processo de evolução se desenvolva dentro das comunidades, mesmo quando ainda não há uma clara definição do objectivo final, principalmente pelo facto de se observar, diariamente, a complexificação dos sistemas desenvolvidos. A evolução é um processo motivado pela insatisfação do uso do sistema (22).

Em oposição, o processo de manutenção define-se através da ideia de continuar a usar um sistema existente, por este continuar a fazer sentido, mas requer mudanças de design correctivas com o objectivo de eliminar bugs, adicionar novas funcionalidades ou até melhorar o seu design interno para facilitar a usabilidade do sistema.

Considerando os dois conceitos acima explorados, pode afirmar-se que o processo de evolução, como sinónimo de inovação, leva à evolução do software, resultando num novo sistema melhor adaptado ao seu ambiente (20).

Actualmente, surgem cada vez mais casos de sucesso no âmbito do desenvolvimento de software: frequentemente aparece uma nova e inovadora *start-up* com um produto inovador para apresentar ao mundo, por exemplo. Com a quantidade de lançamentos de novos produtos, a crescente investigação na área, e o aumento concorrencial, pode observar-se que o ritmo de crescimento desta área é efervescente.

Em 2018, o top cinco de empresas em Oferta Pública Inicial na bolsa de Nova Iorque era composto por três empresas tecnológicas (23). Hoje, quase todos os nossos bens e actividades estão ligados a este sector de actividade, existindo assim uma lista considerável de casos de sucesso.

Por outro lado, existem, também, várias situações em que as falhas no software resultaram em acontecimentos desastrosos. Um infeliz caso destas falhas no software foi a queda do *Ethiopian Airlines Flight ET302* que matou os 157 passageiros a bordo do voo.

Foi provado que este acidente se deu devido a uma falha de software e, consequentemente, a partir do momento em que uma falha de software tem impacto em vidas humanas levantam-se questões éticas e sociais relativamente à importância da qualidade e da fiabilidade do software. Tal como as incertezas que surgiram quanto à credibilidade do software utilizado e à empresa que o desenvolveu, surgem também dúvidas quanto à produção dos equipamentos envolvidos no acidente e a empresas que produzem equipamentos e software semelhantes (24).

Este caso, tal como outros que se apresentaram como casos de insucesso no desenvolvimento de sistemas de software, apresentam-se como motivo para melhorar, progressivamente, o processo de desenvolvimento de software. Este processo de evolução tem como objectivo final aumentar a qualidade dos sistemas de software, a contabilidade e a eficácia dos mesmos.

Em modo de consideração final sobre este tema, é necessário definir o sistema de software como um elemento tecnológico indispensável nos dias de hoje. No entanto, continua a existir uma enorme dificuldade em construir sistemas de software de elevada qualidade no tempo e orçamento disponíveis na maioria dos casos. Nos últimos 50 anos, o crescimento e a especialização na área de Engenharia de Software cresceu exponencialmente, procurando resolver este problema através da disponibilização de uma série de ferramentas e guias para a conceção de sistemas de software de elevada qualidade com o mínimo de custo possível.

Na próxima secção, iremos analisar um dos mais importantes pontos relativos à conceção de sistemas de software: a sua Arquitetura. Será analisada a evolução histórica da arquitetura de software e o impacto da mesma sobre os sistemas de software.

2.5 ARQUITETURA DE SOFTWARE

Durante décadas os designers de software desenvolveram software considerando, exclusivamente, os requisitos técnicos apresentados. Desta forma, os requisitos técnicos definiam o design e este, por sua vez, iria acabar por definir o próprio sistema. Esta dinâmica levou a grandes falhas de design nos sistemas de software por este não ter sido planeado e estrategicamente implementado considerando tanto os requisitos técnicos como a usabilidade do software.

A arquitetura de software de um sistema é a estrutura, ou estruturas, do sistema que contém os elementos de software, propriedades externamente visíveis e as relações entre estes. Assim sendo, considera-se a arquitetura como a abstração mais elevada do próprio sistema (25).

O objectivo final de um arquitecto de software é a conceção de um sistema capaz de corresponder às expectativas e aos objectivos propostos, perante uma carga de esforço infinita, através da combinação de várias técnicas e de estilos de arquitetura diferentes. Este é considerado o Santo Graal da arquitetura de software: o ponto de escalabilidade infinita, tal como ilustrado na figura 2.

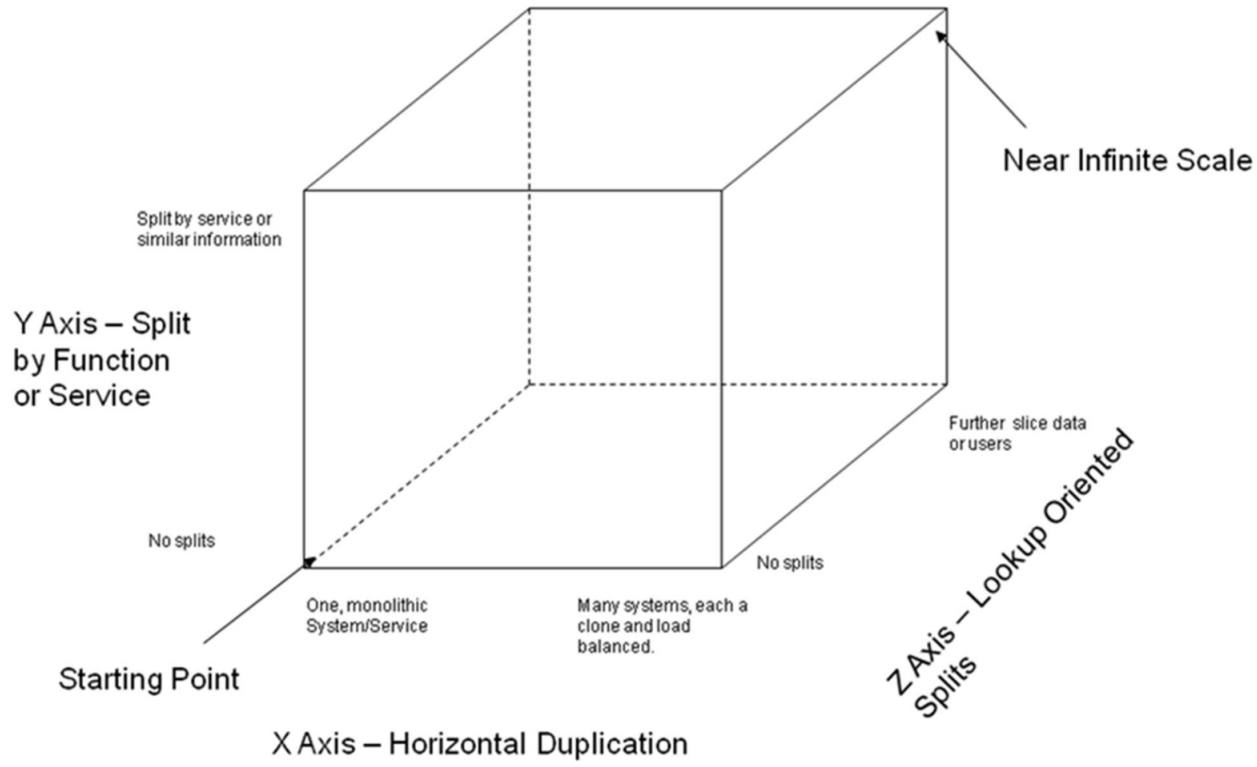


Figura 2: Ponto de Escalabilidade Infinita (25)

Cada vez mais é dada importância ao processo de definição da arquitetura de um sistema de software e, nos últimos 30 anos, a comunidade foi iterando sobre qual a melhor

arquitetura que um sistema de software deve possuir. O foco desta investigação são aplicações cliente-servidor dado que representam a maioria de sistemas de software com os quais interagimos no dia-a-dia e se apresenta como uma área de conhecimento ainda a desenvolver.

2.6 MOTORES DE REGRAS

Os motores de regras, como se pode ver na figura 3, são aplicações especializadas de inferência capazes de, a partir de um conjunto de regras definidas na sua base de conhecimento, identificar um dado modelo de entrada, inferir sobre o seu estado e debitar um resultado. São um excelente recurso para aglomerar processos de decisão e trabalhar conjuntos de dados enormes que seriam humanamente impossíveis de manipular (26).

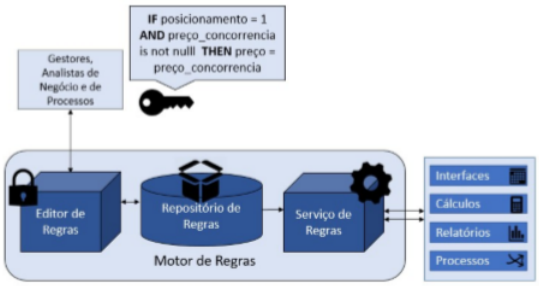


Figura 3: Modelação de um motor de regras

Considere-se como exemplo o processo de *pricing*, isto é, a definição do preço de um dado produto. O preço desse produto é dado pela conjunção de vários elementos, como, por exemplo, o custo de aquisição de materiais, custo de produção ou a sua procura, sendo que diferentes produtos incluem diferentes variáveis. Existe, assim, a necessidade de definir diferentes modelos e regras que se adaptem ao modelo de maneira a inferir correctamente sobre este e determinar o melhor preço.

Os modelos são definidos como os factos que irão ser inferidos perante regras, de forma a debitar um resultado que representa o seu preço de venda.

2.6.1 *Importância da Modelação*

Uma das partes fundamentais na implementação de um sistema de apoio à decisão orientado a modelos, é a modelação dos factos que irão ser inferidos pelas regras. Os factos representam as características do modelo. Estas características têm como principal objectivo ser manipuladas de forma a que, perante a inferência das regras, seja possível simular e

concluir face às diferentes alterações que o modelo pode sofrer. Assim sendo, é de extrema importância a modelação dos factos, quer no seu domínio quer nas regras de negócio.

2.6.2 *Algoritmos de Motores de Regras*

Nesta secção serão apresentados dois dos algoritmos de inferência mais usados em motores de regras. Os algoritmos apresentados são o *forward chaining* e o *backward chaining*, o tipo de inferência usado por estes dois algoritmos é contrária, sendo no final efectuada uma comparação entre a metodologia de execução de ambos (27).

2.6.3 *Forward Chaining*

De forma a simplificar a compreensão destes algoritmos iremos usar como analogia a metodologia de resolução de um crime por parte de um detective, segundo a linha de pensamento de (28). Consideremos um conjunto de provas presentes no local de um crime: um bastão, um fio de cabelo e um corpo. Este conjunto de provas representa o conjunto inicial sobre o qual o detective irá inferir conclusões, até encontrar ligações entre o conjunto de provas e o responsável do homicídio. Este método de investigação representa um processo de inferência *forward chaining*.

A metodologia de inferência *forward chaining* é caracterizada por ser orientada aos dados (*data-driven*) e reactiva. Num processo de inferência, as regras são representadas de forma semelhante aos operadores condicionais das linguagens de programação imperativas. A principal diferença reside no facto de, nas linguagens de programação imperativas, as declarações são executadas sequencialmente. Ou seja, no algoritmo *forward chaining*, a parte *then* só é executada quando satisfeita a parte *if*, perante os factos (dados) inseridos. Isto resulta numa execução menos determinística, sendo que quem decide a ordem de execução é o motor de regras (29).

A figura 4, ilustra o encadeamento da execução do algoritmo de inferência *forward chaining*.

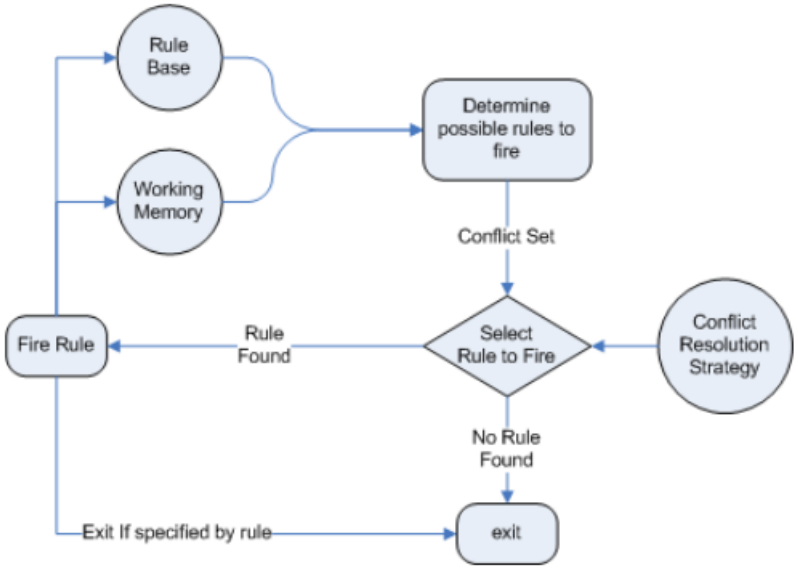


Figura 4: Forward Chaining (29)

2.6.4 *Backward Chaining*

A metodologia de inferência apresentada na secção anterior pode ser utilizada de forma inversa. Isto é, segundo (28), o responsável pelo crime podia ser descoberto a partir da formulação de hipóteses sobre o sucedido, perante a cena do crime, inferindo, posteriormente, sobre as provas que suportassem as hipóteses formuladas. A esta metodologia de inferência dá-se o nome de inferência *backward chaining* (27).

A metodologia de inferência *backward chaining* é orientada ao objectivo, ou seja, primeiro define-se a resposta e, posteriormente, procuram-se pelos factos que comprovem a mesma. Este comportamento é, geralmente, denominado de procura de objectivo (*goal seeking*). Da mesma forma que, no algoritmo de inferência *forward chaining*, também as regras são representadas como declarações *if-then*. Porém, nesta metodologia, o motor tenta chegar activamente à parte *then* procurando pelas regras (if's) que suportam a conclusão.

A figura 5, apresentada abaixo, demonstra o encadeamento da execução do algoritmo de inferência *backward chaining*.

De forma a perspectivar as principais diferenças entre os dois algoritmos de inferência apresentados na figura 5, são apresentadas duas regras aplicadas a cada um dos algoritmos. Na figura 5 é apresentada a execução dos algoritmos para as regras. O algoritmo *forward chaining* deriva **d=4** a partir dos factos **a=1** e **b=2**. O algoritmo *backward chaining* utiliza **d=4** para inferir quais as regras correspondentes que ditam os valores de **a** e **b**.

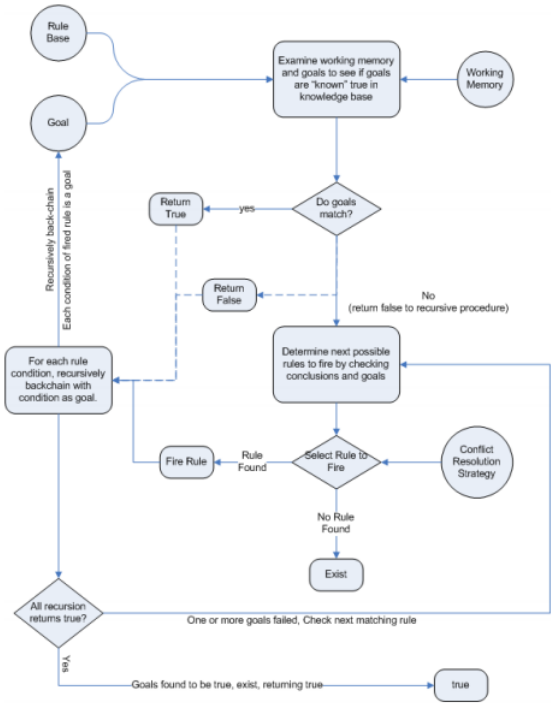


Figura 5: Backward Chaining (29)

A figura 6, para uma melhor compreensão, demonstra a diferença entre as duas metodologias apresentadas.

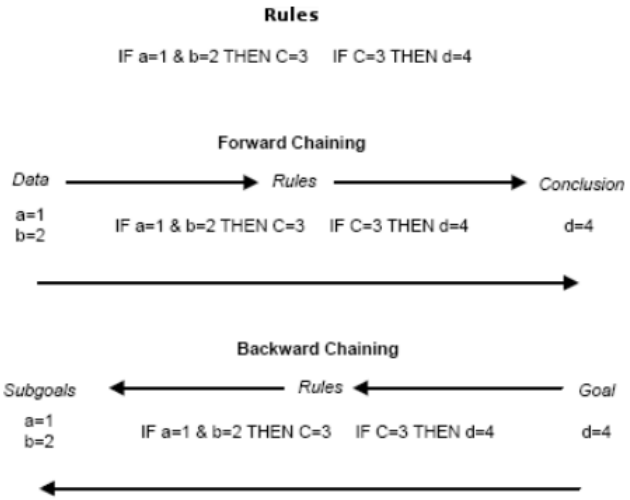


Figura 6: Diferenças entre Forward Chaining e Backward Chaining (27)

3

METODOLOGIA DE INVESTIGAÇÃO E FERRAMENTAS DE DESENVOLVIMENTO

3.1 INTRODUÇÃO

O propósito deste capítulo é apresentar a metodologia de investigação e as ferramentas de desenvolvimento aplicadas neste estudo, tal como o ambiente no qual a solução final foi desenvolvida.

Na primeira secção, é apresentada a metodologia de investigação utilizada para obter respostas às questões de investigação. Após esta fase, apresentam-se cada uma das tecnologias que englobam o ecossistema no qual a solução final se encontra inserida. Durante esta fase, efectua-se uma análise das tecnologias a utilizar a partir do conhecimento retirado tanto de investigações na área como de sistemas previamente desenvolvidos. Assim, foi seleccionado o sistema operativo Windows 10 (30), o *IDE VisualStudio 2019* (31), por ser o que melhor se adapta à plataforma *ASP.NET Core*, e *SQLServer Management* (32) para o desenvolvimento da ferramenta neste estudo apresentada. De forma a garantir a correta evolução do código-fonte, e assegurar eventuais perdas de informação será utilizado o sistema de controlo de versões *GitLab*(33).

3.2 DESIGN RESEARCH

De forma a relacionar, de maneira clara e eficaz, o desenvolvimento de produtos, serviços ou sistemas com as suas necessidades pré-existentes, a metodologia de investigação *Design Research* é considerada, nesta área de conhecimento, como sendo indispensável (34). Por este motivo, e por ser esta a metodologia de investigação que melhor se enquadra nos objectivos teórico-práticos desta dissertação, a metodologia de *Design Research* é o método aplicado a este estudo.

A metodologia *Design Research* contempla um conjunto de fases que são, nesta metodologia, indissociáveis e sequenciais, e têm como principal objectivo dar origem ao um novo sistema. O modelo geral desta metodologia apresenta cinco etapas: a consciencialização das

necessidades; a sugestão do desenvolvimento; o desenvolvimento; a avaliação e a conclusão. Ainda que, cada um deste períodos do processo esteja bem definido relativamente ao seu objectivo, é necessário que sejam adequados ao objectivo final do sistema a desenvolver (35). As etapas definidas do modelo geral do *Design Research* no âmbito desta dissertação são:

- **Consciencialização das Necessidades**: reconhecimento e formalização do problema;
- **Sugestão de Desenvolvimento**: procura e selecção de soluções para o problema apresentado;
- **Desenvolvimento**: implementação de uma ou mais soluções possíveis para o problema levantado;
- **Avaliação**: processo de avaliação do produto final no ambiente para o qual foi desenvolvido;
- **Conclusão**: esta etapa pode terminar a investigação ou iterar novamente todo o processo caso a etapa de avaliação não seja satisfatória;

Figura 7: Etapas do processo *Design Research* (35)

Durante este estudo, seguiram-se as fases da metodologia de *Design Research* como acima definidas na figura 7. No entanto, foram feitas feitas algumas alterações a este processo para que a metodologia se adequasse ao contexto e às necessidades apresentadas. Para que este processo de desenvolvimento seja, também, uma forma de aprendizagem e melhoria em todas as fases alterou-se o processo de *Design Research* de um método linear e por etapas, é um processo aberto. Ou seja, o início de um novo ciclo é iniciado sem ter sido finalizado o último. A esta abordagem designa-se *learning through building*.

3.3 PLATAFORMA ASP.NET CORE

A plataforma *ASP.NET Core* é baseada na *.NET Framework* e pode ser utilizada através de várias linguagens, nomeadamente *C*, *Visual Basic .NET* e *F*. Esta plataforma é muito popular em desenvolvimento Web sendo a mais recente iteração da plataforma *ASP.NET*.

Esta é uma ferramenta lançada em 2016, *open-source*, o seu código-fonte pode ser consultado no *Github* (36) e conta com inúmeros contribuidores de toda a parte do mundo. Esta plataforma pode ser corrida nos 3 sistemas operativos mais utilizados em todo o mundo, *macOs*, *Linux* e *Windows*.

A solução que irá suportar esta dissertação é desenvolvida em *ASP.NET Core* com a utilização de vários *Design Patterns*, nomeadamente o *Repository Pattern* e *Unit of Work*.

O padrão arquitetural usado é o *modelo de N-Camadas* seguindo os ensinamentos do *Domain Driven Design*.

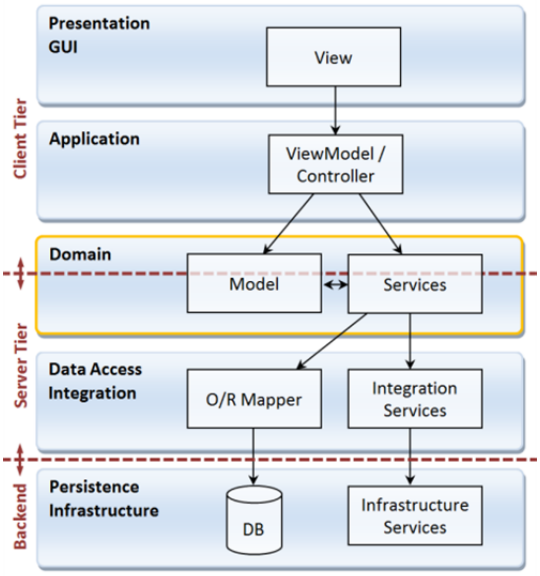


Figura 8: Arquitetura por N-Camadas (37)

Analisando a figura 8, apresentada acima, de uma perspectiva estrutural, pode observar-se que o *modelo de N-Camadas* apresenta 3 grandes camadas:

- Client Tier: Responsável pela interação com o utilizador;
- Server Tier: Responsável pela lógica de negócio existente;
- Backend Tier: Responsável pela gestão da infraestrutura utilizada;

Este padrão arquitetural permite uma maior abstração e centralização das diferentes responsabilidades nos módulos certos. Permitindo, assim, uma maior estabilidade nas manutenções necessárias no código. Devido a esta estrutura, este modelo apresenta-se como um

facilitador das alterações que sejam necessárias, independentemente da frequência com que são aplicadas, e oferece mais segurança durante o processo de manutenção do código dado que a centralização de responsabilidades leva a que uma, ou várias, alterações não afectem os módulos adjacentes.

O ambiente de desenvolvimento integrado mais comum na plataforma *ASP.NET Core* é o *Visual Studio*. Este é o ambiente utilizado para o desenvolvimento da solução, tendo sido utilizada a iteração mais recente do ambiente *Visual Studio 2019* (31).

3.4 DESIGN PATTERNS

O termo *Design Pattern* ganhou relevância no contexto da Engenharia de Software através do livro *Design Patterns: Elements of Reusable Object-Oriented Software*, publicado em 1994, por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, mais conhecidos como *"Gang of Four"* (38). A partir da teoria destes autores, define-se um *Design Pattern* como uma pequena solução de software implementada com o objectivo de resolver um problema recorrente, acelerando assim o processo de desenvolvimento de software no futuro. Os design patterns fornecem uma solução orientada ao problema a resolver e independente da tecnologia de implementação disponível no momento (39).

No livro *Design Patterns: Elements of Reusable Object-Oriented Software*, os autores apresentam 23 *design patterns*, que se dividem, através das suas características, em três grandes grupos, como podemos ver abaixo:

- Padrões de Criação;
- Padrões Estruturais;
- Padrões Comportamentais;

No âmbito desta dissertação, os 2 *design patterns* utilizados foram o *Repository e o Unit of Work*, que explicaremos na seguinte secção.

Repository e Unit of Work Patterns

Os padrões *Repository e Unit of Work* têm como principal objectivo criar uma camada de abstração entre a camada de acesso de dados e a camada de lógica de negócio, isto é, a camada de domínio. A implementação destes padrões permite isolar a solução da forma como as transacções são geridas na base de dados para facilitar a capacidade do código-fonte ser testado (40).

A classe *Unit of Work* tem como responsabilidade coordenar os múltiplos repositórios criando uma única classe de contexto de base de dados que será partilhada por todos estes repositórios.

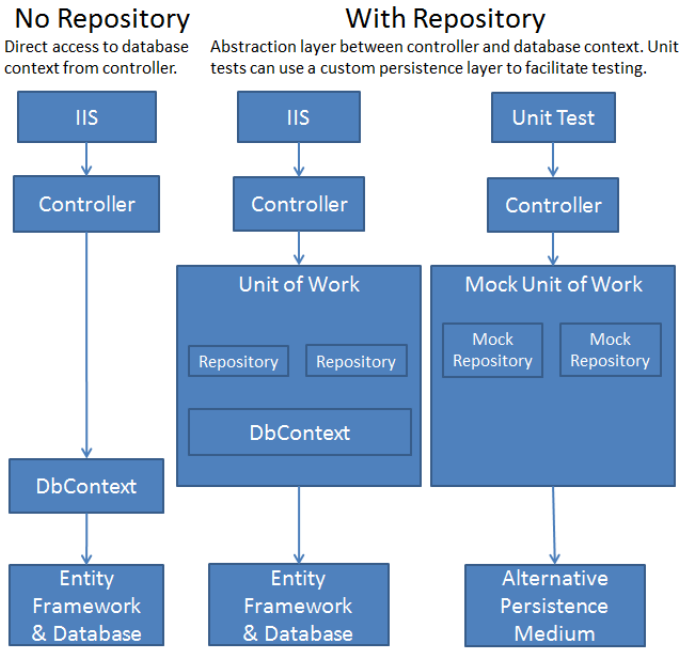


Figura 9: Comparação Repository Pattern e Unit of Work (40)

Com a utilização destes dois padrões advém várias vantagens, entre elas:

- Redução de código duplicado;
- Redução de potenciais erros na implementação;
- Centralização de responsabilidades;
- Maior capacidade de testar o código-fonte.

A implementação destes dois padrões concentra-se, principalmente, nas camadas inferiores, dado que são estas que efectuam a gestão das transacções feitas pelo sistema.

3.5 NRULES

A ferramenta NRules é um motor de regras *open source* (41) para *.NET framework* baseado no algoritmo de *matching Rete* que visa a sua utilização com base em modelos de domínio. Ao contrário da maioria dos motores de regras, que são baseados em mecanismos de *scripting*, não existe uma ordem predefinida para as regras serem executadas. Ao invés disso, o mecanismo de inferência descobre quais as regras que devem ser activadas com base nos

fatos fornecidos a este motor e, em seguida, este procede à execução de acordo com um algoritmo de resolução de conflitos (42).

Assim, para que seja possível utilizar o NRules, existe um conjunto de passos essenciais, mencionados abaixo, que são necessários para implementar a sua especificação, estes são:

Criação do modelo de domínio

```
public class Customer
{
    public string Name { get; private set; }
    public bool IsPreferred { get; set; }

    public Customer(string name)
    {
        Name = name;
    }

    public void NotifyAboutDiscount()
    {
        Console.WriteLine("Customer {0} was notified about a discount", Name);
    }
}

public class Order
{
    public int Id { get; private set; }
    public Customer Customer { get; private set; }
    public int Quantity { get; private set; }
    public double UnitPrice { get; private set; }
    public double PercentDiscount { get; private set; }
    public bool IsDiscounted { get { return PercentDiscount > 0; } }

    public double Price
    {
        get { return UnitPrice*Quantity*(1.0 - PercentDiscount/100.0); }
    }

    public Order(int id, Customer customer, int quantity, double unitPrice)
    {
        Id = id;
        Customer = customer;
        Quantity = quantity;
        UnitPrice = unitPrice;
    }

    public void ApplyDiscount(double percentDiscount)
    {
        PercentDiscount = percentDiscount;
    }
}
```

Figura 10: Exemplo de modelo de domínio (41)

Na figura 10 no modelo de domínio, temos presentes duas entidades: *Customer* e *Order*. A primeira entidade, *Customer*, é referente a um cliente que tem as suas propriedades (como o Nome, por exemplo) e pode ser notificado acerca de um desconto. A segunda entidade, a *Order*, representa a encomenda de um cliente. Esta entidade, para além das suas propriedades, define também a regra de negócio de cálculo do seu preço. Este exemplo, serve

como ilustração daquilo que poderão ser definidas como as especificações numa camada de domínio.

Criação de Regras

Uma regra consiste num conjunto de condições (padrões que correspondem a factos na memória do motor de regras) e no conjunto de acções executadas pelo mecanismo como consequência da execução dessa regra. Ou seja, se o motor de regras reconhecer um conjunto de factores que correspondam a uma regra há um conjunto de acções que irão decorrer desse reconhecimento porque assim se definiu no mecanismo de regras. Veja-se o exemplo abaixo.

```
public class PreferredCustomerDiscountRule : Rule
{
    public override void Define()
    {
        Customer customer = null;
        IEnumerable<Order> orders = null;

        When()
            .Match<Customer>(() => customer, c => c.IsPreferred)
            .Query(() => orders, x => x
                .Match<Order>(
                    o => o.Customer == customer,
                    o => o.IsOpen,
                    o => !o.IsDiscounted)
                .Collect()
                .Where(c => c.Any()));

        Then()
            .Do(ctx => ApplyDiscount(orders, 10.0))
            .Do(ctx => ctx.UpdateAll(orders));
    }

    private static void ApplyDiscount(IEnumerable<Order> orders, double discount)
    {
        foreach (var order in orders)
        {
            order.ApplyDiscount(discount);
        }
    }
}
```

Figura 11: Regra 1 (41)

Esta primeira regra, apresentada na figura 11, tem como objectivo encontrar todos os clientes preferidos, e para cada um destes aplicar um desconto de 10% a cada uma das suas encomendas. Cada correspondência encontrada no bloco *When* é armazenada numa variável que irá ser, posteriormente, usada no bloco *Then*.

```
public class DiscountNotificationRule : Rule
{
    public override void Define()
    {
        Customer customer = null;

        When()
            .Match<Customer>(() => customer)
            .Exists<Order>(o => o.Customer == customer, o => o.PercentDiscount > 0.0)

        Then()
            .Do(_ => customer.NotifyAboutDiscount());
    }
}
```

Figura 12: Regra 2 (41)

A segunda regra, presente na figura 12, procura encontrar todos os clientes que têm encomendas com descontos e enviar uma notificação. É importante que esta segunda regra dependa da primeira, dado que, desta forma, quando a primeira for accionada irá actualizar a memória do motor de regras desencadeando a segunda. Isto é execução directa de um algoritmo *forward chaining*.

Execução das Regras

```
//Load rules
var repository = new RuleRepository();
repository.Load(x => x.From(typeof(PreferredCustomerDiscountRule).Assembly));

//Compile rules
var factory = repository.Compile();

//Create a working session
var session = factory.CreateSession();

//Load domain model
var customer = new Customer("John Doe") {IsPreferred = true};
var order1 = new Order(123456, customer, 2, 25.0);
var order2 = new Order(123457, customer, 1, 100.0);

//Insert facts into rules engine's memory
session.Insert(customer);
session.Insert(order1);
session.Insert(order2);

//Start match/resolve/act cycle
session.Fire();
```

Figura 13: Execução das Regras (41)

O NRules é um mecanismo de inferência. Isto significa que não existe uma ordem pre-definida na qual as regras são executadas. Ao invés disso, este mecanismo executa um

ciclo de *match/resolve/act*. Ou seja, primeiro faz *match* dos factos (instâncias de entidades de domínio) com as regras e determina que regras estão aptas a ser activadas. Seguidamente, resolve os conflitos existentes (*resolve*, escolhendo apenas uma destas regras para activar, executando as suas acções (*act*. Este ciclo é repetido até que não haja mais regras para activar.

Para que o mecanismo entre no ciclo de *match/resolve/act*, existem pré-requisitos a ser definidos. Primeiro, é necessário carregar as regras e compilar as mesmas numa estrutura de rede *Rete*, para que o mecanismo saiba quais são as regras existentes e possa fazer *match*, de maneira eficiente, com os fatos. Após isto, compilam-se as regras numa sessão, e criando-se uma sessão de trabalho que irá inserir os fatos na memória do mecanismo. Desta forma, reúnem-se todas as condições para ser iniciado o ciclo de *match/resolve/act* (42).

3.6 COMMAND QUERY SEPARATION

O CQS, *Command Query Separation*, é um padrão arquitetural referido pela primeira vez por Bertrand Meyer (43), e apresenta a noção de separação de escritas e leituras sendo representadas, respectivamente, por *commands* e *queries*.

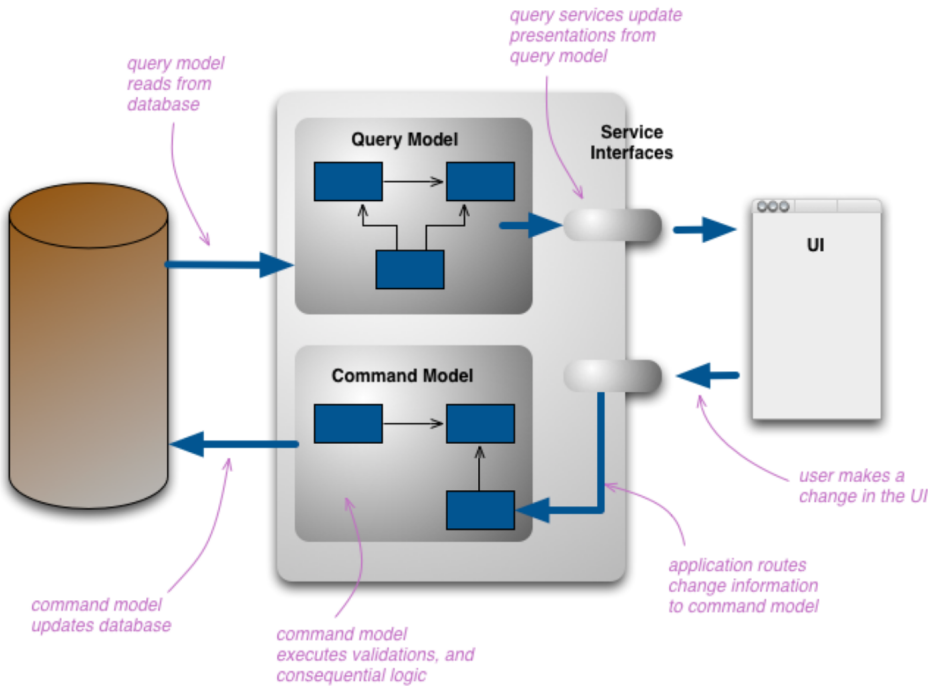


Figura 14: Command Query Separation (44)

O conceito, representado na figura 14, que serve como base a este padrão desenvolve-se a partir da propriedade de idempotência das leituras, as *queries*. Idempotência é a proprie-

dade das *queries* de poderem ser aplicadas um número infindável de vezes sem que se altere o valor do resultado após a primeira execução. Ou seja, independentemente das vezes que são efectuadas as *queries*, o estado do sistema não irá mudar, as leituras são livres de efeitos colaterais. Por outro lado, as escritas, *commands*, efectuam transacções de dados e alteram o estado do sistema, sendo necessário especial atenção relativamente à forma como estes são executados e ordenados.

3.7 REST APIS

A proposta de uma arquitetura REST foi feita no ano de 2000, na tese de doutoramento do Dr. Roy Thomas Fielding (45). Na sua tese de doutoramento, Fielding definiu uma arquitetura que objectivava melhorar a confiabilidade, visibilidade, segurança e escalabilidade de serviços Web. O resultado foi uma arquitetura consistente e estruturada, que se baseia em 5 princípios-chave:

- Cada recurso tem um identificador único (URI);
- A comunicação entre clientes e servidores é padronizada;
- Todos os recursos estão ligados entre si;
- Os recursos podem ser representados através de vários formatos;
- A comunicação é desprovida de estado.

Uma arquitetura REST tem como base o tradicional modelo cliente-servidor. Embora, a arquitetura REST, não exija a utilização de nenhum protocolo específico, o protocolo mais usado é o protocolo HTTP. Uma das principais preocupações desta estrutura é a separação de responsabilidades entre os diferentes intervenientes. Esta separação permite uma maior escalabilidade do servidor, e facilitar a portabilidade da interface do utilizador (46). No âmbito desta solução é utilizada uma API, *Application Programming Interface*, como uma das camadas de apresentação.

3.8 HTTP

O *HTTP*, *Hyper Text Transfer Protocol*, é um protocolo da camada de aplicação usado desde 1990, data do lançamento da sua primeira versão, e tem como principal objectivo permitir a comunicação entre cliente e servidor (47). Actualmente, a versão mais usada do protocolo é a versão *HTTP/1.1* (48).

A comunicação no protocolo *HTTP* é feita através de pedidos por parte de clientes que solicitam um dado recurso. Estes pedidos são recebidos pelo servidor, interpretados e, em

seguida, este servidor envia a resposta ao pedido (que poderá ser, também, um recurso ou apenas um cabeçalho). Abaixo, é apresentado um exemplo de um pedido e resposta em *HTTP/1.1* para melhor ilustrar a explicação dada.

```
Request;
GET /cars HTTP/1.1
HOST: https://carstand.api.tech

Response:
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Date: Tue, 06 Aug 2019 07:43:54 GMT

{
    "Cars": [
        { "name":"Ford", "models":[ "Fiesta", "Focus", "Mustang" ] },
        { "name":"BMW", "models":[ "320", "X3", "X5" ] },
        { "name":"Fiat", "models":[ "500", "Panda" ] }
      ]
}
```

Um pedido *HTTP* começa com um verbo, ou método. Os verbos são métodos para indicar a ação desejada a ser aplicada no recurso identificado. Existem 8 tipos de verbos *HTTP* na versão 1.1, estes são:

- **GET** : efectua o pedido de um determinado recurso;
- **POST** : submete uma entidade a um determinado recurso;
- **PUT** : actualiza o estado de um recurso alvo;
- **DELETE** : apaga um determinado recurso;
- **HEAD** : efectua o pedido de um determinado recurso como o GET, mas não traz o corpo;
- **TRACE** : efectua um teste ao caminho de um determinado recurso;
- **OPTIONS** : descreve as opções de comunicação de um dado recurso;
- **CONNECT** : estabelece uma conexão com um determinado recurso;

Após o verbo *http* do pedido, define-se o identificador do recurso e a versão do protocolo a utilizar. Um pedido poderá, também, conter campos de cabeçalho e de corpo de modo a suplementar ou trocar informação.

O servidor, após receber um pedido, procede à interpretação e processamento do mesmo, retornando uma resposta *HTTP* composta pela versão do protocolo utilizada, código de resposta e significado (a resposta poderá incluir mais campos como cabeçalhos e corpo). O código de resposta é de elevada importância, pois indica qual o resultado do pedido. Este código de resposta é composto por 3 algarismos, dos quais o primeiro indica a gama da resposta e os dois restantes a natureza da resposta. As respostas *HTTP* encontram-se dentro dos seguintes intervalos numéricos:

- **100-199** : Informação. Pedido recebido e interpretado com sucesso mas não processado.
- **200-299** : Sucesso. Pedido recebido, interpretado e processado com sucesso.
- **300-399** : Redireccionamento. Informa que são precisas acções adicionais para terminar o pedido.
- **400-499** : Erro Cliente. Indica que ocorreu um erro do lado do cliente.
- **500-599** : Erro Servidor. Indica que ocorreu um erro do lado do servidor.

Da gama de respostas possíveis em *HTTP/1.1* acima apresentadas, existe uma série de códigos de resposta que são usados mais frequentemente. Em seguida, detalhamos esses códigos de resposta para um melhor entendimento de cada um deles (49).

200 OK

O pedido foi bem sucedido. O significado de um *OK* varia dependendo do método *HTTP*:

- GET : O recurso foi obtido e é transmitido no corpo da mensagem.
- POST/PUT : O recurso que descreve o resultado da ação é transmitido no corpo da mensagem.

204 NO CONTENT

Não há conteúdo para enviar para este pedido, mas os cabeçalhos podem ser úteis. O *user-agent* pode actualizar os seus cabeçalhos em *cache* para este recurso com os novos cabeçalhos.

400 BAD REQUEST

Esta resposta significa que o servidor não consegue interpretar o pedido devido a sintaxe inválida.

404 NOT FOUND

O servidor não consegue encontrar o recurso solicitado. No *browser*, isso significa que o URL não é reconhecido. Caso o pedido seja feito a uma *API* pode significar que o *endpoint* é válido, mas o recurso não existe.

500 Internal Server Error

O servidor encontrou um pedido ao qual não sabe responder.

3.9 JSON

O *JSON*, *JavaScript Object Notation*, é uma notação de formato leve usada para troca de informação e serialização de dados (50). Esta notação é baseada no conceito de *chave-valor*, sendo, actualmente, o formato mais usado para este género de operações.

Esta linguagem de notação apresenta-se dividida entre dados primitivos e estruturas de dados. Assim, existem quatro tipos de dados primitivos, *boolean*, *string*, *number* e *null*, e dois tipos de estruturas de dados, *objects* e *arrays*. A principal vantagem da utilização de *JSON* é obter uma forma legível de utilizar colecções de chaves e os respectivos valores, suportando estruturas de dados e tipos primitivos. Abaixo, é apresentado o exemplo de um objecto *JSON* para uma melhor exposição do conceito.

```
{
    "name":"John",
    "age":30,
    "cars": {
        "car1":"Ford",
        "car2":"BMW",
        "car3":"Fiat"
    }
}
```

3.10 SOFTWARE TESTING

Como forma de detectar erros e falhas no software desenvolvido, surgiu a necessidade de testar os desenvolvimentos e implementações do mesmo. Assim, e de modo a garantir o correto comportamento do software, surge o conceito de *software testing* (51). Ou seja, *software testing*, é uma técnica que procura avaliar o software e garantir que este se comporta perante as suas especificações.

O termo inglês para um erro presente num segmento de software é *Bug*. Este erro pode, ou não, ser crítico, mas resulta num comportamento inesperado por parte do sistema: sem graves consequências ou com consequências catastróficas como o caso do *Boeing* (52), referido anteriormente.

Existem diferentes formas de testar software: desde testes que contemplam fluxos completos do início ao fim, até testes que se limitam a testar um pequeno detalhe no sistema. Uma das formas mais comuns de *software testing* é a implementação de testes unitários, isto é, testes que se debruçam sobre uma determinada parte do desenvolvimento.

Testes unitários

Unit tests ou testes unitários, referem-se aos testes individuais ou de grupos das unidades relacionadas (53). O principal objectivo dos testes unitários é garantir que o seu comportamento corresponda aos requisitos definidos, e que futuras alterações não quebrem o comportamento definido. Desta forma, é possível garantir que sucessivas iterações sobre a mesma base de código continuem a ter o comportamento esperado perante o resultado dos testes unitários implementados.

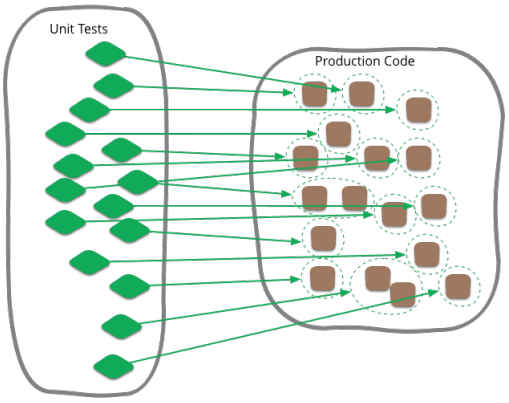


Figura 15: Testes Unitários

A implementação dos testes unitários seguiu uma metodologia derivada do *test driven development*, que surge com o acrónimo de FIRST (54):

- *Fast* (Rápidos) : Devem ter um tempo de execução rápido, dado que devem ser executados frequentemente;
- *Independent* (Independentes) : Não devem depender de outros testes, cada teste tem de ser capaz de ser executado individualmente e em qualquer ordem;
- *Repeatable* (Repetíveis) : Devem ser capazes de ser executados em qualquer ambiente seja de teste ou produção;
- *Self- Validating* (Auto-validação) : Devem devolver um valor lógico. Isto é, um teste apenas pode devolver um de dois valores, verdadeiro ou falso;
- *Timely* (A tempo) : Devem ser desenvolvidos num período de tempo aceitável, antes do código estar em produção, dado que, caso contrário, o seu propósito não fará sentido;

Os testes são uma peça tão importante para um projecto de desenvolvimento como o código de produção. Estes, permitem aumentar a flexibilidade, manutenção e reusabilidade do código, preservando os contratos existentes. Os testes têm que ser mantidos, caso contrário o código de produção irá sofrer as consequências disso (54).

3.11 SWAGGER

Consumir uma *WebApi*, e compreender os seus métodos, pode ser desafiante. Com vista a combater este desafio, foi criada a ferramenta*Swagger*, também conhecido como *OpenAPI*. Esta ferramenta permite gerar documentação e interfaces gráficas capazes de auxiliar a utilização de uma *WebAPI* (55), tal como ilustrado na figura 16.

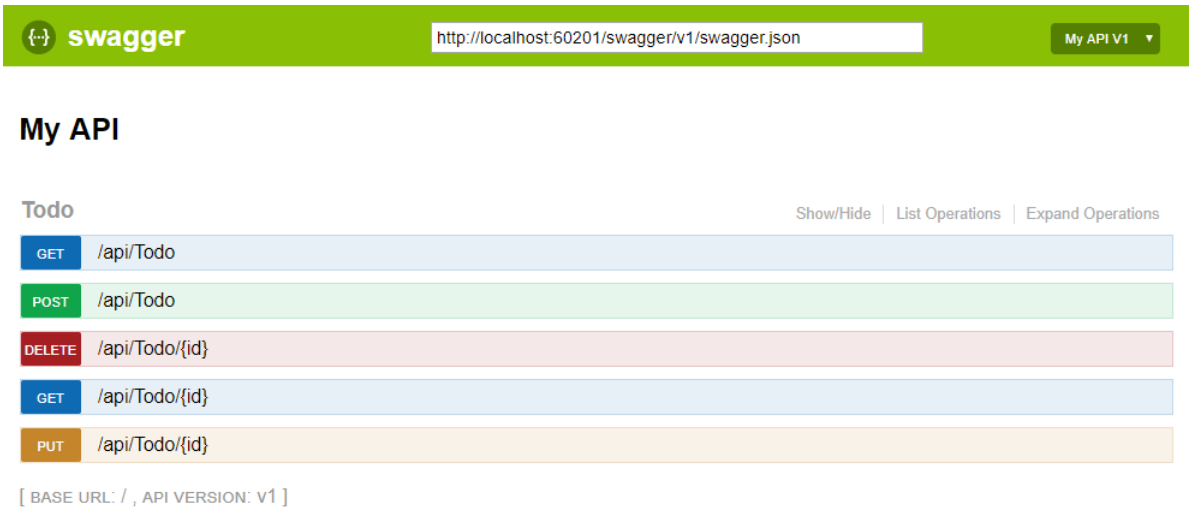


Figura 16: Interface de utilização Swagger

O *Swagger* apresenta cada *endpoint* disponível numa *WebAPI*. Isto é, cada método público disponível em cada um dos controladores, permitindo a sua utilização. Isto permite, a um utilizador não-técnico, servir-se de uma interface que poderá utilizar para manipular um sistema. Assim, após cada pedido feito através desta interface, para além das informações relativas à resposta de uma dada ação, esta interface fornece também o pedido *http* efectuado (55).

3.12 REDIS

Um dos principais objectivos de um sistema de software é a sua velocidade de resposta. Qualquer tipo de acesso à base de dados apresenta custos acrescidos, e assim, estes custos traduzem-se numa diminuição do desempenho do sistema e do seu tempo de resposta. Ou seja, o tempo que é investido no acesso à base de dados deve ser o mais diminuto possível (56). Para dar resposta a esta questão, foi implementado o *cacheamento* dos preços dos produtos, de forma a diminuir o tempo de resposta do sistema.

Com o objectivo de potencializar o desempenho do sistema, foi implementado um motor de *cache* de dados no âmbito dos preços dos produtos, através do *Redis*. O processo de colocar dados em *cache* consiste numa camada de armazenamento de alta velocidade que contem um *subset* de dados disponíveis através de uma colecção de chave-valor (57). A primeira linha de pesquisa por um determinado valor é na *cache*, caso não exista, é procurado na base de dados. A escolha dos valores que devem, ou não ser postos em cache é muito importante e deve ser feita com ponderação, sendo aconselhados valores que são procurados muitas vezes, como por exemplo, os dados apresentados na *homepage* de um *website*.

Esta implementação oferece benefícios de elevado valor como:

- Melhorar o desempenho da aplicação;
- Reduzir a carga na Base de Dados;
- Reduzir carga no *back-end* da aplicação;
- Eliminar *Hotspots* na base de dados;
- Aumentar a capacidade de Leitura.

Um objecto em *cache* é caracterizador por: pela chave que o identifica na base de dados chave-valor *Redis*, o valor que é correspondido através de uma estrutura de dados *JSON*; e finalmente o seu TTL, *time-to-live*, que define quando tempo este par chave-valor irá estar disponível;

4

DESENVOLVIMENTO DO TRABALHO

4.1 INTRODUÇÃO

Após esta análise inicial, na qual se expõe um conjunto de abordagens, ferramentas e avaliação dos potenciais benefícios das mesmas, é necessário proceder ao processo de desenvolvimento de uma solução. Esta solução, que é desenvolvida ao longo deste capítulo, terá que ser passível de gerir, em tempo real, o processo de *pricing* de um retalhista, bem como auxiliar na decisão da definição de preços. Assim sendo, este capítulo ira apresentar o desenvolvimento de uma solução responsável pelo processo de *pricing* e pelo apoio na selecção do melhor preço.

Assim, e considerando as características analisadas nos capítulos anteriores, foi definido, inicialmente, o seguinte conjunto de requisitos para a solução:

- Manipular o catálogo de produtos de um retalhista;
- Definir os preços dos diferentes produtos com várias datas de início;
- Definir as diferentes regras de *pricing*;
- Auxiliar a definição de um preço perante factores externos;

Concluído o levantamento inicial dos requisitos a implementar na solução, propomo-nos a iniciar a implementação da solução. A apresentação da implementação será feita segundo uma abordagem *bottom-up*, começando pela base de dados, expõem-se depois cada uma das camadas existentes e sequenciais até a camada de apresentação. Este capítulo tem como objectivo apresentar, detalhadamente, as diferentes fases do desenvolvimento da solução, as decisões tomadas, as estratégias implementadas e os objectivos atingidos.

4.2 ARQUITETURA DO SISTEMA

O sistema implementado enquadra-se no padrão arquitetural *Service Oriented Architecture*. Esta é uma arquitetura orientada a serviços, e consiste num conjunto de componentes chamados de serviços que comunicam entre si e disponibilizam as suas funcionalidades através

de interfaces (58). É baseada nos princípios base da computação distribuída, na escalabilidade do sistema, dado ser possível aumentar os recursos de um serviço específico apenas, e na separação de responsabilidades. Esta característica de separação de responsabilidades permite que, cada serviço, tenha apenas o seu domínio, e seja responsável por tudo o que a este domínio pertence.

Segundo Sprott e Wilkes (2004), um serviço é um componente capaz de realizar uma ou mais tarefas (59). No âmbito desta dissertação, o serviço implementado é denominado de *PricingService*, tendo como principal responsabilidade efectuar a gestão dos preços e produtos de um retalhista. Na seguinte figura, é apresentada a arquitetura geral do sistema implementado.

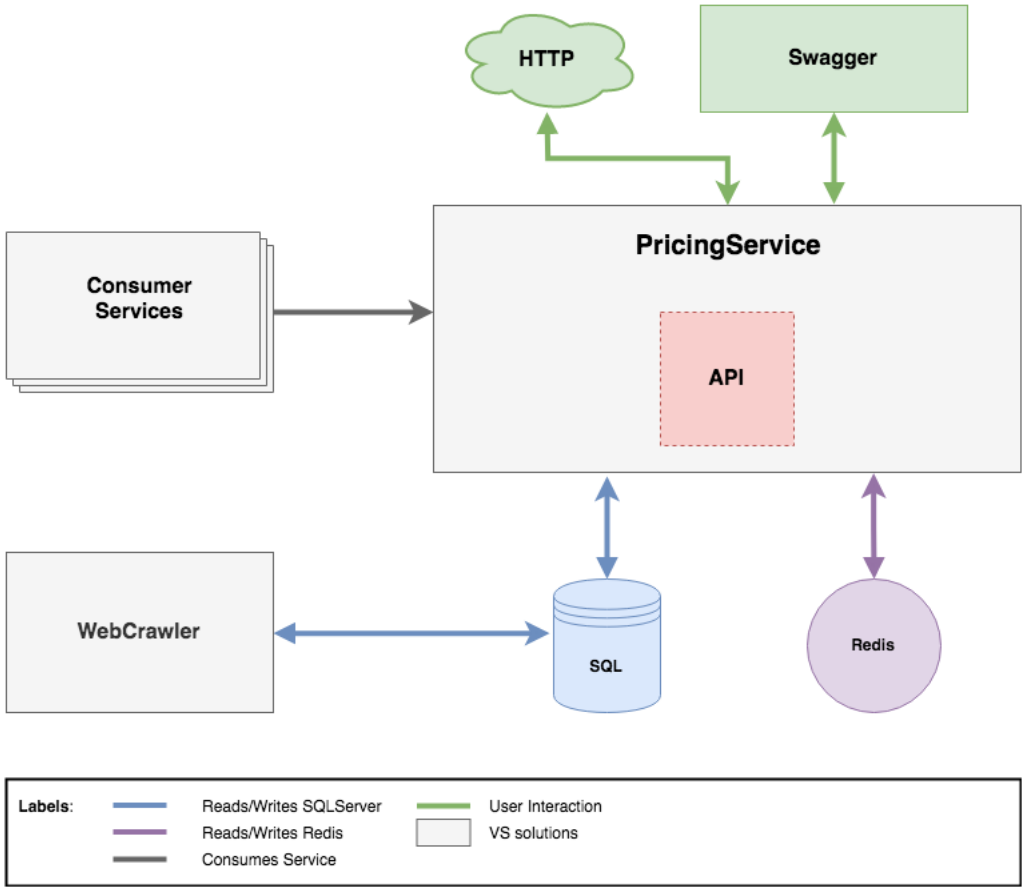


Figura 17: Arquitetura do Sistema

O sistema desenvolvido neste projeto de investigação, *PricingService*, utiliza uma base de dados relacional com o motor de base de dados *SQLServer* e um motor de *cache Redis*. De modo a potenciar a utilização do motor de regras, recorre-se a um *WebCrawler* que irá recolher dados externos, e estes serão utilizados na tomada de decisão. Esta solução comunica

com o exterior através do protocolo *HTTP* e *JSON*, tendo também as suas funcionalidades disponíveis através de uma interface gerada pela ferramenta *Swagger*.

4.3 WEB CRAWLER

De forma a obter a vantagem sobre a concorrência, ao utilizar os dados disponíveis online sobre a mesma para perceber padrões e históricos de preços, utilizar-se-á a ferramenta *Web Crawler* que irá povoar as tabelas prefixadas com *External*. Esta ferramenta tem como principal objectivo recolher informações sobre o preço de venda de produtos equivalentes nos principais atores concorrenciais. O *Web Crawler* consegue, também, tratar e inseri-los nas respectivas tabelas na base de dados. A figura 18 demonstra o fluxo de execução de um *Web Crawler* desde a fase de recolha até à fase de inserção de dados.

Figura 18: Funcionamento de um Web Crawler (60)

Um *Web Crawler*, assim sendo, é definido como uma ferramenta capaz de pesquisar dados na Internet, e recolher informação relevante, considerando os locais já visitados e a informação já levantada (61). Após a recolha de informação, a mesma é tratada e preparada de modo a ser possível, posteriormente, armazená-la na base de dados. A este processo chama-se um processo de mineração de dados.

A utilização desta ferramenta irá ter impacto no apoio da decisão da definição dos preços, pois enriquece o processo de tomada de decisão com factores externos que influenciam o processo de ponderação conferindo vantagem sobre a concorrência.

4.4 ARQUITETURA DA SOLUÇÃO

A solução *PricingService* centraliza o domínio do *pricing*, sendo responsável pela definição dos preços, a gestão da gama de produtos e pelo apoio na decisão da definição dos preços finais. Esta solução foi implementada com base na plataforma *ASP.NET Core* e segue dois padrões arquiteturais de implementação, o *Command Query Separation* e o modelo de *N-Camadas*. As principais vantagens na aplicação destes dois padrões são: a separação das escritas e leituras na base de dados, *commands* e *queries* respectivamente, uma melhor definição das responsabilidades de cada um dos projectos de classes da solução, e uma maior estabilidade e segurança na manutenção do código-fonte.

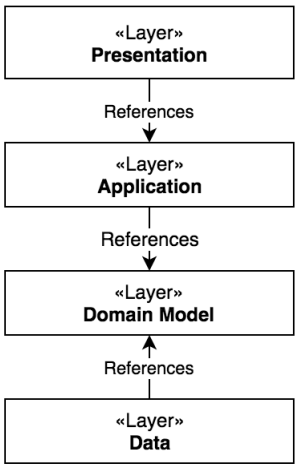


Figura 19: Vista Geral da Organização da Solução N-Camadas

A solução está dividida em quatro principais camadas, apresentadas acima na figura 19, e as suas responsabilidades são:

- **Apresentação**: A responsabilidade da camada de apresentação é a comunicação da solução com o exterior;
- **Aplicação**: Gerir a manipulação dos dados vindos da camada de apresentação;
- **Domínio**: Gerir a lógica de negócio da solução e utilização do motor de regras;
- **Dados**: Efectuar o controlo de todos os acessos à base de dados e a gestão transacional do sistema;

Para além destas quatro camadas, expostas acima, foi desenvolvida uma quinta camada: a camada de Infraestrutura. Esta camada tem o objectivo de auxiliar e responder às necessidades de qualquer uma das outras camadas. Esta pode ser referenciada por qualquer uma das outras camadas, mas não pode referenciar nenhuma das camadas existentes.

Desta forma, a estrutura da solução final desenvolvida, e que considera os pontos acima mencionados, pode ser observada na figura 20. Esta solução apresenta as cinco camadas referidas e classes de testes unitários para cama uma das camadas.

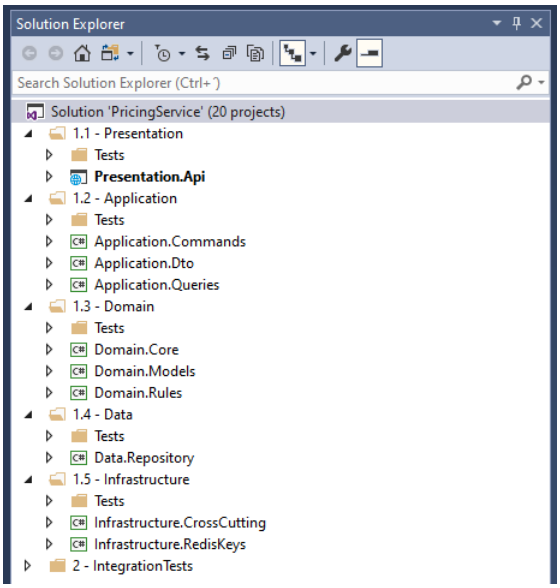


Figura 20: PricingService

4.5 MODELO DE DADOS

Com base nos requisitos definidos acima, foi escolhido um modelo de base de dados relacional de forma a expressar as relações entre as diferentes entidades presentes. O modelo de dados relacional é composto por seis tabelas: três consideradas internas à solução e três externas à mesma. As tabelas externas são identificadas pelo prefixo *External*, e surgem devido à necessidade de enriquecer a quantidade de dados disponíveis relativos à concorrência, de modo a oferecer uma visão mais abrangente do mercado e apoiar, de forma sustentada, a decisão da definição do preço.

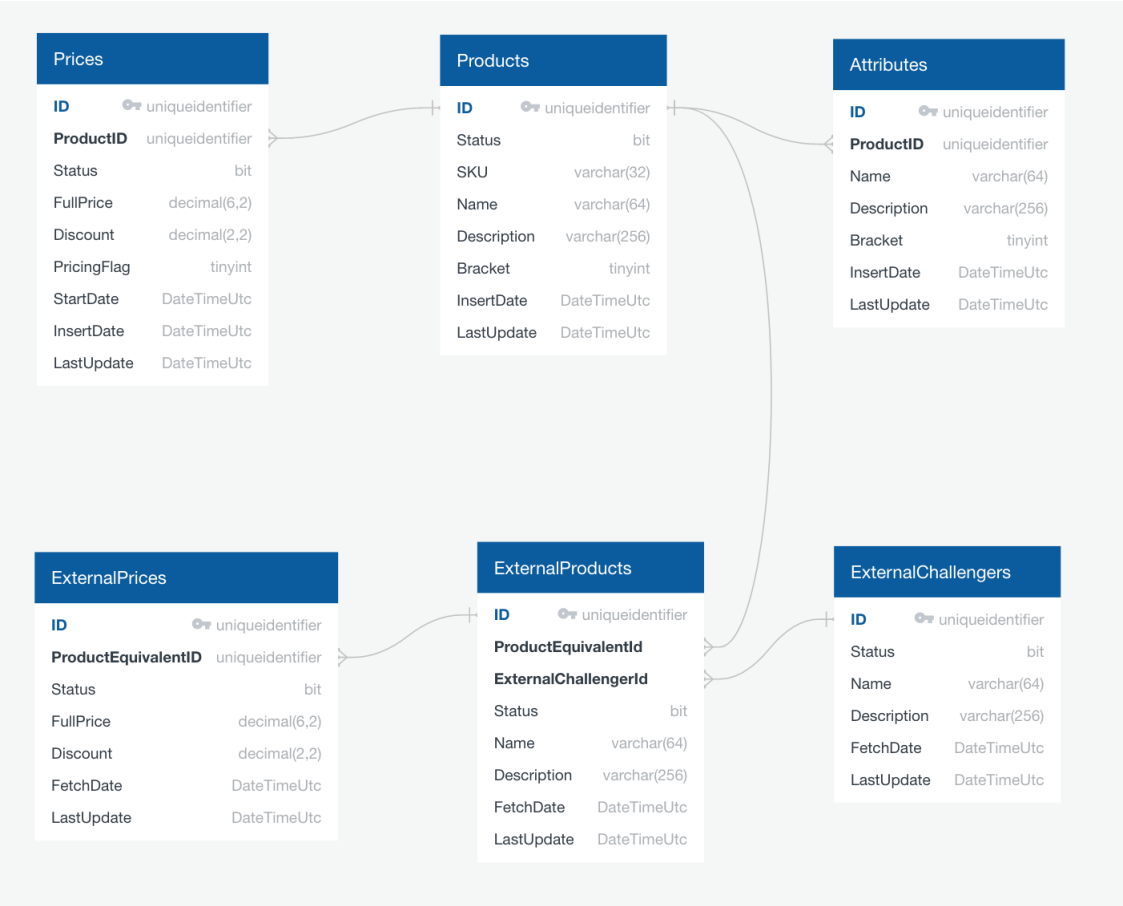


Figura 21: Modelo de Dados Relacional

O esquema apresentado na figura 21, para uma melhor visualização da estrutura, expõe as tabelas presentes no modelo de dados e as relações estabelecidas entre estas. Como referido anteriormente, o conjunto total de seis tabelas é dividido entre: tabelas que contêm dados internos, as três tabelas na linha superior; e as tabelas prefixadas com a *keyword External*, que contêm dados recolhidos em fontes externas de informação, isto é, possíveis concorrentes de negócio.

Abaixo, passaremos a analisar e detalhar cada uma das tabelas consideradas internas, *Products, Prices, Attributes*, explicando cada um dos atributos e o seu propósito no âmbito do panorama geral deste modelo de dados.

PRODUCTS

Products			
Atributo	**Descrição**	**Chave**	**Tipo**
ID	Identificador único do produto	PK	uniqueidentifier
Status	Estado do produto		bit
SKU	Identificador do produto e atributos		varchar(32)
Name	Nome do produto		varchar(64)
Description	Descrição do produto		varchar(256)
Bracket	Segmento do produto		tinyint
InsertDate	Data de inserção do produto		DateTimeUtc
LastUpdate	Data da última atualização do produto		DateTimeUtc

Figura 22: Caracterização da tabela *Products*

A tabela *Products*, apresentada na figura 22, define a representação da entidade *Produto*. Como se pode observar na figura acima, um produto é constituído pelo seu *ID*, o identificador único do produto, e o seu estado é representado pelo atributo: *Status*. Este atributo, sendo um bit, pode tomar apenas dois valores, 0 e 1, que significam que o produto está inactivo ou activo, respectivamente.

Os atributos, *SKU (Stock Keeping Unit)* e *Name*, são também identificadores do produto. O *SKU* define um identificador único do produto e uma dada combinação dos seus atributos. O *Name* representa o nome do produto que será apresentado ao utilizador final na venda do produto. A descrição de cada produto está presente no atributo *Description* e o atributo *Bracket* identifica o segmento respectivo ao produto. O segmento de um produto é considerado na execução do motor de regras de modo a suportar o cálculo final do preço de um produto.

Finalmente, os atributos *InsertDate* e *LastUpdate* correspondem às datas nas quais o produto foi inserido e actualizado pela última vez, respectivamente.

Esta tabela tem como principal objectivo agregar toda a informação global da entidade *Produto*. Pode ser considerada uma das peças centrais de todo o modelo de dados, sendo referenciada através de chaves estrangeiras nas tabelas *Prices*, *Attributes* e *ExternalProducts*.

PRICES

Prices			
Atributo	**Descrição**	**Chave**	**Tipo**
ID	Identificador único do preço	PK	uniqueidentifier
ProductID	Identificador único do produto	FK	uniqueidentifier
Status	Estado do preço		bit
FullPrice	Valor total do preço de venda		decimal(6,2)
Discount	Valor do desconto do produto		decimal(2,2)
PricingFlag	Regra default de cálculo de preço		tinyint
StartDate	Data de ativação do produto		DateTimeUtc
InsertDate	Data de inserção do produto		DateTimeUtc
LastUpdate	Data da última atualização do produto		DateTimeUtc

Figura 23: Caracterização da tabela Prices

A tabela *Prices*, representada na figura 23, agrega toda a informação contida na definição do preço de venda interno de um determinado produto. Observando a figura 23, verificamos que a chave-primária é o atributo *ID*, que é o identificador único do preço de um dado produto referenciado através da chave estrangeira *ProductID*. O seu estado é representado pelo atributo *Status* que, tal como na tabela analisada anteriormente, poderá tomar os mesmos valores.

O valor total de venda de um produto é dado pelo *FullPrice*, que é armazenado no *datatype decimal(6,2)*, isto significa que poderá conter seis algarismos significativos e duas casas decimais. A percentagem de desconto definida para um dado produto é representada no *datatype decimal(2,2)*. Ou seja, poderá ter dois algarismos significativos e duas casas decimais. A regra utilizada no cálculo do preço de venda final de um produto está presente na linha *PricingFlag*.

As últimas três linhas da tabela *Prices* são: a linha *StartDate*, que define a data e hora a partir da qual o valor definido passará a ter efeito e, desta forma, é possível definir antecipadamente os preços do catálogo de produtos presente; e as tabelas *InsertDate* e *LastUpdate* que fornecem informação sobre a data de inserção do preço e da última atualização do mesmo, respectivamente.

ATTRIBUTES

Attributes			
Atributo	**Descrição**	**Chave**	**Tipo**
ID	Identificador único do atributo	PK	uniqueidentifier
ProductID	Identificador único do produto	FK	bit
Name	Nome do atributo		varchar(64)
Description	Descrição do atributo		varchar(256)
Bracket	Segmento do atributo		tinyint
InsertDate	Data de inserção do atributo		DateTimeUtc
LastUpdate	Data da última atualização do atributo		DateTimeUtc

Figura 24: Caracterização da tabela *Attributes*

A tabela *Attributes*, ilustrada na figura 24, representa o conjunto de características que definem um dado produto e será, também, um ponto de extensibilidade futuro. Esta característica de extensibilidade possibilita no futuro, caso exista esta necessidade, aumentar a complexidade das regras de cálculo de preço, introduzindo mais elementos que podem ser considerados na definição de regras. A linha *ID* identifica, unicamente, o atributo do produto (sendo a chave-primária desta tabela), o produto, por sua vez, é referenciado pela sua chave-estrangeira presente na linha *ProductID*.

Cada atributo tem um nome, descrição e categoria presentes nas linhas *Name*, *Description* e *Bracket*, respetivamente. Sempre que um atributo é inserido, ou modificado, a data da operação é registada nas linhas *InsertedDate* e *LastUpdate*.

EXTERNAL

De modo a potencializar o motor de regras utilizado, são considerados dados externos relacionados com os dados existentes nas tabelas de cariz interno. Assim sendo, é possível ter uma decisão fundamentada tanto no contexto interno como no mercado concorrencial, no momento em que o cálculo do preço de um produto é efectuado. Existem 3 tabelas externas, prefixadas com *External*, que são sustentadas através de um processo de mineração de dados levado a cabo por um *Web Crawler*. Este processo, tal como descrito acima, recolhe, trata e insere os dados nas três tabelas.

A estrutura das tabelas *ExternalProducts* e *ExternalPrices* é muito semelhante às tabelas homólogas internas, *Products* e *Prices*. Esta estrutura existe para que se represente e aproxime, da melhor forma, a informação recolhida em fontes externas ao contexto interno e para garantir que exista, apenas, uma discrepância mínima entre as duas fontes de dados.

Estas tabelas irão ter como principal objectivo obter vantagem sobre os outros retalhistas no mercado ao considerar, para definição do preço final, a análise das práticas que estes exercem. Estes dados aparecerão representados em tabelas de *External Prices*, como se pode verificar na tabela a seguir.

EXTERNALPRICES

ExternalPrices			
Atributo	**Descrição**	**Chave**	**Tipo**
ID	Identificador único do preço externo	PK	uniqueidentifier
ProductEquivalentID	Identificador do produto equivalente	FK	uniqueidentifier
Status	Estado do preço		bit
FullPrice	Valor do preço de venda		decimal(6,2)
Discount	Valor do desconto		decimal(2,2)
FetchDate	Data de recolha do preço		DateTimeUtc
LastUpdate	Data da última atualização		DateTimeUtc

Figura 25: Caracterização da tabela ExternalsPrices

Tal como referido acima, a tabela *ExternalPrices* é estruturalmente semelhante à tabela homóloga, a tabela de *Prices*. Isto porque, esta tabela de *External Prices*, obedece a uma estrutura que tem como objectivo que, as informações presentes em ambas, assumam uma forma correspondente de modo a não criar conflito no processamento dos seus dados. Assim sendo, um preço externo é identificado pelo seu *ID*, pela referência ao produto equivalente correspondente na tabela *ExternalProducts* e pelo seu estado presente na coluna *Status*.

O preço de um produto praticado por um dado competidor e o seu desconto é representado, respectivamente, nas colunas *FullPrice* e *Discount*. Caso estes valores recolhidos deixem de estar disponíveis, o estado do preço é actualizado passando para inactivo.

Finalmente, nas colunas *FetchDate* e *LastUpdate*, são guardadas as datas da recolha do preço em questão e da última actualização do mesmo, nesta ordem.

EXTERNALCHALLENGERS

ExternalChallengers			
Atributo	**Descrição**	**Chave**	**Tipo**
ID	Identificador único do competidor	PK	uniqueidentifier
Status	Estado do competidor		bit
Name	Nome do competidor		varchar(64)
Description	Descrição do competidor		varchar(256)
FetchDate	Data de recolha do competidor		DateTimeUtc
LastUpdate	Data da última atualização		DateTimeUtc

Figura 26: Caracterização da tabela ExternalChallengers

A tabela *ExternalChallengers* agrega a informação relativa aos concorrentes de mercado, sendo novamente identificada pela linha *ID*, que corresponde à chave-primária desta tabela. Cada competidor tem um nome, presente na linha *Name*, pela breve descrição na linha *Description*, e pelo seu estado representado na linha *Status*. De acordo com as tabelas expostas anteriormente, também aqui, a informação referente à data de inserção e atualização de um competidor pode ser encontrada, respetivamente, nas linhas *InsertedDate* e *LastUpdate*.

EXTERNALPRODUCTS

ExternalProducts			
Atributo	**Descrição**	**Chave**	**Tipo**
ID	Identificador único do produto externo	PK	uniqueidentifier
ProductEquivalentID	Identificador do produto equivalente	FK	uniqueidentifier
ExternalChallengerID	Identificador do competidor	FK	uniqueidentifier
Status	Estado do produto externo		bit
Name	Nome do produto externo		varchar(64)
Description	Descrição do produto externo		varchar(256)
FetchDate	Data de recolha do competidor		DateTimeUtc
LastUpdate	Data da última atualização		DateTimeUtc

Figura 27: Caracterização da tabela ExternalsProducts

Tal como a semelhança estrutural que se observa nas tabelas homólogas *Products* e *External Prices*, também a tabela *ExternalProducts* se apresenta como homóloga à tabela *Products*. Assim sendo, e respeitando a estrutura definida pela tabela *Products*, um produto recolhido num competidor é identificado pelo atributo *ID*, e referencia as chaves primárias das ta-

belas *Products* e *ExternalChallengers*, nas linhas *ProductEquivalentID* e *ExternalChallengerID*, respectivamente.

O nome de cada produto no concorrente é inserido na linha *Name*, juntamente com uma breve descrição presente na linha *Description*. Caso um produto deixe de ser vendido por um concorrente, o seu estado é actualizado na linha *Status* para inactivo.

Finalmente, nas linhas *FetchDate* e *LastUpdate*, são guardadas as datas da recolha do preço em questão e da última actualização do mesmo, como se viu anteriormente.

4.6 CAMADA DE DADOS

A camada de dados, ou *DAL - data access layer* que é o acrónimo usado em ambientes *Microsoft*, corresponde à camada responsável por providenciar o acesso à base de dados, e manter a persistência da mesma perante as transacções ocorridas. Nesta camada, a forma como os dados estão organizados na base de dados é abstraída na forma de entidades. Esta forma de representação abstracta permite uma maior modularidade e independência em relação ao motor de base de dados, facilitando a mudança, considerando que apenas as entidades irão necessitar de alterações. Tal como as entidades são abstraídas da estrutura do motor de base de dados utilizado, também a lógica de negocio é abstraída. Assim sendo, a única responsabilidade desta camada é o tratamento, a leitura e a escrita de dados.

Esta camada está contida numa única biblioteca de classes *Data.Repository* com a seguinte estrutura:

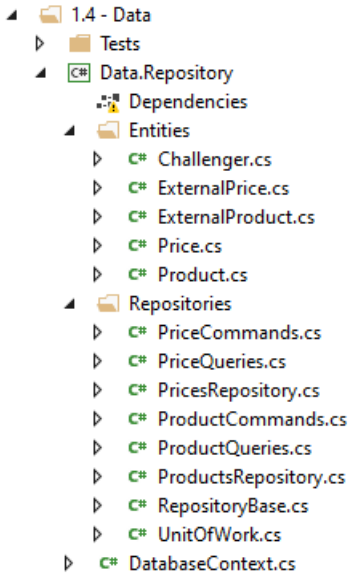


Figura 28: Camada de Dados

De forma a mitigar potenciais problemas de persistência de dados foram utilizados os padrões *Repository e Unit of Work*.

Data.Repository

A figura 28, acima ilustrada, apresenta a camada de dados da solução. Esta camada é composta por uma biblioteca de classes de nome *Data.Repository*, e pela classe de testes que define os testes unitários a fazer aos métodos implementados sobre a pasta *Tests*.

A biblioteca *Data.Repository*, como se pode ver na figura 28 representada acima, é dividida em 3 partes que estão organizadas nas devidas pastas. A pasta *Entities* contém as classes que mapeiam as tabelas da base de dados para as entidades do sistema. Este mapeamento possibilita que os dados presentes na base de dados sejam convertidos para objectos posteriormente utilizados. A pasta *Repositories* contém as classes de repositório responsáveis pela gestão das transacções efectuadas, estando separadas entre comandos e pesquisas, que são agregados na classe-mãe de repositório. Por último, a classe *DatabaseContext*, por ser a peça central desta camada, organiza-se separadamente. Esta classe é responsável por efectuar a ligação à base de dados que, no final, permite a escrita e leitura dos dados nas respectivas tabelas.

Database Context

A classe de contexto, *DatabaseContext*, é a classe mais importante quando é utilizada a *Entity Framework*. Esta classe representa a ligação estabelecida com a base de dados, e permite efectuar as operações de leitura, escrita, actualização e eliminação de dados (*create, read, update, delete*) (62). Ao conjunto destas quatro operações atribui-se o acrónimo CRUD.

A classe *DatabaseContext* estende a classe *DbContext* fornecida pela *Entity Framework*. Desta forma, em combinação com o *Repository Pattern* e *Unit of Work*, é possível consultar a base de dados e agrupar as alterações que serão efectuadas de modo a serem feitas numa única transação. As restantes responsabilidades desta classe são: configurar o mapeamento das classes de entidades para as tabelas das bases de dados, e vice-versa; e, também, manter o estado do contexto no sistema, agrupando a informação sobre as transacções a efectuar sobre a base de dados.

Entities

Na *Entity Framework*, um dos principais objectivos é possibilitar a interação entre o sistema e a base de dados. Desta forma, é necessário mapear as tabelas para objectos permitindo a sua manipulação independente da tecnologia utilizada no armazenamento de dados. O

nome dado a um objecto resultante do mapeamento de uma tabela de base de dados para um objecto é *Entidade*.

Com a finalidade de permitir a manipulação dos dados, presentes nas tabelas, foi criado um conjunto de classes que mapeiam cada uma das linhas de uma tabela de base de dados para o respectivo objecto manipulável.

As tabelas de cariz interno, foram mapeadas, nas classes *Price, Product e Challenger*. Cada uma das colunas presentes nas suas tabelas irá corresponder, na classe de destino, a um atributo do objecto. Por sua vez, cada linha irá corresponder a um objecto diferente. Para tornar mais clara esta exposição, daremos como exemplo a estruturação da classe *Price*. A estrutura desta classe é definida por: o identificador do tipo*Guid*; o estado, que é definido através de um valor *booleano*; o preço total e o desconto, através de um valor decimal; a regra de preço por um inteiro; as datas (de activação, de inserção e actualização); e, por último, o produto correspondente à sua chave-estrangeira, *Product*. Se analisarmos os atributos desta classe, verificamos que a sua estrutura é muito semelhante à tabela correspondente na base de dados. Assim, e alterando-se apenas os atributos e os seus objectos correspondentes, para as classes de *Product e Challenger*, o mapeamento foi feito de igual forma.

De modo a mapear as tabelas *ExternalProducts* e *ExternalPrices*, foi utilizada uma funcionalidade primária do paradigma de Programação Orientada a Objectos: a Herança (63). Sendo que as tabelas têm diversas propriedades em comum, as classes prefixadas com *External* irão herdar as propriedades da respectiva classe-base interna, apenas adicionando as propriedades que não são comuns a ambas as tabelas.

Por exemplo, a classe *ExternalPrices* irá compartilhar, ou herdar, todos os atributos presentes na classe *Prices*, por se encontrarem em tabelas com estruturas semelhantes relativamente às suas colunas. A estes atributos, acrescentam-se, assim, a data em que os valores foram recolhidos (*FetchDate*, e o objecto *ExternalProduct* que corresponde ao preço em questão.

Cada uma das entidades definidas sobre o *namespace PricingService.Data.Repository.Entities* serão referenciadas na classe de contexto, de modo a, através da *Entity Framework*, ser possível transaccionar dados sobre a base de dados.

Repositories

S A pasta *Repositories* define o *namespace PricingService.Data.Repository.Repositories*. Sobre este *namespace* encontram-se todas as classes responsáveis pela gestão transaccional das operações efectuadas. Neste *namespace*, a denominação das classes é de extrema relevância pois o nome que é atribuído a cada classe deve identificar, de forma clara, a sua responsabilidade inerente. Esta é considerada uma boa prática de programação, pois facilita a leitura e organização da estrutura do código. Assim, as classes são nomeadas da seguinte forma:

< Entidade >< Responsabilidade >

Através deste processo de denominação das classes, é possível que, apenas através do nome de uma classe, se identifique qual a entidade sobre a qual é responsável e que tipo de responsabilidades contém.

Tal como abordado anteriormente, esta solução segue uma implementação do padrão *Command Query Separation*. Assim sendo, as operações de escrita e leitura são segregadas em *Commands* e *Queries*, respectivamente. Sendo estes, também, os sufixos que identificam esta separação. A orquestração da escrita e da leitura de cada entidade são da responsabilidade das classes sufixadas com *Repository*.

Commands

Para um melhor entendimento destas operações, escrita e leitura, passaremos a expor as responsabilidades de cada uma destas classes. As classes sufixadas por *Commands*, são responsáveis pelas escritas na base de dados. As responsabilidades desta classe podem ser definidas através de três operações: a inserção, a actualização e a remoção de dados de uma determinada entidade na sua respectiva tabela. Assim, as classes *Commands*, extendem a classe-base *RepositoryBase* que contém o conjunto de operações comuns entre todas as classes de repositório. Esta é uma boa prática de programação, centralizando as responsabilidades e evitando a repetição do código.

Queries

As classes sufixadas por *Queries*, são responsáveis por efectuar as leituras de uma dada entidade na base de dados, quer nas tabelas externas *External*, como internas. Isto é, a classe correspondente às leituras de uma entidade, efectua a leitura em todas as tabelas referentes a esta entidade. Por exemplo, a classe *PriceQueries*, realiza leituras sobre as tabelas *Prices* e *ExternalPrices*. Esta leitura pode resultar na materialização de um ou mais objectos ou, apenas, na confirmação de alguma entrada que verifica uma dada condição. Observe-se, a título de exemplo, a pesquisa de todos os preços, com uma determinada data de início, que irá resultar numa lista de preços ou na verificação da existência de algum produto externo com o preço de venda inferior a um dado valor.

Repositories

Após a definição e a clarificação das responsabilidade das classes *Commands* e *Queries*, explicaremos aqui qual o papel das classes de *Repositories* dentro do sistema e para com as classes referidas. As classes sufixadas por *Repositories* são responsáveis por fazer a orquestração das classes de *Commands* e de *Queries*, fazendo, também, o tratamento de erros que ocorram durante as operações de escrita e leitura.

Esta classe é a ponte de ligação com a camada superior. Ou seja, disponibiliza as suas funcionalidades sem revelar a sua implementação, através do seu ponto de extensão que é a interface que implementa. Por exemplo, a classe *PricesRepository* contém ambas as classes de escrita e leitura, *PricesCommands* e *PricesQueries*, e, com estas, implementa os contratos definidos na interface correspondente da camada de domínio disponibilizada.

Unit of Work

A classe sufixada por *Unit of Work* é a classe responsável por gerir todas as operações efectuadas. Ou seja, a responsabilidade desta classe é controlar e dirigir todas as transacções a efectuar dentro da base de dados do sistema. Tal como o nome indica, esta classe existe como uma unidade de trabalho que centraliza a gestão de acções, de forma a agrupar uma ou mais operações, transformando-as numa única transação a realizar sobre a base de dados.

4.7 CAMADA DE DOMÍNIO

A camada de domínio tem como principal responsabilidade definir e representar os conceitos e regras de negócio de um dado sistema. Nesta camada é reflectido o estado, o controlo, e o uso das regras de negócio. Esta é considerada, portanto, a peça central do sistema.

Esta camada contém as entidades de domínio que, por sua vez, contêm os dados que se pretendem representar juntamente com a lógica de negócio associada aos mesmos. Os contratos de negócio, que devem ser seguidos pelas camadas adjacentes, fazem, também, parte da camada de domínio. Estes são representados na forma de interfaces de modo a estabelecer o contrato sobre as suas implementações. Os detalhes de persistência de dados são ignorados nesta camada, devendo esta preocupação residir na camada de dados. Desta forma, a camada de domínio não tem qualquer dependência, sendo possível atingir uma maior versatilidade perante os detalhes técnicos da implementação de toda a lógica de negócio.

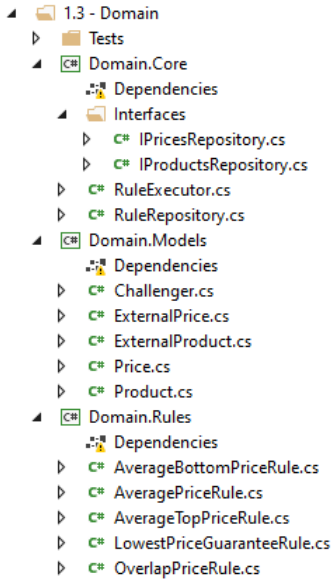


Figura 29: Camada de Domínio

A figura 29 mostra a estrutura e a organização da camada de domínio. Através desta imagem, consegue mostrar-se a divisão da camada em três bibliotecas de classes: *Domain.Core*, que contém as peças fundamentais para a execução do motor de regras e os contratos de negócio definidos; *Domain.Models*, que contém os modelos que definem as entidades de domínio; e, finalmente, *Domain.Rules*, que define as regras que irão ser utilizadas pelo motor de regras.

Domain.Rules

A biblioteca de classes *Domain.Rules*, contém as classes que definem as regras utilizadas pelo motor de regras. De forma a facilitar a exposição das regras definidas, representa-se abaixo, em forma de equação, a lógica inerente a cada uma delas. A execução destas regras é ditada pelo atributo *PricingFlag*, presente no modelo de domínio da entidade *Price*, que define qual a regra a ser executada. As *PricingFlags* disponíveis para utilização deste sistema são as seguintes:

- 0: nenhuma regra é executada;
- 1: preço mínimo garantido;
- 2: preço superior à média da concorrência;
- 3: preço inferior à média da concorrência;
- 4: preço igual à média da concorrência.

PricingFlag 1: definição do preço mínimo garantido. Significa que, caso o menor valor recolhido da concorrência for superior ao valor definido para um dado produto, o valor de venda será 5% inferior ao valor menor registado na concorrência, tal como ilustrado na equação 1.

$$min\left(external_{prices}\right) >= price \rightarrow price = min\left(external_{prices}\right) \cdot 95\% \quad (1)$$

PricingFlag 2: preço superior à média da concorrência. Significa que, caso o valor médio recolhido da concorrência for superior ao valor definido para um dado produto, o valor de venda será superior segundo o coeficiente definido.

$$\Pi\left(external_{prices}\right) > price \rightarrow price = \Pi\left(external_{prices}\right) \cdot K \quad (2)$$

PricingFlag 3: preço inferior à média da concorrência. Significa que, caso o valor médio recolhido da concorrência for inferior ao valor definido para um dado produto, o valor de venda será inferior segundo o coeficiente definido.

$$\Pi\left(external_{prices}\right) < price \rightarrow price = \Pi\left(external_{prices}\right) \cdot K \quad (3)$$

PricingFlag 4: preço igual à média da concorrência. Quando esta *PricingFlag* estiver activa, o valor médio recolhido da concorrência será atribuído ao preço de venda de um produto.

$$\Pi\left(external_{prices}\right) \rightarrow price = \Pi\left(external_{prices}\right) \quad (4)$$

A correspondência entre cada uma destas equações e as classes que as definem é:

- *PricingFlag* 1: *LowestPriceGuaranteeRule*
- *PricingFlag* 2: *AverageTopPriceRule*
- *PricingFlag* 3: *AverageBottomPriceRule*
- *PricingFlag* 4: *AveragePriceRule*

Domain.Models

Uma entidade de domínio é definida sobre a forma de um *POCO*, *Plain Old Crl Object*, que é um objecto criado na *Common Language Runtime* da *.NET Framework*(64). Com esta forma, é criado um mecanismo eficaz de armazenamento de dados que permite:

- a simplificação da serialização e da transmissão de dados;
- a minimização da complexidade e das dependências;

- a compatibilidade com os princípios de *dependency injection* e *repository pattern* utilizados;
- e, por último, aumentar a capacidade de isolamento de testes.

Para além destas responsabilidades, uma entidade de domínio deve, também, expressar a sua lógica única que é representativa das regras de negócio. As regras de negócio traduzem-se nas acções disponíveis numa dada entidade. Em domínios mais pequenos, como é o caso do sistema aqui desenvolvido, as entidades de domínio podem não conter toda a lógica única, porque as regras de negócio são já representadas nas regras de domínio, *Domain.Rules*, com a forma de *flags*.

Desta forma, os modelos definidos acabam por ser muito semelhantes às entidades de dados que lhes deram origem, no que às propriedades dos objectos diz respeito, adicionando a estas propriedades a lógica de negócio associada a cada entidade de domínio, caso esta exista.

Domain.Core

A biblioteca de classes denominada como *Domain.Core*, é considerada o núcleo da camada de domínio, tal com o nome indica. A relevância desta peça traduz-se no facto de ser aqui que estão estabelecidas as interfaces que servem de contrato às camadas adjacentes. Estas interfaces definem os métodos disponíveis que terão de ser, obrigatoriamente, implementados pelas classes adjacentes que as pretendam utilizar. Através do mecanismo de *dependency injection*, estes contratos são cumpridos na íntegra pois, para a sua utilização, uma classe necessita de uma implementação da interface e, caso esta não exista, o motor de resolução de dependências irá retornar um erro.

A camada de domínio é a peça central de todo o sistema aqui desenvolvido, pois é no núcleo desta camada que o processo de apoio à decisão acontece. Ou seja, o processo de apoio à decisão encontra a sua dinâmica central na execução do motor de regras utilizado, o *NRules*. O motor de regras é executado nas acções de recolha de preços quando a *PricingFlag* presente é diferente do valor zero. A orquestração da execução do motor de regras é realizada em duas classes presentes nesta biblioteca: *RuleRepository* e *RuleExecutor*.

A primeira, *RuleRepository*, contém a colecção de todas as regras existentes. Estas regras definem o alvo da inferência feita pelo motor de regras durante a sua execução. No arranque do ciclo de execução do motor de regras esta colecção é inserida em memória no motor de regras através da classe *RuleExecutor*. A classe *RuleExecutor* é responsável por gerir a execução do motor de regras, estando a sua execução dividida nas seguintes fases: o carregamento e compilação das regras para memória, a injecção de dados da entidade de domínio na sessão do motor de regras, e, finalmente, a iniciação da execução do motor de

regras (que irá efectuar o ciclo de *match/resolve/act*). Por exemplo, quando um preço interno e o conjunto homólogo de preços externos forem inseridos em memória, caso a *pricingflag* detectada seja 1, o motor irá fazer *match* com a regra *lowestpriceguarantee* e resolverá a condição definida nesta regra, apenas se esta se verificar. Ou seja, se o preço mais baixo dos *externalprices* for inferior ao preço interno, o motor de regras irá actuar alterando o preço final interno.

4.8 CAMADA DE APLICAÇÃO

A camada de aplicação serve como intermediário entre a camada de apresentação, *Presentation*, e a camada responsável pela lógica de negócio, *Domain*. Esta camada orquestra as acções necessárias a executar, delegando-as para as classes correspondentes na camada de domínio.

As responsabilidades desta camada são significativas, quer do ponto de vista de lógica de negócio, quer do ponto de vista da interação com outros sistemas. Isto porque, a camada de aplicação não contém nem conhece regras de negócio: apenas coordena os representantes da lógica de negócio de forma a isolar a lógica de negócio das camadas adjacentes. Assim, pretende-se que, esta camada, seja de baixa complexidade.

Nesta camada, a passagem de informação é feita através de *DTOs*, *Data Transfer Objects*. Um DTO é estrutura de dados que permite modelar os dados que irão ser enviados na comunicação de rede. O uso deste tipo de estruturas de dados apresenta, como principais vantagens, reduzir o *payload* na transferência de dados, garantir que apenas os dados necessários são enviados, e reduzir o *coupling* entre a camada de apresentação e a camada de domínio.

É, também, responsabilidade desta camada a implementação da separação inicial entre escritas, *commands* e leituras, *queries*, seguindo o padrão já mencionado *Command Query Separation*(65). Esta separação deu origem a duas bibliotecas de classes, *Application.Commands* e *Application.Queries*, que, sendo disponibilizadas para a camada de apresentação, têm como única responsabilidade fazer a ponte de ligação entre a camada de apresentação e a de domínio.

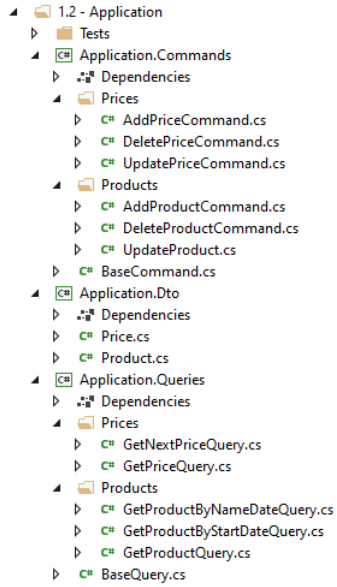


Figura 30: Camada de Aplicação

A figura 30, acima representada, expõe a forma como a camada de aplicação está estruturada e organizada. Tendo, esta camada, a única responsabilidade de orquestração da separação de leituras e escritas dos pedidos recebidos, efectuando a ligação entre a camada de apresentação e a de domínio, o seu nível de complexidade é baixo. Desta forma, as classes implementadas nesta camada são, também, de baixa complexidade, respeitando o objectivo único de ligação mencionado acima. Observando esta figura, consegue perceber-se que esta camada é composta por três peças distintas, ou seja, três bibliotecas de classes. Considerando as diferenças entre as três bibliotecas, estas distinguem-se da seguinte forma:

- *Application.DTO*: contém os DTOs, *data transfer objects*, utilizados na camada de apresentação com a finalidade de interação com o utilizador;
- *Application.Queries*: contém a responsabilidade da delegação dos pedidos de leituras para a camada inferior;
- *Application.Commands*: é responsável pela gestão dos pedidos de leituras recebidos.

Application.DTO

A centralização de todos os objectos recebidos e retornados, durante a interação com o utilizador, está presente na biblioteca de classes *Application.DTO*. Esta biblioteca tem como única responsabilidade definir e agregar todos os objectos que são utilizados na interação com o utilizador, isto é, a camada de apresentação.

Os objectos aqui definidos têm como nome DTO, *data transfer objects*. Um *DTO* tem como função encapsular os dados que são enviados de uma aplicação ou sistema para outra. A principal diferença entre estes objectos e as entidades de domínio, definidas na camada de domínio, reside no facto de não conterem qualquer lógica, sendo apenas compostos pelas propriedades do objecto em questão. Isto permite tornar a comunicação de dados mais leve, rápida e menos susceptível a perdas de informação, dado que apenas a informação necessária é transportada.

Application.Queries e Application.Commands

Sobre o *namespace Application.Queries* encontram-se as classes responsáveis pela delegação dos pedidos de leitura para a camada inferior, a camada de domínio. Ou seja, estas são as classes disponibilizadas para a camada de apresentação com o objectivo de separar as leituras das escritas. Estas classes encontram-se organizadas segundo a entidade alvo a que se referem: *Prices* e *Products*. Por sua vez, sobre o *namespace Application.Commands*, encontram-se as classes responsáveis pela delegação dos pedidos de escrita, adição, alteração e remoção de dados. Da mesma forma, estas classes encontram-se organizadas segundo a entidade onde actuam: *Prices* e *Products*. Tal como se verifica na figura acima, a estrutura organizacional é semelhante entre as classes de leitura e de escrita, tendo como objectivo final a definição e separação claras das responsabilidades de cada uma destas bibliotecas de classes.

4.9 Camada de Apresentação

Um dos objectivos fundamentais de um sistema de apoio à decisão é a sua usabilidade perante utilizadores técnicos ou não-técnicos, através de uma interface gráfica clara e acessível. Esta interface deverá permitir a fácil utilização do sistema e possibilitar a manipulação e gestão de dados, sem necessitar de conhecimentos técnicos ao nível de desenvolvimento de software.

A versatilidade na manuseabilidade do sistema permite que este possa ser utilizado de duas formas distintas:

- pode ser utilizado por um utilizador com conhecimentos técnicos, um programador, que irá utilizar o sistema e fazer a integração no seu ambiente, através de chamadas HTTP;
- ou, poderá ser utilizado como referido acima, por um utilizador não-técnico, que irá servir-se da interface gráfica disponibilizada para interagir com o sistema.

Desta forma, dois tipos distintos de utilizadores podem interagir e utilizar o sistema, podendo ambos, facilmente, consultar e modificar o catálogo de produtos e preços.

Assim, e considerando que o sistema pode ser consumido por estes dois tipos de utilizador, a camada de apresentação do sistema é composta pelo projeto *Presentation.API*. Aqui, são disponibilizadas, para o exterior, as funcionalidades do sistema através da instanciação de um servidor feita através do *Krestel*, fornecido pela *framework .NETCORE* (36). Este servidor pode, posteriormente, ser utilizado através de chamadas HTTP, para utilizadores considerados técnicos, ou pela interface gráfica *Swagger*, aconselhada a utilizadores não-técnicos. Assim, é possível abranger diferentes tipos de utilizadores que podem utilizar o sistema da forma que melhor se adequar às suas capacidades técnicas.

Presentation.API

O projecto *Presentation.Api* contém a camada de apresentação da aplicação que, sendo a camada mais superior no sistema, será o ponto de contacto com o exterior e o utilizador do mesmo. Este ponto de contacto é definido nos controladores, classes sufixadas por *Controller*. Cada controlador tem a responsabilidade de controlar, tal como o nome indica, a interação do utilizador com a lógica de execução do pedido e da resposta. Para além disto, também define qual o verbo *HTTP* de cada pedido, e quais as respostas possíveis para um dado pedido consoante o decorrer da execução do mesmo.

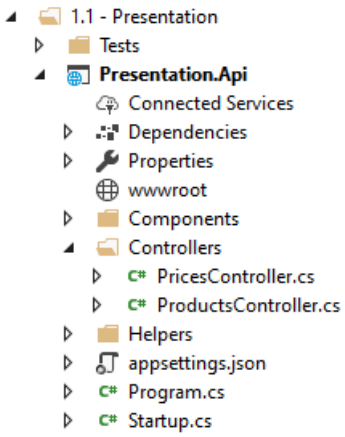


Figura 31: Camada de Apresentação

Tal como se observa na figura 31, os dois controladores, *PricesController* e *ProductsController*, aparecem como peças centrais deste sistema. Estas duas classes definem quais as acções disponibilizadas para os utilizadores, quais os seus requisitos de utilização e que respostas podem retornar. O nome da classe e do método são de extrema importância para a organização deste projecto, dado que fornecem a indicação de qual o método *http* usado e também qual o domínio alvo da ação. Por exemplo, todas as acções relacionadas com preços estão

na classe *PricesController*, e o método *GetPrice* fornece a recolha de um determinado preço identificado pelo seu identificador.

A gestão das funcionalidades que estão, ou não, disponibilizadas para o exterior é feita através da definição das variáveis de acesso nos métodos dos controladores. Isto porque, apenas os métodos que possuem a variável de controlo de acesso *public* podem ser utilizadas.

A utilização destas funcionalidades pode ser feita de duas formas que passaremos a expor. Uma das formas de utilização é através da elaboração de um pedido *http* que terá impacto na ação de um dado recurso. Para fazer este pedido, será necessária a correta utilização do verbo da ação e, caso seja necessário, o envio do objecto no corpo da mensagem no formato *JSON*. A título de exemplo, para inserir um novo preço de um dado produto, é necessário utilizar o verbo *http POST* no caminho */prices*, e fornecer, no corpo da mensagem, o objecto correspondente ao preço que se pretende inserir. Para que isto aconteça, este objecto deverá ter determinados atributos bem definidos, nomeadamente: o identificador do produto correspondente ao preço, o seu estado, o preço de venda, a percentagem de desconto, a regra de preço a aplicar, e, finalmente, a data a partir da qual este preço estará activo. O pedido e resposta resultantes desta ação são:

```
Request;
POST /prices HTTP/1.1
HOST: http://localhost:3000

{
    "ProductID": "3b8c4c8a-92fe-4001-92b8-bee06d634c08",
    "Status": 1,
    "FullPrice": 19.99,
    "Discount": 5.00,
    "PricingFlag": 0,
    "StartDate": 2019-9-03T00:00:00.000Z,
}

Response:
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Date: Tue, 02 Set 2019 17:21:04 GMT
```

Esta forma de utilização do sistema é direccionada a utilizadores técnicos, que irão consumir o sistema desenvolvido e fazer a integração do mesmo com outros. Como, por exemplo, um serviço responsável pelo *checkout* do carrinho de compras numa plataforma *E-Commerce*.

A outra forma possível de utilização do sistema é através da interface gráfica disponibilizada pela ferramenta *Swagger*. Esta ferramenta faz a recolha de todas as funcionalidades disponibilizadas pelo sistema, traduzindo-se na pesquisa por todos os métodos públicos e etiquetados nos controladores. Na figura seguinte, é apresentada a interface gráfica gerada:

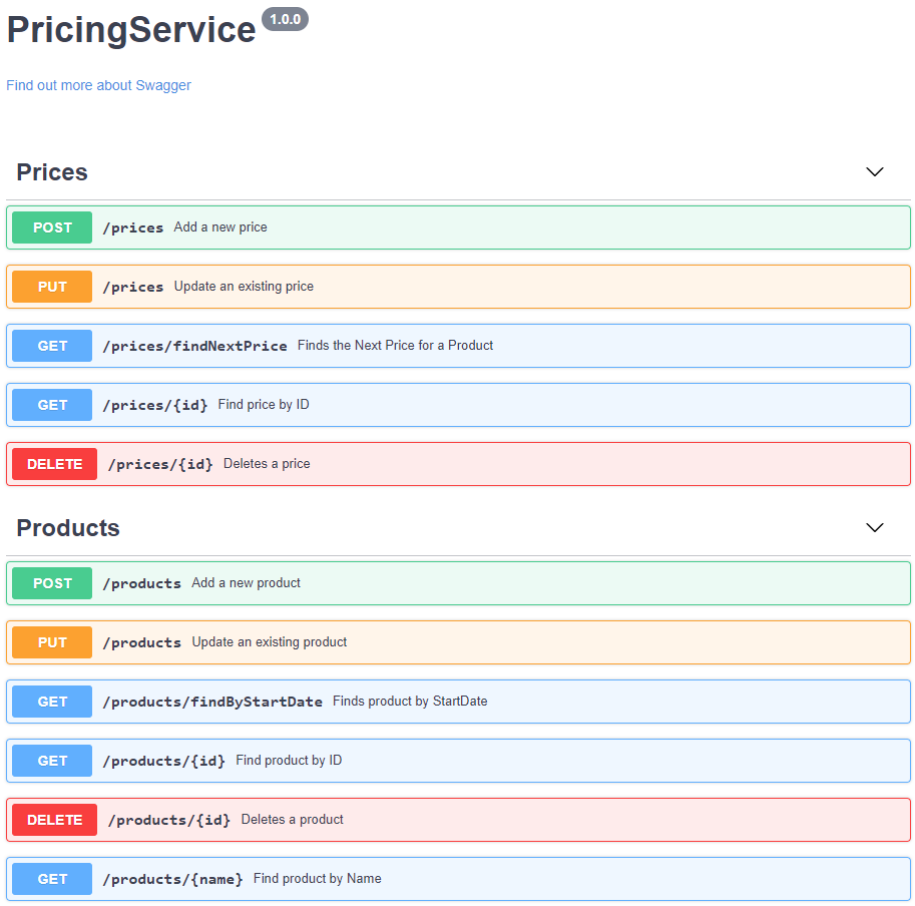


Figura 32: Interface representante dos endpoints disponíveis na aplicação

Observando a figura 32, percebe-se que a interface gráfica gerada está dividida em dois blocos distintos. Cada um destes blocos apresenta os *endpoints* presentes nos controladores existentes, *PricesController* e *ProductsController*. Estes *endpoints* contêm os métodos implementados que, por sua vez, têm a variável de acesso *public* estando, assim, disponíveis publicamente.

Cada um dos *endpoints* tem, também, discriminadas as informações gerais de informação relativas ao mesmo, os parâmetros de entrada necessários à sua utilização e os códigos e corpo de resposta que podem ser retornados.

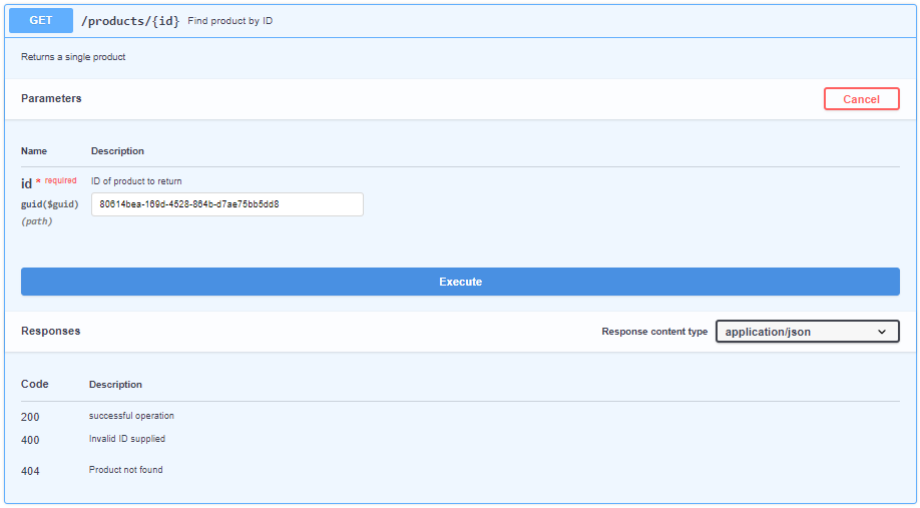


Figura 33: Exemplo da interface de pesquisa de produto pelo identificador

Para além das funcionalidades já descritas oferecidas pela ferramenta *Swagger*, a funcionalidade mais importante desta ferramenta é a execução dos pedidos. Esta ferramenta faz a recolha de todos os dados inseridos no formulário do *endpoint*, constrói um pedido *http* e envia o mesmo para o servidor. O pedido construído pela ferramenta apresenta-se com a seguinte estrutura:

```
curl -X
GET "http://localhost:3000/products/3b8c4c8a-92fe-4001-92b8-bee06d634c08/"
-H "accept: text/plain"
```

Desta forma, é possível para um utilizador não-técnico utilizar o sistema sem qualquer tipo de conhecimentos técnicos ou tecnológicos, tendo apenas de seguir as informações apresentadas pela interface gráfica.

5

PROVA DE CONCEITO

Um projecto de desenvolvimento de software apresenta várias versões antes de ser lançado oficialmente e deverá ser devidamente testado de modo a validar se os objectivos definidos no início do projecto foram, ou não, atingidos.

Através de uma prova de conceito, *Proof-of-Concept* (POC) em inglês, é possível antecipar alterações no projecto devido à realização de testes durante o desenvolvimento do mesmo, com vista a validar os pressupostos iniciais do plano (66).

Para este projeto de investigação, a prova de conceito realizou-se através de uma análise SWOT sobre a solução implementada e apresentada no capítulo 4.

Este capítulo tem como objectivo analisar os princípios teóricos que servem de base a provas de conceito, seguindo-se pela exposição da análise SWOT (*Strengths, Weaknesses, Opportunities, Threats*). O desenvolvimento desta análise é enquadrado na metodologia de investigação definida inicialmente, *Design Research*, avaliando a solução desenvolvida de modo a perceber se os requisitos iniciais foram cumpridos.

ANÁLISE SWOT

De modo a orientar um projecto pelo seu objectivo inicial e garantir que este seja atingido, as organizações utilizam a análise SWOT como prova de conceito (67). A análise SWOT está divididas em 4 partes principais: forças (*strengths*), fraquezas (*weaknesses*), oportunidades (*opportunities*) e ameaças (*threats*) (67). A vantagem desta análise é oferecer uma perspectiva geral e contextualizada do sistema tanto a nível da sua estrutura e da qualidade do produto oferecido (forças e fraquezas), como do seu posicionamento face ao mercado concorrencial (oportunidades e ameaças).

Segundo *Ifediora Christian Osita*, a análise SWOT é a melhor ferramenta de análise do ponto de vista estratégico, sendo fundamental para avaliar o potencial de um projecto, as suas limitações, oportunidades e ameaças externas existentes (67). Desta forma, a análise SWOT, representa uma parte essencial do planeamento estratégico de um projecto, procurando perceber os factores positivos e negativos, tanto a nível interno como externo (concorrência).

Figura 34: Esquema geral da análise SWOT

A figura 34 apresenta a matriz de foco da análise SWOT, as Forças e Fraquezas são representadas na primeira linha e, na segunda linha, as Oportunidades e Ameaças, respectivamente. Ou seja, os fatores internos aparecem representados na primeira linha e os fatores externos estarão incluídos na linha inferior.

Os quatro componentes da análise SWOT podem ser descritos da seguinte forma:

- FORÇAS: representam as qualidades do sistema e os seus fatores de diferenciação;
- FRAQUEZAS: representam as características que podem dificultar ou impedir o sucesso do sistema;
- OPORTUNIDADES: representam aspectos do sistema, num ambiente macro-económico, que podem contribuir para o sucesso do mesmo em relação à concorrência;
- AMEAÇAS: representam as condições do ambiente concorrencial ou macro-económico que trazem risco à evolução do projecto;

Assim, a análise SWOT, é utilizada nesta investigação como uma ferramenta de avaliação da solução desenvolvida e apresentada ao longo desta dissertação, tanto a nível interno como a nível externo.

ANÁLISE SWOT À SOLUÇÃO PRICINGSERVICE

A solução implementada, *PricingService*, é um Sistema de Apoio à Decisão baseado em motores de regras que é responsável pela gestão do processo de *pricing* e manutenção do catálogo de produtos. Este sistema foi desenvolvido com vista a centralizar o processo de *pricing* e, consequentemente, auxiliar na decisão do preço de venda de um produto perante factores externos. Assim, passaremos a expor abaixo os tópicos que constituem esta análise.

Os pontos fortes, as forças, presentes nesta solução são:

- A utilização de uma interface que possibilita a utilização do sistema a utilizadores não-técnicos;
- A possibilidade de integração numa arquitetura orientada a micro-serviços;
- A plataforma *.NETCORE* permite uma maior manutenção das funcionalidades existentes e adição de novas, trazendo potencial de extensão ao sistema;
- A utilização de *HTTP* e *JSON* permitem uma carga leve na comunicação e transferência de dados;
- A separação das escritas e leituras permite aumentar a capacidade de resposta do sistema;
- A utilização de cache permite poupar muitos pedidos à base de dados.

Por oposição às forças descritas acima, o sistema apresenta fraquezas que deverão ser encaradas como pontos a desenvolver num trabalho futuro de forma a fortalecer e aprimorar o sistema. Assim, as fraquezas identificadas foram:

- A utilização de ferramentas *Microsoft* requerem licenciamento para uso não-académico;
- A complexidade das regras de *pricing* definidas é baixa;
- Devido à política de cache definida, os preços têm de ser inseridos com seis horas de antecedência;
- Requer interação com a base de dados o que pode causar erros.

Considerando o contexto económico e o mercado concorrencial, definiram-se como oportunidades os seguintes pontos:

- Crescente interesse do mercado em ferramentas de análise de mercado;
- Utilização de dados referentes à concorrência de modo a efectuar uma melhor análise.

Para terminar, e aparecendo como pontos a observar ao sistema considerando o contexto externo em que está inserido, as ameaças identificadas na solução implementada foram:

- De modo a obter informações sobre a concorrência é necessária ligação à Internet;
- Devido à elevada recolha de dados o espaço ocupado para armazenar dados pode crescer imenso, aumentando os custos de infraestrutura;
- É possível que sejam recolhidos dados com erros caso estejam presentes nos dados da concorrência;
- Dependência de um *Web Crawler* para alimentar a inferência feita pelo motor de regras.

6

CONCLUSÕES E TRABALHO FUTURO

O projecto desenvolvido no âmbito desta dissertação tem como principal objectivo definir e modelar o processo de *pricing* implementado num Sistema de Apoio à Decisão. Este sistema está aliado a um motor de regras que é capaz de potenciar a posição de mercado de um retalhista. A importância de ser um Sistema de Apoio à Decisão reside no facto de, através dos dados referentes a concorrentes de mercado, este sistema conseguir auxiliar na melhor execução do processo de *pricing* de um retalhista. O objetivo deste Sistema de Apoio à Decisão passa por oferecer uma forma fácil e eficaz de automatizar o processo de *pricing*, através de uma decisão contextualizada no ambiente micro e macro-económicos onde o negócio se insere. Desta forma, é possível, para além de diminuir a taxa de esforço investida neste processo, também oferece ao retalhista vantagem concorrencial.

A primeira parte desta dissertação é dedicada ao estado da arte, de forma a perceber qual é o conhecimento desenvolvido academicamente em relação ao tema proposto nesta investigação. Foram estudados e abordados, durante o capítulo 2, os conceitos de Sistemas de Apoio à Decisão, o conceito de processo de *pricing*, a definição de motores de regras e o processo de desenvolvimento de *software*, de modo a avaliar qual o panorama actual relativo a estes conceitos.

Após a análise dos conceitos essenciais a este projeto de investigação, e ao estudo actual feito relativamente a estes temas, expõe-se durante o capítulo 3, a metodologia de investigação e as ferramentas de desenvolvimento que irão sustentar o processo de aplicação prática destes fundamentos acima pesquisados.

Desta forma, com base na informação recolhida, o motor de regras escolhido utiliza o algoritmo *forward chaining*. Este problema resume-se a ter um conjunto inicial de dados, sobre o qual se pretende inferir qual o preço de venda final a ser aplicado a um produto. Esta inferência é feita com base nas regras de *pricing* implementadas. Após reunidas estas condições, o motor de regras irá executar o ciclo *match/resolve/act*.

Para dar resposta às questões de investigação colocadas no capítulo de introdução, desenvolve-se, durante o Capítulo 4, a organização, estruturação e implementação da solução *PricingService*. No capítulo 1, como fio condutor deste investigação, foram colocadas quatro questões de investigação que foram abordadas e respondidas ao longo deste estudo. Assim, e de

forma a concluir esta investigação, e relacionar todos os conceitos abordados e tecnologias utilizadas, as questões de investigação serão respondidas abaixo.

QUESTÃO 1: QUAIS SÃO AS VANTAGENS DA CENTRALIZAÇÃO DO PROCESSO DE *pricing* NUM ÚNICO SISTEMA?

Para um melhor entendimento do conceito de centralização, define-se esta ideia como sendo o encapsulamento de todo um domínio apenas num local. Este local será um sistema que é correspondente à solução implementada.

O processo de *pricing* procura definir o preço de um produto e as suas oscilações, consoante o mercado e o contexto no qual se encontra inserido. Deste modo, este processo apresenta responsabilidades e limites bem definidos. A solução implementada, *PricingService*, procurou centralizar este processo numa única solução, tendo como responsabilidades a gestão de todo o catálogo de produtos e preços de um retalhista e, também, o processo de apoio à decisão implementado.

As principais vantagens da centralização de um dado domínio num único sistema são: a maior facilidade na construção e manutenção do código-fonte; um serviço estruturalmente mais organizado, devido à melhor definição das suas responsabilidades; uma menor dependência tecnológica; a possibilidade de alocação de recursos em situações de maior carga; e, por último, uma maior facilidade de integração com outros sistemas.

Para além das vantagens acima mencionadas, uma das mais-valias deste sistema, é a facilidade de integração com outros sistemas, nomeadamente plataformas de *e-commerce*. Estas plataformas podem ser vistas como o retalhista virtual que irá utilizar este serviço. Considerando o aumento do número de plataformas deste tipo, actualmente, esta facilidade na integração é uma vantagem que confere adaptabilidade e versatilidade ao sistema.

QUESTÃO 2: QUAIS SÃO AS PRINCIPAIS CARACTERÍSTICAS E FUNCIONALIDADES QUE O SISTEMA DEVE FORNECER E QUAIS AS QUE FORNECE?

As características e funcionalidades que o sistema deve fornecer, foram definidas com base na investigação realizada relativamente a Sistemas de Apoio à Decisão. Estas características estão inerentemente relacionadas com as vantagens que este sistema confere ao utilizador, nomeadamente a facilidade da utilização, do manuseamento, a automatização do processo e a estruturação da definição de preço através de uma decisão fundamentada.

Assim, as principais características e funcionalidades que se propõe conferir ao sistema inicialmente e que, após o seu desenvolvimento, se conferem são:

- Gerir catálogo de produtos;

- Gerir catálogo de preços;
- Efectuar o apoio à definição do preço de venda de produtos;
- Centralizar o processo de *pricing*;
- Agregar todos os dados recolhidos referentes aos concorrentes de mercado;

QUESTÃO 3: QUAIS SÃO AS VANTAGENS E DESVANTAGENS DA APLICAÇÃO DE REGRAS DE *pricing* DINAMICAMENTE?

As regras de *pricing* definidas, *PricingRules*, são utilizadas durante a execução do motor de regras. Estas regras representam a lógica de negócio do sistema implementado, estando definidas no seu núcleo e sendo essenciais para todo o sistema. Permitem, do seu lado, uma maior versatilidade no sistema e fundamentar o apoio à decisão inicialmente proposto.

Assim sendo, as suas principais vantagens da utilização de regras de *pricing* dinamicamente, são:

- Definição de estratégias de *pricing* bem definidas;
- Utilização de dados de mercado de modo a melhor suportar a decisão;
- Não necessitar de intervenção humana;

Em oposição, e como pontos que podem ser desenvolvidos no futuro, as desvantagens identificadas da utilização de regras de *pricing* dinâmicas, são:

- Dependência de dados que podem ser mal recolhidos por depender de informação externa;
- Dependência de uma ferramenta externa para obter dados de mercado.

QUESTÃO 4: DE QUE FORMA UM SISTEMA DE APOIO À DECISÃO PODE MELHORAR O PROCESSO DE *pricing* DE UM RETALHISTA?

O Sistema de apoio à Decisão implementado durante esta investigação permite, através dos dados recolhidos sobre a concorrência e das estratégias de *pricing* implementadas, automatizar o processo de definição de preço tornando-o mais rápido, simples e eficaz. A decisão que decorre da aplicação deste Sistema de Apoio à Decisão é uma decisão fundamentada e contextualizada, conferindo, muitas vezes vantagens competitivas.

Assim, este Sistema de Apoio à Decisão, permite:

- utilizar os dados recolhidos para definir um preço mais competitivo face ao mercado;

- ter uma visão externa do mercado que é incluída na decisão do preço;
- a definição final do preço automática e não manualmente;
- a possibilidade de definir estrategicamente regras de *pricing* diferentes para produtos distintos;
- uma rápida adaptação ao contexto do mercado externo.

TRABALHO FUTURO

Face ao resultado final apresentado deste projeto de investigação , foi necessário refletir acerca dos pontos que precisam de ser refinados e evoluídos sobre a solução final. De modo a expor uma perspectiva de evolução contínua, existem certos pontos que parecem ser o seguimento lógico e estruturado do trabalho futuro a desenvolver no âmbito deste sistema:

Assim, e como pontos que parecem relevantes para o desenvolvimento futuro deste estudo, propõe-se investigar:

- O aumento da complexidade das regras de *pricing* definidas;
- A implementação da recolha de dados utilizando filtros mais específicos;
- A avaliação das vantagens da migração para uma base de dados não-relacional;
- A implementação de um *message-bus* para reagir mais rapidamente a factores externos.

A solução apresentada como resposta aos objectivos propostos é considerada um POC, *Proof-of-Concept*, apresentando aspectos bastantes positivos face aos objectivos a atingir. A análise SWOT revelou fraquezas na solução proposta que devem ser desenvolvidas futuramente de forma a serem suprimidas, tal como ameaças externas que devem ser mitigadas.

Para finalizar, propõe-se a revisitação de conceitos como Sistemas de Apoio à Decisão, Motores de Regras e Desenvolvimento de Software, como forma de manter esta solução actualizada e de enriquecer as vantagens e funcionalidades da mesma.

BIBLIOGRAFIA

[1] Lauretta Gyedua Shardow and Rose-Mary Owusuaa Mensah. A proposed harmonisation framework for e-commerce websites across the globe. 2018.

[2] Evandro Sylvio Lima Sinisgalli, Ligia Maria Soto Urbina, and João Murta Alves. O custeio abc e a contabilidade de ganhos na definição do mix de produção de uma metalúrgica. *Production*, 19(2):332–344, 2009.

[3] A brief history of decision support systems. http://dssresources.com/history/dsshistoryv28.html, 2018(Accessed on 07/11/2018).

[4] Andreas Felsberger, Bernhard Oberegger, and Gerald Reiner. A review of decision support systems for manufacturing systems. In *SAMI@ iKNOW*, 2018(Accessed on 12/11/2018).

[5] Daniel J Power. Web-based and model-driven decision support systems: concepts and issues. *AMCIS 2000 Proceedings*, page 387, 2000.

[6] M Levy, D Grewal, P Kopalle, and J Hess. Emerging trends in retail pricing practice: implications for research. *Journal of Retailing*, 80(3):xiii–xxi, 2004.

[7] R. Mayer. EstratÉgias de pricing em marketing.

[8] Martin Natter, Andreas Mild, and Thomas Reutterer. Dynamic pricing support systems for diy retailers - a case study from austria. *Faculty of Commerce - Papers*, 1, 01 2014.

[9] Manfred Krafft and Murali K. Mantrala. *Retailing in the 21st century: current and future trends*. Springer, 2010.

[10] Hermann Simon, Danilo Zatta, and Martin Fassnacht. *Price management*. Springer, 1989.

[11] Sheng Li. Introducing a rule-based architecture for workflow systems in retail supply chain management, 2012.

[12] P. Vieira. Os 4 tipos de elasticidade da demanda na escolha da estratégia, 2012.

[13] Geraldo Luciano Toledo, Sergio Bandeira de Mello Júnior, et al. Política de preços e diferencial competitivo: um estudo de casos múltiplos na indústria de varejo. *Revista de Administração-RAUSP*, 41(3):324–338, 2006.

[14] Michael Levy, Dhruv Grewal, Praveen K Kopalle, and James D Hess. Emerging trends in retail pricing practice: implications for research, 2004.

[15] Robert Lewis Phillips. *Pricing and revenue optimization*. Stanford University Press, 2005.

[16] Philip Kotler and Gary Armstrong. *Principles of marketing*. Pearson education, 2010.

[17] Philip Kottler and Kevin Lane Keller. Marketing management. *Analyse, Planung, Umsetzung und*, 2006.

[18] Gilbert A Churchill Jr. *Marketing*. Editora Saraiva, 2017.

[19] What are the objectives of software engineering? - find 16 answers solutions: Learnpick resources, 2018(Accessed on 09/04/2019).

[20] Michael W Godfrey and Daniel M German. The past, present, and future of software evolution. In *2008 Frontiers of Software Maintenance*, pages 129–138. IEEE, 2008.

[21] Manny M Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124. Springer, 1996.

[22] Welf Löwe and Thomas Panas. Rapid construction of software comprehension tools. *International Journal of Software Engineering and Knowledge Engineering*, 15(06):995–1025, 2005.

[23] Nathan Reiff. Top ipos of 2018, 2019(Accessed on 14/05/2019).

[24] How the boeing737 max disaster looks to a software developer.

[25] Evolution of software architecture. http://www.entarchs.com/blog/evolution-of-software-architecture.html.

[26] Rule engines. https://www.thbs.com/thbs-insights/rule-engines, 2018(Accessed on 12/14/2018).

[27] Dennis Merritt. *Building expert systems in Prolog*. Springer Science & Business Media, 2012.

[28] José Oliveira. Utilização de motores de regras em sistemas informáticos. Master's thesis, Universidade do Minho, 10 2007.

[29] Jboss rules user guide. http://labs.jboss.com/file-access/default/members/jbossrules/freezone/docs/3.0.6/html_single/index.html, 2018(Accessed on 07/11/2018).

[30] Windows documentation - windows apps. https://docs.microsoft.com/en-us/windows/, 2019(Accessed on 03/06/2019).

[31] Visual studio documentation. https://docs.microsoft.com/pt-pt/visualstudio/?view=vs-2019, 2019(Accessed on 17/06/2019).

[32] Sql server documentation - sql server. https://docs.microsoft.com/en-us/sql/sql-server/sql-server-technical-documentation?view=sql-server-2017, 2019(Accessed on 13/06/2019).

[33] Gitlab. https://docs.gitlab.com/, 2019(Accessed on 07/06/2019).

[34] Carl Bereiter. Design research for sustained innovation. *Cognitive Studies*, 9(3):321–327, 2002.

[35] Vijay K Vaishnavi and William Kuechler. *Design science research methods and patterns: innovating information and communication technology*. Crc Press, 2015.

[36] Aspnetcore documentation. https://github.com/aspnet/AspNetCore, Aug 2019.

[37] Distributed domain layer. http://blog.trivadis.com/b/christofsenn/archive/2013/07/18/distributed-domain-layer.aspx, 2018(Accessed on 12/05/2019).

[38] Ralph Johnson and John Vlissides. Design patterns. *Elements of Reusable Object-Oriented Software Addison-Wesley, Reading*, 1995.

[39] Design patterns and refactoring. https://sourcemaking.com/design_patterns, 2018(Accessed on 09/05/2019).

[40] Implementing the repository and unit of work patterns. https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application, 2019(Accessed on 19/05/2019).

[41] Nrules documentation. https://github.com/NRules/NRules, 2019(Accessed on 27/04/2019).

[42] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Readings in Artificial Intelligence and Databases*, pages 547–559. Elsevier, 1989.

[43] Bertrand Meyer. On to components. *Computer*, 32(1):139–143, 1999.

[44] Martin Fowler. Cqrs. https://martinfowler.com/bliki/CQRS.html, Jul 2011.

[45] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[46] Zhigang Wen, Xiaoqing Liu, Yicheng Xu, and Junwei Zou. A restful framework for internet of things based on software defined network in modern manufacturing. *The International Journal of Advanced Manufacturing Technology*, 84(1-4):361–369, 2016.

[47] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[48] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Rfc 2616: Hypertext transfer protocol–http/1.1, 1999.

[49] Http response status codes, 2019(Accessed on 01/05/2019).

[50] Douglas Crockford. Rfc 4627-the application/json media type for javascript object notation (json). *Network Working Group*, 2006.

[51] Abhijit A Sawant, Pranit H Bari, and PM Chawan. Software testing techniques and strategies. *International Journal of Engineering Research and Applications (IJERA)*, 2(3):980–986, 2012.

[52] Zachary B Ratliff, D Richard Kuhn, Raghu N Kacker, Yu Lei, and Kishor S Trivedi. The relationship between software bug type and number of factors involved in failures. In *2016 IEEE international symposium on software reliability engineering workshops (ISSREW)*, pages 119–124. IEEE, 2016.

[53] Kent Beck. Embracing change with extreme programming. *Computer*, (10):70–77, 1999.

[54] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[55] Asp.net core web api help pages with swagger / openapi. https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-2.2.

[56] Tiago Macedo and Fred Oliveira. *Redis Cookbook: Practical Techniques for Fast Data Manipulation*. "O'Reilly Media, Inc.", 2011.

[57] What is caching and how it works: Aws. https://aws.amazon.com/caching/.

[58] Wei-Tek Tsai, Xin Sun, and Janaka Balasooriya. Service-oriented cloud computing architecture. In *2010 seventh international conference on information technology: new generations*, pages 684–689. IEEE, 2010.

[59] David Sprott and Lawrence Wilkes. Understanding service-oriented architecture. *The Architecture Journal*, 1(1):10–17, 2004.

[60] Web crawling. http://prowebscraping.com/web-scraping-vs-web-crawling/, Mar 2016.

[61] Carlos Castillo. Effective web crawling. In *Acm sigir forum*, volume 39, pages 55–56. Acm, 2005.

[62] Atul Adya, José A Blakeley, Sergey Melnik, and S Muralidhar. Anatomy of the ado. net entity framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 877–888. ACM, 2007.

[63] Ewan Tempero, James Noble, and Hayden Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *European Conference on Object-Oriented Programming*, pages 667–691. Springer, 2008.

[64] Vijay P Mehta. Getting started with object-relational mapping. *Pro LINQ Object Relational Mapping with C# 2008*, pages 3–15, 2008.

[65] Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.

[66] Bernd Schmidt. Proof of principle studies. *Epilepsy research*, 68(1):48–52, 2006.

[67] Ifediora Christian Osita, Idoko Onyebuchi, and Justina Nzekwe. Organization's stability and productivity: the role of swot analysis an acronym for strength, weakness, opportunities and threat. *International Journal of Innovative and Applied Research*, 2(9):23–32, 2014.

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Despacho RT - 31 /2019 - Anexo 5

Anexo 5A

Declaração relativa ao depósito de teses de Doutoramento (ou equivalente) e de trabalhos de Mestrado (Dissertações, Relatórios de Estágio, Projetos ou outros) no RepositóriUM

Nome: Nelson Arieira Parente

Correio eletrónico: nelson_parente@live.com.pt Tel./Telemóvel: 967514309

Número do Cartão de Cidadão/Bilhete de Identidade: 14818520

Tipo de trabalho académico a depositar:

- Doutoramento: Tese de doutoramento ☐ Outro ☐ Qual?
- Mestrado: Dissertação ☑ Relatório de Estágio ☐ Projeto ☐ Outro ☐ Qual?

Título do trabalho: Sistema de Apoio à Decisão baseado em regras para Pricing

Orientador(es): Prof. Dr. José Manuel Ferreira Machado

Data de conclusão:

Mestrado e área de especialização/Doutoramento e especialidade: Mestrado Integrado em Engenharia Informática

Declaro que concedo à Universidade do Minho (UMinho) o direito não-exclusivo e irrevogável de arquivar, reproduzir, comunicar e/ou distribuir através do seu repositório institucional, nas condições abaixo indicadas, a versão final do meu trabalho (acima referido), documento entregue em suporte digital, aprovada após a realização das provas de defesa pública e, quando for caso disso, após confirmação pelo(s) orientador(es) e homologação pelo presidente do júri[1] da introdução das alterações solicitadas.

Declaro que autorizo a UMinho a arquivar mais de uma cópia do documento e a, sem alterar o seu conteúdo, converté-la para qualquer formato de ficheiro, meio ou suporte, para efeitos de preservação e acesso

Declaro que o documento agora entregue é um trabalho original e que, contendo material do qual não detenho direitos de autor, obtive autorização prévia do detentor dos referidos direitos para conceder à UMinho os termos requeridos por esta licença.

Declaro também que a entrega do documento não infringe, tanto quanto me é possível saber, os direitos de qualquer outra pessoa ou entidade.

Se o documento entregue é baseado em trabalho financiado ou apoiado por organismo/financiador que não a UMinho, declaro que cumpri quaisquer obrigações exigidas pelo respetivo contrato ou acordo.

Retenho todos os direitos de autor relativos ao trabalho e o direito de o usar em trabalhos futuros, como artigos ou livros.

Concordo que esse meu trabalho seja colocado no repositório da UMinho, com o seguinte estatuto (assinale apenas um dos três estatutos):

1. ☑ Disponibilização imediata do conjunto do trabalho para acesso mundial.
2. ☐ Disponibilização do conjunto do trabalho para acesso exclusivo na Universidade do Minho durante o período de:
   ☐ 1 ano ☐ 2 anos ☐ 3 anos. Acesso mundial depois do período indicado.
3. ☐ Outro. Qual?

Se assinalei o estatuto 2 ou 3, comprometo-me a entregar, no prazo máximo de cinco dias úteis após ter tomado conhecimento, cópia do despacho que tiver sido emitido sobre o requerimento de excecionalidade por mim apresentado ao Sr. Reitor da Universidade do Minho.

Declaro ter sido informado pela UMinho que os meus dados pessoais constantes nesta declaração serão tratados com o único propósito de gerir o depósito do trabalho, a que se refere, no repositório institucional da UMinho. Fui ainda informado que poderei exercer os meus direitos, quanto à proteção de dados pessoais, junto dos Serviço de Documentação, serviço responsável pelo RepositóriUM. Tomei também conhecimento que a UMinho dispõe de Encarregado de Proteção de Dados, cujos contactos estão publicados em http://www.uminho.pt/protecaodados.

Braga/Guimarães, 25 / 12 / 2019

Assinatura:

Nelson Arieira Parente

[1] *Vide* artigo 124.º Anexo ao Despacho RT-41/2014.

Anexo 5B

Formulário para depósito legal[2] de teses de Doutoramento (ou equivalente) e de trabalhos de Mestrado (Dissertações, Relatórios de Estágio, Projetos ou outros) no RepositóriUM

Elementos de identificação

Nome do autor: Nelson Arieira Parente

Tipo de trabalho académico:

- Doutoramento: Tese de doutoramento ☐ Outro ☐ Qual?
- Mestrado: Dissertação ☑ Relatório de Estágio ☐ Projeto ☐ Outro ☐ Qual?

Título do trabalho: Sistema de Apoio à Decisão baseado em Regras para Pricing

Supervisor(es): Prof. Dr. José Machado Data da concessão do grau: 12/12/19

Mestrado e Área de Especialização/Doutoramento e Especialidade: Mestrado Integrado em Engenharia Informática

Escola/Instituto: Escola de Engenharia

Departamento/Centro de Investigação[3]: Departamento de Informática

Área disciplinar (área FOS)[4]: Engenharia e Tecnologia::Engenharia Eletrotécnica, Eletrónic

Identificador único e permanente do trabalho (TID) atribuído pelo RENATES:

Classificação

No caso de teses de doutoramento

Classificação final:

No caso de trabalhos de mestrado (Dissertações, Relatórios de Estágio, Projetos ou outros)

N.º ECTS: 45

Classificação em valores (0-20): 16

Classificação ECTS, com base no percentil (A a F):

Financiamento público

Sem financiamento público: ☑

Financiamento pelo estabelecimento que confere o grau (caso seja público): ☐

Financiamento pela Fundação para a Ciência e Tecnologia (FCT): ☐

Identificador da Bolsa FCT:

Outro financiamento público: ☐

Obs:

[2] *Vide* Decreto-Lei n.º 115/2013 e Portaria n.º 285/2015.

[3] Indicar (se aplicável) o Departamento/Centro para efeitos de associação à respetiva comunidade no RepositóriUM.

[4] Preencher de acordo com a lista apresentada na página seguinte.

Lista de Áreas Disciplinares (áreas FOS)

Agricultura, Silvicultura e Pescas	Engenharia Médica
Artes	Engenharia Química
Biotecnologia Agrária e Alimentar	Filosofia, Ética e Religião
Biotecnologia Ambiental	Física
Biotecnologia Industrial	Geografia Económica e Social
Biotecnologia Médica	História e Arqueologia
Ciência Animal e dos Lacticínios	Línguas e Literaturas
Ciências Biológicas	Matemática
Ciências da Computação e da Informação	Medicina Básica
Ciências da Comunicação	Medicina Clínica
Ciências da Educação	Nanotecnologia
Ciências da Saúde	Não Classificado
Ciências da Terra e Ciências do Ambiente	Outras Ciências Agrárias
Ciências Politicas	Outras Ciências de Engenharia e Tecnologias
Ciências Veterinárias	Outras Ciências Médicas
Direito	Outras Ciências Naturais
Economia e Gestão	Outras Ciências Sociais
Engenharia Civil	Outras Humanidades
Engenharia do Ambiente	Psicologia
Engenharia dos Materiais	Química
Engenharia Eletrotécnica, Eletrónica e Informática	Sociologia
Engenharia Mecânica	