CENTRO UNIVERSITÁRIO UNIVATES
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
CURSO DE SISTEMAS DE INFORMAÇÃO

UM ESTUDO DE CASO SOBRE A UTILIZAÇÃO DE PADRÕES DE PROJETO NA DEFINIÇÃO DE UMA ARQUITETURA DE SOFTWARE VOLTADA AO DESENVOLVIMENTO DE SISTEMAS DE GESTÃO

Douglas Franciel Lagmann

Lajeado, Novembro de 2013

Douglas Franciel Lagmann

UM ESTUDO DE CASO SOBRE A UTILIZAÇÃO DE PADRÕES DE PROJETO NA DEFINIÇÃO DE UMA ARQUITETURA DE SOFTWARE VOLTADA AO DESENVOLVIMENTO DE SISTEMAS DE GESTÃO

Trabalho de Conclusão de Curso apresentado ao Centro de Ciências Exatas e Tecnológicas do Centro Universitário UNIVATES, como parte dos requisitos para a obtenção do título de bacharel em Sistemas de Informação.

Área de concentração: ENGENHARIA DE SOFTWARE

Orientador: Pablo Dall'Oglio

Lajeado, Novembro de 2013

Douglas Franciel Lagmann

UM ESTUDO DE CASO SOBRE A UTILIZAÇÃO DE PADRÕES DE PROJETO NA DEFINIÇÃO DE UMA ARQUITETURA DE SOFTWARE VOLTADA AO DESENVOLVIMENTO DE SISTEMAS DE GESTÃO

Este trabalho foi julgado adequado para a obtenção do título de bacharel em Sistemas de Informação do CETEC e aprovado em sua forma final pelo Orientador e pela Banca Examinadora.

Banca Examinadora:

Prof. Pablo Dall'Oglio, UNIVATES - Orientador
Mestre pela Universidade do Vale do Rio dos Sinos – São Leopoldo, Brasil

Prof. Evandro Franzen, UNIVATES
Mestre pela Universidade Federal do Rio Grande do Sul – Porto Alegre, Brasil

Prof. Paulo Roberto Mallmann, UNIVATES
Mestre pela Universidade do Vale do Rio dos Sinos – São Leopoldo, Brasil

Lajeado, Novembro de 2013.

Dedico este trabalho a minha mãe e a minha namorada, em especial pela dedicação e apoio em todos os momentos difíceis.

AGRADECIMENTOS

A minha família, por todo apoio e ajuda prestada durante o período que estive cursando a faculdade, pois, sem eles, a concretização deste sonho não seria possível.

A minha namorada Bruna Duarte, pelo seu apoio e compreensão nos momentos de minha ausência, e por ser paciente comigo enquanto me dedicava a conclusão deste trabalho.

Ao meu orientador Pablo Dall'Oglio, por me auxiliar em todos os momentos, desde a escolha da proposta, desenvolvimento, até à conclusão da mesma.

RESUMO

O ambiente mercadológico atual exige das empresas mais rapidez nas entregas, mais eficiência e mais qualidade em seus produtos, aliado ao fato de a demanda por software estar cada vez maior, devido ao fato de ele estar cada vez mais presente em nossas vidas. Isso faz com que desenvolvedores necessitem produzir seus softwares dentro de um prazo menor, para conseguir atender a todas as demandas de seus clientes. A Engenharia de Software é uma disciplina que foi criada com o objetivo de fornecer um conjunto de métodos e processos que visa a auxiliar e melhorar as atividades envolvidas na construção de software, a fim de aumentar a produtividade, reduzir custos e garantir a qualidade. Sendo assim, o presente trabalho tem o objetivo de estudar um conjunto de soluções técnicas que aplicam os conceitos e práticas da Engenharia de Software, a fim de propor uma arquitetura de software para a construção de um sistema de gestão empresarial o qual tenha, como objetivo, suportar mudanças e poder evoluir com o tempo. Para a construção dessa arquitetura, optou-se por buscar soluções técnicas que tenham aderência a padrões de projeto, pois eles são as melhores práticas de soluções para problemas comuns de software, porque buscam priorizar aspectos de simplicidade e de flexibilidade no desenvolvimento, visando a facilitar a reutilização de componentes, diminuir o tempo para o desenvolvimento e propiciar a criação de sistemas flexíveis que rapidamente possam se adaptar a novas necessidades dos processos de negócios.

Palavras-chave: Engenharia de Software, Padrões de Projeto, Arquitetura de Software.

ABSTRACT

The current market environment requires companies deliver faster than ever, efficiency and quality in its products, coupled with the fact the demand for software is increasing, due to it being ever more present in our lives. This means that developers need to produce their software within a shorter time, to be able to meet all the demands of their customers. Software Engineering is a discipline that was created with the goal of providing a set of methods and processes to assist and improve the activities involved in building software in order to increase productivity, reduce costs and ensure quality. Thus, the present work aims to study a set of technical solutions, applying the concepts and practices of software engineering in order to propose a software architecture for building a business management system, which has the objective support changes and to evolve with time. For the construction of this architecture we chose to seek technical solutions that have adherence to design standards, as they are best practice solutions to common software problems, because they seek to prioritize aspects of simplicity and flexibility in development, to facilitate reuse components, reducing the time to develop and facilitate the creation of flexible systems that can quickly adapt to the changing needs of business processes.

Keywords: Software Engineering, Design Patterns, Architecture.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

CRM-	Customer Relationship Management
CT-e-	Conhecimento de Transporte Eletrônico
DAO-	Data Access Object
DLL-	Dynamic-link library
DTO-	Data Transfer Object
ER-	Entidade Relacionamento
GoF-	Gang of Four
HTML-	Hyper Text Markup Language
HTTP-	Hypertext Transfer Protocol
IEEE-	Institute of Electrical and Electronics Engineers
IIS-	Internet Information Services
KIS-	Keep It Simple
MSMQ-	Microsoft Message Queue Server
MVP-	Model View Presenter
NF-e-	Nota Fiscal Eletrônica
NFS-e-	Nota Fiscal de Serviço Eletrônica
OMG-	Object Management Group
OMT-	Object Modeling Tecnique
OOSE-	Object-Oriented Software Engineering
ORM-	Mapeamento Objeto Relacional
PAF-ECF-	Programa Aplicativo Fiscal
PCP-	Planejamento e Controle da Produção
SOA-	Service-Oriented Architecture

SOAP-	Simple Object Access Protocol
SPED-	Escrituração Fiscal Digital
SQL-	Structured Query Language
SQL-	Structured Query Language
SWEBOK-	Software Engineering Body of Knowledge
TCP-	Transmission Control Protocol
TI-	Tecnologia da Informação
UML-	Unified Modeling Language
WCF-	Windows Comunication Fundation
XML-	Extensible Markup Language
XP-	Extreme Programming
XSLT-	Extensible Stylesheet Language for Transformation

SUMÁRIO

1 INTRODUÇÃO

Na atualidade, estamos, frequentemente, utilizando algum tipo de software, seja de maneira direta - por meio de um computador -, ou mesmo indireta - por meio de produtos eletrônicos, como celulares, *smartphones*, TVs, dentre outros. Um software pode ser encontrado e utilizado em diversos locais, como indústrias, governos, escolas, universidades, serviços de saúde e bancos, onde é utilizado para agilizar e automatizar as operações e os processos. (SOMMERVILLE, 2007)

Todos os aspectos da produção de um software, tais como, especificação, desenvolvimento, manutenção, gerenciamento e evolução, estão relacionados com a Engenharia de Software que trata da aplicação de ferramentas, métodos e processos que visam à organização, à produtividade e à qualidade do software, além de oferecer mecanismos de planejamento e gerenciamento.

A primeira definição de Engenharia de Software foi proposta em 1968, em meio a uma conferência que tratava sobre a “crise de software”, que se originara devido à introdução do hardware baseado em circuitos integrados, o que proporcionou o aumento do poder computacional, tornando necessária a criação de novas técnicas e métodos para controlar a maior complexidade dos grandes sistemas de software que surgiram a partir de então.

Nessa conferência, o alemão Fritz Bauer definiu que a "Engenharia de Software é a criação e a utilização de sólidos princípios de engenharia, a fim de obter software de maneira econômica, mas que seja confiável e que trabalhe em máquinas reais". (PRESSMAN, 2010)

A Engenharia de Software, assim como qualquer outra Engenharia, tem como base o foco pela qualidade, adotando filosofias que buscam continuamente o aperfeiçoamento. O alicerce da Engenharia de Software são os processos que descrevem o que deve ser feito. Acima dos processos, estão os métodos os quais fornecem técnicas que respondem o modo de como devem ser construídos os softwares e, junto aos processos e aos métodos, estão as ferramentas que auxiliam a automatizar os processos e os métodos.

Conforme os processos, métodos e ferramentas que constituem a Engenharia de Software foram evoluindo com o passar do tempo, houve também um aumento considerável pela demanda de softwares no mercado.

Além disso, eles vêm assumindo um papel cada vez mais importante dentro das organizações. Assim, é necessário que o mesmo esteja bem alinhado ao negócio, pois ele pode ser responsável pelo aumento da produtividade e pela redução de custos, o que acaba agregando mais valor ao negócio. Para tanto, softwares devem ser produzidos com a utilização de metodologias que garantam não somente o atendimento aos requisitos de negócio, mas também que possibilitem a adaptação às constantes mudanças e condições do mercado.

Para conseguir atender a necessidade de entregar um software de qualidade, cumprindo os requisitos, dentro do custo e do prazo estimado, é necessário que as empresas de desenvolvimento utilizem e apliquem um processo organizado de Engenharia de Software, sendo que, para isso, existem diferentes abordagens que podem ser utilizadas, como o modelo em cascata, incremental e espiral.

O modelo em cascata, também chamado de modelo clássico, consiste em dividir e sequenciar sistematicamente todas as etapas da construção de um software, dessa forma, a etapa seguinte se inicia somente quando a anterior já tenha terminado. Esse modelo é um dos paradigmas mais antigos da Engenharia de Software.

Segundo Pressman (2010), o modelo de desenvolvimento em cascata vem sendo criticado nas duas últimas décadas em relação a sua eficácia, pois, por melhor que sejam definidas as especificações e os requisitos, os mesmos mudam constantemente de acordo com as necessidades das diferentes empresas e dos usuários desse software. Para possibilitar um *feedback* mais rápido aos clientes, as empresas de desenvolvimento de software vêm adotando o modelo incremental de desenvolvimento.

O modelo incremental organiza as fases do modelo em cascata de uma maneira mais interativa, pois permite que o cliente receba uma versão inicial do projeto, que atenda suas principais necessidades num prazo de tempo mais curto. Os demais requisitos do projeto vão sendo desenvolvidos e refinados de forma incremental, sendo entregues por meio de várias versões.

Nesse modelo, as etapas do ciclo de vida do software vão se intercalando em diversas fases. Cada fase corresponde a uma versão do software que é entregue ao cliente e atende uma parte do total dos requisitos do software. Dessa forma, conforme o cliente compreender melhor seu problema, ocorre uma evolução no software, portanto esse modelo está mais preparado para atender às mudanças do negócio.

A abordagem incremental pode ser mais eficaz que o modelo em cascata, já que os clientes têm uma resposta imediata, mas essa abordagem também têm seus problemas, porque, nesse modelo, os processos não são visíveis, pois os requisitos vão evoluindo com o tempo. Outro problema que pode ocorrer em softwares maiores é a dificuldade de incorporar novas mudanças, pois, como o software passa por alterações contínuas, se o núcleo do software não for bem estruturado, ele tenderá a corromper.

Portanto, indiferente do modelo de processo de software adotado no desenvolvimento, é importante considerar e elaborar cuidadosamente a arquitetura de um software, pois, quanto maior e mais complexo for o software, consequentemente mais processos e funcionalidades ele terá, sendo assim, a definição da arquitetura desse software deverá ser a melhor possível.

Uma forma de se elaborar uma arquitetura que seja robusta e flexível, que possua qualidade, é com a utilização e a adoção de técnicas e boas práticas que a Engenharia de Software sugere, sendo que uma dessas técnicas é a utilização de Padrões de Projeto, ou também conhecido por *Design Patterns.*

A utilização de padrões de projeto é voltada à arquitetura de software e, portanto, uma das mais complexas de ser implementada. O primeiro autor que começou a abordar o assunto de padrões de projetos foi o arquiteto, matemático e urbanista Christopher Alexander, que teve a ideia de documentar os pontos em comum de projetos de arquitetura e urbanismo, para possibilitar a utilização em novos projetos.

Alexander define *Design Patterns*, em sua obra *A Pattern Language: Towns, Buildings, Constructions* (ALEXANDER, 1977 apud GAMMA et al., 2008, p. 10), como: “cada padrão descreve um problema no nosso ambiente e o cerne de sua solução, de tal forma que se possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira”.

Os princípios de Alexander, mesmo direcionados à arquitetura, foram trazidos posteriormente para o desenvolvimento de software, porém isso apenas ocorreu em 1987, quando os programadores Kent Beck e Ward Cunningham propuseram os primeiros padrões de projetos para a área da ciência da computação.

Design Pattern é descrito como uma solução para um problema, no qual é identificado o que constantemente se repete em diversas situações. Sendo que essa solução foi baseada em experiências anteriores, comprovadamente testadas em problemas que ocorrem com frequência em situações específicas.

Os padrões de projetos devem ser vistos como uma forma de trazer clareza para um sistema, além de permitir melhorá-los, promovendo a extensibilidade e a reusabilidade, pois eles são altamente estruturados, e documentados em um modelo que identifica o que é necessário para entender o problema e a solução.

De acordo com Shalloway e Trott (2004), os padrões de projeto têm como princípio a reutilização de ideias que foram bem sucedidas, reaproveitando não apenas o código, mas também a experiência adquirida, resultando numa solução confiável, mais robusta, reduzindo a complexidade e simplificando a manutenção.

1.1 Motivação

É comum que, no início de todo projeto, devam ser tomadas decisões de arquitetura relacionadas à divisão da solução em camadas, bem como à seleção de soluções técnicas que atendam demandas específicas, tais como apresentação e persistência.

A existência de catálogos de padrões de projeto, de certa forma, auxilia na tomada desse tipo de decisão, pois estes apresentam soluções abstratas para problemas frequentes no

desenvolvimento de um software. Mesmo assim, a elaboração da arquitetura de um software é um problema complexo, devido à variabilidade das soluções disponíveis e às diferentes combinações que podem ser realizadas com as mesmas.

Em face desses desafios, faz-se necessário documentar as decisões que levam um arquiteto de softwares a realizar determinadas escolhas durante o projeto da arquitetura de um software. Esse tipo de documentação, geralmente, é escasso e, mesmo quando produzido, muitas vezes, o seu acesso é restrito às empresas desenvolvedoras.

Em virtude do desenvolvimento de um novo software de gestão empresarial para a empresa ABC Informática LTDA, que atua no ramo de desenvolvimento e consultoria de sistemas de gestão empresarial há mais de 20 anos, será necessário o desenvolvimento de uma nova arquitetura de software, combinando diferentes soluções técnicas de maneira a dar suporte ao desenvolvimento de software em camadas, com fácil manutenibilidade, buscando a conformidade com os principais *Design Patterns* catalogados.

A empresa em questão tem necessidade de ter um sistema que possa atender a todas as necessidades das legislações vigentes, como Nota Fiscal Eletrônica (NF-e), Nota Fiscal de Serviço Eletrônica (NFS-e), Conhecimento de Transporte Eletrônico (CT-e), Escrituração Fiscal Digital (SPED), Programa Aplicativo Fiscal (PAF-ECF), além de permitir fazer customizações solicitadas pelos clientes, utilizando tecnologias que permitam a integração do sistema nos mais variados dispositivos, como *tablets* e *smartphone*s, possibilitando visualizar e disponibilizar informações na internet.

O presente trabalho irá documentar e analisar as decisões tomadas para o desenvolvimento deste projeto, buscando estudar diversas soluções técnicas que possam atender as necessidades do projeto, analisando as que agregam um maior valor ao produto, para possibilitar acomodar as futuras modificações e necessidades do sistema.

1.2 Objetivos

1.2.1 Objetivo geral

O objetivo do presente trabalho é escolher, analisar e documentar a utilização de diferentes soluções técnicas visando à formação de uma arquitetura central para o desenvolvimento de um software de gestão, conforme a aderência dessas soluções à *Design Patterns* previamente estabelecidos.

1.2.2 Objetivos específicos

Dentre os objetivos específicos do presente trabalho, podemos citar:

- Buscar soluções técnicas para o desenvolvimento do software;
- Analisar e classificar as soluções técnicas em relação à *Design Patterns*;
- Escolher as soluções técnicas e documentar a arquitetura do software.

1.3 Organização do trabalho

Para a melhor compreensão do presente trabalho, seus capítulos foram divididos em uma ordem de apresentação.

No Capítulo 1, é realizada a contextualização do trabalho e apresentada sua motivação, bem como os objetivos. Nesse sentido, são apresentadas as origens da Engenharia de Software e os principais modelos de processos de software. Além disso, é proposta a utilização de *Design Patterns* para aplicar conceitos que venham a resultar uma arquitetura de software viável e que possibilite a adequação das mudanças que o mercado impõe.

No capítulo 2, é apresentado todo o embasamento utilizado para o desenvolvimento da proposta, explorando os referenciais teóricos que tratam de assuntos como a Engenharia de Software, Qualidade de Software, Processos de Software, a importância da definição de uma

arquitetura sólida e os benefícios da reutilização de código num sistema, assim como é apresentado o conceito de padrões de projetos, sua origem, seus benefícios e suas classificações.

No capítulo 3, é apresentada a metodologia utilizada para o desenvolvimento deste trabalho. Pelo fato de se tratar de um estudo de caso com uma abordagem qualitativa, faz-se necessária a familiarização com o problema, buscando conhecer os fatos relacionados ao tema, descrevendo e documentando todos os passos deste estudo.

No capítulo 4, é apresentada a empresa na qual será desenvolvido o trabalho e alguns detalhes de sua equipe de desenvolvimento, após isso, são apresentadas as definições estabelecidas em relação ao detalhamento da arquitetura que está sendo proposta.

No capítulo 5, é apresentado o protótipo que foi desenvolvido para validar a arquitetura que é proposta e a avaliação dos resultados obtidos com o estudo realizado, buscando verificar se os objetivos propostos foram alcançados.

E, por final, é apresentada a conclusão obtida diante do estudo realizado, também são descritas as justificativas dos resultados obtidos e feitas as demais considerações.

2 REFERENCIAL TEÓRICO

Neste capítulo serão descritos os conceitos e fundamentos relacionados à pesquisa bibliográfica realizada, a fim de abordar e esclarecer conceitos e definições estudados e relacionados ao presente trabalho.

2.1 Engenharia de Software

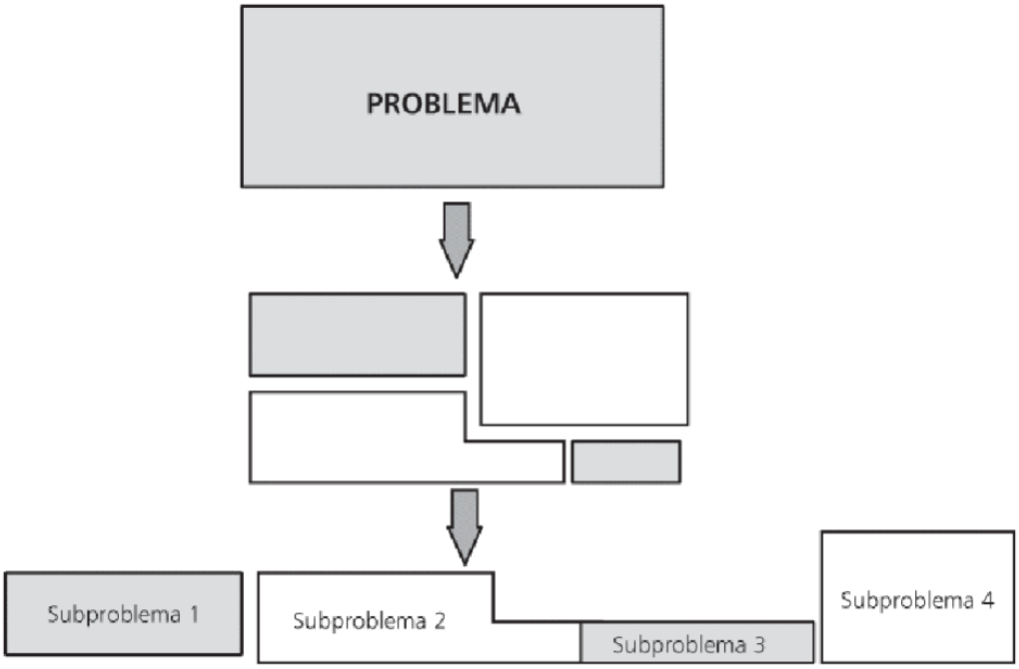
Hoje em dia, o software é encontrado em vários lugares ao nosso redor e reconhecemos que seu papel é tornar nossas vidas mais confortáveis e melhores. Dessa forma, a Engenharia de Software sugere a aplicação de um conjunto de métodos e técnicas que aplicam os conhecimentos da computação e de computadores para resolver problemas, utilizando a tecnologia como ferramenta para implementar a solução, sugerindo boas práticas que asseguram que o software resultante tenha uma contribuição positiva em nossas vidas, para tornar os produtos mais confiáveis, seguros, úteis e acessíveis.

A Engenharia de Software tem o intuito de fornecer uma estrutura que auxilie na construção de um software, cujo foco é garantir a qualidade. Para viabilizar isso, a Engenharia de Software engloba todas as fases da produção de um sistema, desde a especificação, desenvolvimento, gerenciamento até a evolução. (PRESSMAN, 2010)

De acordo com Pressman (2010), “reconhecer os problemas e as suas causas e desmascarar os mitos do software são os primeiros passos em direção à solução”.

Pfleeger (2004) também sugere que o primeiro passo necessário a ser feito para se encontrar a solução de um problema é conseguir identificá-lo, pois a maioria dos problemas é complexa e, algumas vezes, difícil de resolver. Então é preciso analisar o problema, dividindo-o em partes menores para facilitar o seu entendimento, assim consegue-se descrever um problema maior, com um conjunto de pequenos problemas e suas inter-relações. Todo esse processo de análise pode ser identificado na Figura 1.

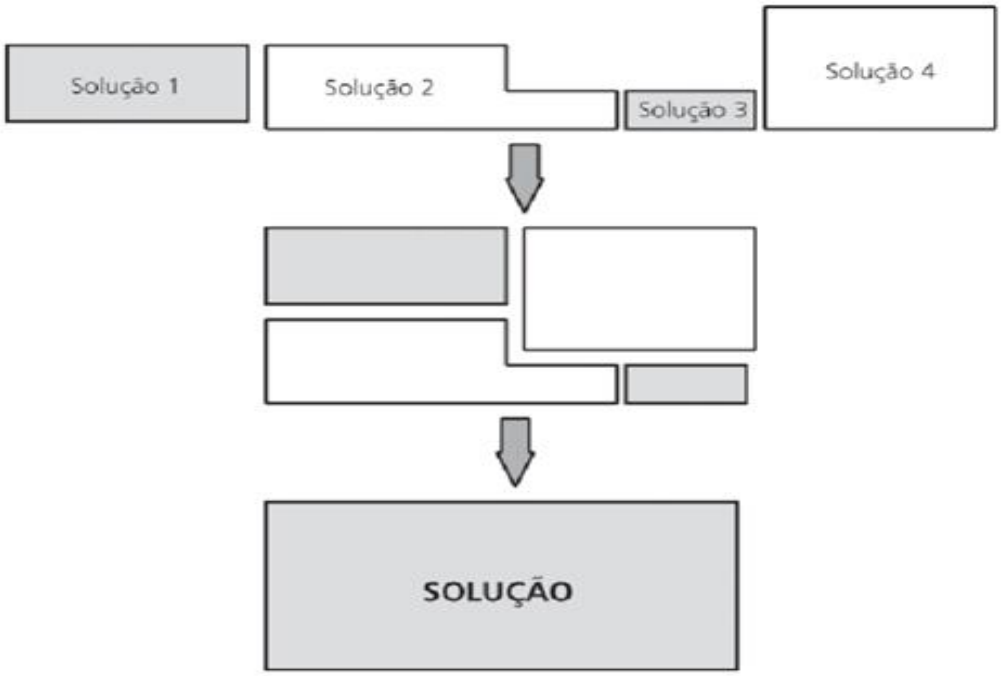
Figura 1 - Processo da análise



Fonte: Pfleeger (2004, p. 2).

Após a realização da análise do problema, deve-se efetuar a síntese da solução. Esse processo se assemelha ao anterior, porém em ordem inversa, pois seu objetivo é elaborar a solução para uma parte do problema e relacionar esses conjuntos de soluções em uma estrutura maior. A Figura 2 ilustra o processo de síntese.

Figura 2 - Processo de síntese



Fonte: Pfleeger (2004, p. 3).

Pfleeger (2004) ressalta que a dificuldade de encontrar a solução individual de uma parte do problema é a mesma de organizar o conjunto de soluções em uma única solução, isso também se aplica à análise do problema. Porque, em alguns casos, são as relações entre as pequenas partes (subproblemas e soluções individuais) que vão nos demonstrar como resolver o problema como um todo, sendo assim, as relações entre as pequenas partes têm a mesma importância quanto às próprias partes menores.

Pressman (2010) afirma que a Engenharia de Software contempla todo um conjunto de ações que ajudam no desenvolvimento de software, incluindo questões técnicas da implementação, o gerenciamento de todas as etapas da construção, teorias e métodos que possam apoiar a criação do mesmo.

O surgimento do software ocorreu na década de 60, tendo como origem o aumento pela procura por sistemas cada vez mais robustos, com mais recursos e com funções mais complexas. Tais sistemas necessitavam de uma estrutura mais organizada e documentada, com controle de qualidade e mais agilidade na construção, tornando necessária a criação de novas técnicas e modelos para controlar e gerenciar esses sistemas, pois a falta desses modelos e técnicas na sua construção implicavam atrasos nas entregas. Dessa forma, o custo superava as previsões, tornava difícil as manutenções, além disso, os sistemas eram instáveis e possuíam um desempenho insatisfatório. (SOMMERVILLE, 2007)

Dentre as mais diversas definições de Engenharia de Software, é possível observar que todas contemplam um conjunto de três elementos básicos: métodos, ferramentas e procedimentos.

O método representa o caminho pelo qual se atinge um objetivo, ferramenta é um instrumento ou sistema que é utilizado para auxiliar ou melhorar a execução das tarefas e, por fim, o procedimento é uma combinação entre métodos e ferramentas, a fim de produzir um resultado específico.

> Os fundamentos científicos para a Engenharia de Software envolvem o uso de modelos abstratos e precisos que permitam ao engenheiro especificar, projetar, implementar e manter sistemas de software, avaliando e garantindo sua qualidade, …deve oferecer mecanismos para se planejar e gerenciar o processo de desenvolvimento. (TONSIG, 2008, p 65)

O *Software Engineering Body of Knowledge* (SWEBOK - Corpo de Conhecimento da Engenharia de Software) é um guia mantido pelo *Institute of Electrical and Electronics Engineers* (IEEE - Instituto de Engenheiros Eletricistas e Eletrônicos), que trata de assuntos relacionados à Engenharia de Software, organizando-a em dez áreas de conhecimento distintas. (ABRAN et al., 2004)

Sendo elas:

- **Requisitos de software**: tem o objetivo de identificar as necessidades do cliente (problemas), definindo o que o sistema deve fazer, suas restrições, através de quatro processos: identificação, análise e negociação, especificação e documentação, além da validação;

- **Projeto de software**: trata das questões de transformar os problemas em soluções, ou seja, faz o detalhamento de como serão resolvidos os requisitos em relação ao software;

- **Construção ou implementação de software**: área do conhecimento responsável pela implementação/codificação do software;

- **Teste de software**: tem objetivo de validar o software, verificando se os requisitos especificados foram devidamente atendidos;

- **Manutenção de software**: trata de questões para dar continuidade ao software, como correções de erros, melhorias e novas funcionalidades;

- **Gerência de Configuração**: busca identificar, documentar, controlar e aditar as mudanças ocorridas durante o desenvolvimento;

- **Gerência de Engenharia**: tem o objetivo de fazer o gerenciamento do projeto da construção do software, envolvendo planejamento, coordenação, medição, monitoração, controle e documentação;

- **Processos de Engenharia**: estabelece uma sequência prática para auxiliar o desenvolvimento do software;

- **Ferramentas e Métodos**: estabelece um conjunto de ferramentas e estruturas para auxiliar o desenvolvimento e manutenção do software de forma sistemática e automatizada;

- **Qualidade**: trata de verificar se o software produzido atende a suas especificações, de acordo com os padrões de qualidade estabelecidos.

2.2 Qualidade de software

Com o passar do tempo, aliado ao aumento da demanda por software, ocorreu não somente um aumento da quantidade de software, mas também um aumento na procura por software de qualidade, tornando este um quesito indispensável para empresas de desenvolvimento de software.

A ideia da definição de qualidade pode parecer intuitiva, porém é bastante subjetiva, podendo ter diversas interpretações. No contexto da Engenharia de Software, qualidade é definida pela conformidade do software a seus requisitos, ou seja, o quanto ele se adéqua a sua finalidade, suprindo as necessidades do negócio, como as necessidades dos usuários. Um software de qualidade deve ter as características certas e os atributos certos.

Pressman (2010) definiu qualidade de software como:

> A conformidade a requisitos funcionais e de desempenho que foram explicitamente declarados, a padrões de desenvolvimento claramente documentados, e a características implícitas que são esperadas de todo software desenvolvido por profissionais. (PRESSMAN, 2010, p. 580)

A Engenharia de Software define que um software de qualidade é aquele que atende a todos os requisitos, dentro do prazo, tempo e custo estipulados. Para atender a essas necessidades, a fim de cumprir a qualidade do software, empresas de desenvolvimento devem adotar métodos e processos de desenvolvimento bem definidos e uma arquitetura cuidadosamente projetada, pois a qualidade também está diretamente relacionada com a maneira de como o software é construído.

Pressman (2010) diz que: "independentemente do tamanho, complexidade ou domínio de aplicação, o software de computador vai evoluir com o tempo". Com isso o autor afirma que todos os softwares acabam sofrendo alguma mudança com o passar do tempo. Essas mudanças podem ser originadas por diversos fatores, como alguma mudança na legislação, no aprimoramento de um processo da empresa, ou a necessidade de alterações dos usuários que utilizam o sistema. Sendo assim, é visível que a qualidade de um software, deve contemplar também a possibilidade deste suportar mudanças e poder evoluir com o tempo.

A arquitetura de um software representa grande parte da qualidade do mesmo, visto que uma arquitetura mal definida pode resultar em um software de má qualidade, por isso a arquitetura de software engloba os principais atributos de qualidade, como acessibilidade, confiabilidade, segurança, possibilidade de modificação e desempenho. (PRESSMAN, 2010)

Para Pressman (2010), um software que possui um código confuso e/ou complicado, que tenha uma documentação precária, ou nem possua documentação, que não tenha casos de testes e no qual as modificações foram mal geridas: esses fatores acabam sendo exemplos de casos de má qualidade de software.

A qualidade de software é uma complexa combinação de fatores que variam de acordo com diferentes sistemas e clientes. Podem ser definidos dois tipos de padrões para auxiliar no processo de garantia da qualidade (SOMMERVILLE, 2007):

- Padrões de processos: definem processos que devem ser seguidos no desenvolvimento de software. Incluem definições de processos, de especificação, de projetos e de validação, e uma descrição dos documentos que devem ser escritos ao logo dos processos;

Padrões de produto: se aplicam ao produto de software em desenvolvimento, incluem padrões de documentos, padrões de codificação, padrões de comportamento do software.

2.3 Processos de software

A maneira como um software é construído pode ser considerada semelhante ao modo da construção de uma casa, nos dois casos é preciso seguir uma sequência de etapas para completar um conjunto de tarefas. Por exemplo, primeiramente é construído o fundamento da casa, que servirá como base, após são erguidas as paredes, o telhado, a instalação das tubulações hidráulicas e elétricas, então se reboca as paredes. Esse conjunto de tarefas geralmente é executado sempre na mesma ordem, pois, como no exemplo, não podemos rebocar uma parede sem antes colocar a tubulação da rede hidráulica e elétrica. Esse conjunto de tarefas ordenadas é considerado como um processo.

Pfleeger (2004) define processo como "uma série de etapas que envolvem atividades, restrições e recursos para alcançar uma saída desejada".

Um processo de software pode ser definido como um conjunto estruturado de atividades e resultados associados que produzem um produto de software, que satisfaça uma série de objetivos e padrões. Sendo exigidas basicamente quatro atividades de processos fundamentais, comuns a todos os modelos de processos de software, que são: especificação de software, desenvolvimento de software, validação de software e evolução de software. (SOMMERVILLE,2007)

Para Tonsig (2008), a maioria das empresas que não utiliza qualquer método formal na gestão do projeto e documentação do produto, acaba entregando a seus clientes um software defeituoso, mal estruturado, que não atende completamente às necessidades dos usuários, além de, frequentemente, atrasarem as entregas.

Os processos de software buscam melhorar o desenvolvimento com a utilização da padronização, o que proporciona a redução da diversidade de modelos existentes que, muitas vezes, acabam sendo baseados em costumes e manias pessoais, melhorando, dessa forma, a organização dos processos de software.

Durante a evolução das metodologias de desenvolvimento, surgiram os modelos de processos de software que são o detalhamento de um determinado processo de software que apresenta uma visão dele, englobando as atividades dos processos e os papéis das pessoas envolvidas. Também estabelece um fluxo que define como os elementos do processo se inter-relacionam uns com os outros, pois todos os modelos de processos de software possuem atividades semelhantes, mas cada modelo aplica uma ênfase diferente às atividades e estabelece um fluxo de maneira diferente. (SOMMERVILLE, 2007) (PRESSMAN, 2010)

O processo de desenvolvimento de software é denominado, por muitos autores, como ciclo de vida do software, visto que ele descreve toda a "vida" do produto de software, desde a concepção até a implementação, entrega, utilização e manutenção. (PFLEEGER, 2004)

2.4 Arquitetura de software

No início da etapa da criação de um software, é necessário definir a arquitetura deste software, pois, como foi visto no subcapítulo 2.2, ela influencia diretamente na qualidade do mesmo, portanto precisamos estruturá-lo de maneira que ele possa evoluir e suportar mudanças.

> Sistemas grandes são sempre decompostos em subsistemas que fornecem algum conjunto de serviços relacionados. O processo inicial de projeto, que consiste em identificar esses subsistemas e estabelecer um *framework* para o controle e a comunicação de subsistemas, é denominado projeto de arquitetura. (SOMMERVILLE, 2007, p. 161)

A análise da arquitetura de um sistema é importante, porque, com ela, consegue-se ter uma visão do sistema como um todo, reduzindo o risco de esquecer algo essencial, além de conseguir priorizar o que realmente é relevante, ajudando a enquadrar o sistema aos objetivos do negócio. (LARMAN, 2008)

Quanto maior e mais complexo for o software, melhor deverá ser a definição de sua arquitetura, pois, consequentemente, ele terá um número maior de processos e funcionalidades, além de necessitar fornecer uma boa usabilidade, ser estável e ter um bom desempenho.

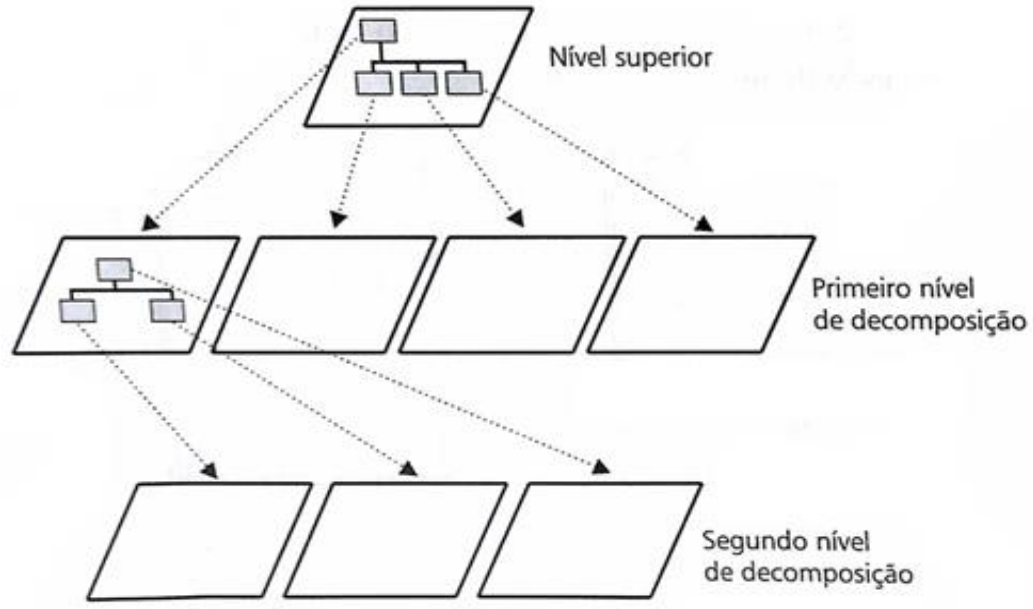
A definição da arquitetura deve ser realizada antes do início do desenvolvimento do sistema, porque ela ajuda a definir questões chave, que, caso não fossem identificadas na arquitetura, iriam gerar retrabalho no desenvolvimento.

Pfleeger (2004) afirma que “o projeto é o processo criativo de transformar o problema em uma solução; a descrição de uma solução também é chamada de projeto.”

Com a definição da arquitetura de um sistema, deve-se estabelecer a organização dele, definindo um modelo para seus módulos e componentes, de como será a separação das funcionalidades e a interação delas, de um modo que cada um tenha suas responsabilidades bem definidas. Dessa forma, a arquitetura traz clareza e organização ao sistema, pois estabelece uma base para a construção do sistema, o que auxilia na tomada de decisão e possibilita analisar os riscos e efetuar o planejamento e gerenciamento do projeto.

O projeto da arquitetura do sistema consiste em descrevê-lo inicialmente num nível mais alto, com seus principais elementos, que seriam os módulos e, em seguida, deve-se refinar e detalhar seus recursos e suas funcionalidades, definindo como interagem entre si, efetuando a decomposição do sistema em subsistemas. A Figura 3 descreve essa etapa.

Figura 3 - Níveis de Decomposição



Fonte: Pfleeger (2004 p. 162).

Pfleeger (2004) afirma que a modularidade é uma característica de um bom projeto, pois, dessa forma, o sistema está dividido em componentes ou módulos, sendo que cada componente tem seu propósito bem especificado, tendo definido suas entradas e saídas. Os

módulos são organizados em uma hierarquia que resulta da decomposição, isso possibilita investigar o sistema, considerando um nível de cada vez.

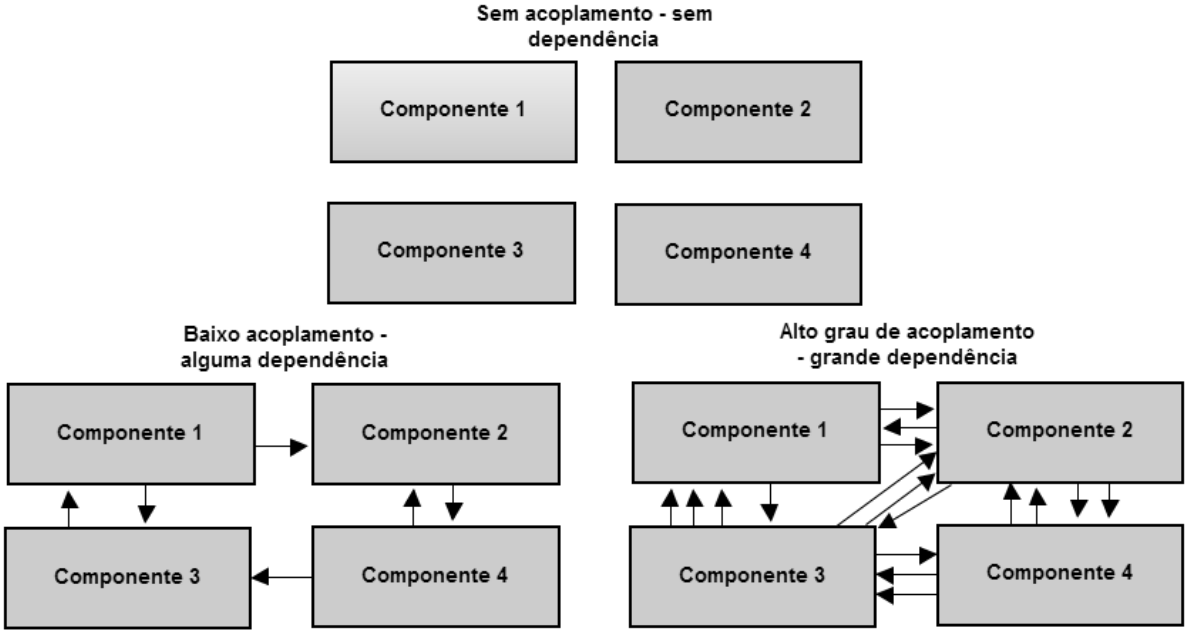
Durante a decomposição, a cada nível de um módulo busca-se aperfeiçoar e detalhar melhor os componentes de seu nível acima. Dessa forma, consideramos o nível superior como sendo o mais abstrato, pois ele oculta informações dos detalhes funcionais e de processamento. A vantagem da abstração e de ocultar informações é que isso permite que um componente seja alterado, sem que isso interfira em outros componentes, mantendo o projeto como um todo intacto.

Segundo Pfleeger (2004), um bom projeto deve buscar aplicar alguns princípios da Engenharia de Software, como acoplamento e coesão, pois cada um desses dois princípios visa a melhorar a arquitetura do software.

O acoplamento está relacionado com o quanto um componente depende de outro para funcionar, de modo que quanto maior a dependência entre os componentes, diz-se que eles estão fortemente acoplados, o que produz problemas, porque qualquer alteração em um componente impactará no funcionamento do outro, dificultando, assim, a manutenção e as modificações.

A Figura 4 demonstra os diferentes tipos de acoplamento entre componentes. Podemos ter um grupo de componentes que não possuam nenhum tipo de ligações e conexões com os outros, esses não possuem acoplamento. Outro caso é o inverso da situação anterior, onde um grupo de componentes possui diversas ligações e conexões entre si, o que representa um alto grau de acoplamento. Já, o terceiro exemplo da figura representa um grupo de componentes com poucas ligações, representando o baixo acoplamento.

Figura 4 - Representação dos tipos de acoplamentos



Fonte: Adaptado de Pfleeger (2004, p. 180).

Quanto maior o acoplamento entre os componentes, mais dependentes eles se tornam, portanto deve-se buscar uma solução que objetive o baixo acoplamento, para diminuir a dependência entre eles, facilitando a manutenção e a evolução dos componentes.

Já, a coesão está relacionada com o princípio da responsabilidade única. Esta define que um componente deve ter apenas um único objetivo, evitando assumir uma função que não é sua, o que, de certa forma, simplifica e facilita o entendimento.

Para Pressman (2010), outro ponto importante, dentro da arquitetura de um sistema, é a técnica da refatoração. Essa técnica consiste em reorganizar o código do sistema de forma que não se altere o comportamento do mesmo, apenas busca-se melhorar sua estrutura interna, simplificando-a.

A refatoração de um sistema ou componente consiste em reestruturar o código sem alterar seu comportamento, permitindo identificar e remover redundâncias e códigos ineficientes ou não utilizados, efetuar a correção de falhas identificadas, buscando simplificar e acrescentar flexibilidade ao código.

Para Sommerville (2007), a análise da arquitetura de um sistema também se preocupa com questões ligadas à resolução e identificação dos requisitos não funcionais. Basicamente são encontrados três tipos diferentes de requisitos não funcionais:

- Restrições técnicas: dão um limite ao projeto e geralmente tratam questões ligadas à definição da tecnologia utilizada, como as linguagens de programação, banco de dados, sistemas operacionais;

- Restrições do Negócio: também impõem limites ao projeto, mas por questões impostas pelo negócio e não pela tecnologia, como a utilização de ferramentas de código aberto, a necessidade do projeto de ter interfaces com determinadas ferramentas, motivos relacionados a potenciais clientes;

- Restrições de Qualidade: desempenho - operações mais importantes são restringidas a um pequeno número de subsistemas, para minimizar a comunicação entre estes; proteção - uma arquitetura em camadas com os itens mais importantes nas camadas mais internas; segurança - isolamento de componentes críticos para segurança; disponibilidade - inclusão de componentes redundantes na arquitetura; facilidade de manutenção - utilizar componentes com baixo acoplamento e maior coesão.

Pfleeger (2004) ainda afirma que não existe uma única solução para a construção de um software. Podemos criar diversas soluções para a arquitetura de um software, sendo que todas atendam plenamente aos requisitos, portanto cada equipe de desenvolvimento opta por uma arquitetura de software que melhor se encaixe em suas necessidades.

Para Sommerville (2007), a utilização do projeto e documentação da arquitetura do software promove algumas vantagens, como:

- Comunicação de *stakeholders*[1]: pois gera uma apresentação do sistema em alto nível, o que facilita uma visão do sistema como um todo de forma simples;

- Análise de sistema: força a identificar principais aspectos no início do projeto, influenciando no atendimento de requisitos não funcionais como: desempenho, confiabilidade, facilidade de manutenção;

- Promove reutilização: a arquitetura é uma descrição compacta do sistema, de como ele está organizado e como seus componentes operam entre si. Em caso

[1]***Stakeholders***é o termo como são chamados os interessados no sucesso do projeto na Engenharia de Software.

> de sistemas similares, a arquitetura ajuda a identificar componentes que possam ser reutilizados.

Para auxiliar na elaboração e documentação da arquitetura, geralmente são utilizados alguns diagramas da UML, que é descrita no subcapítulo 2.6. Ela permite representar a arquitetura com modelos gráficos. A UML dispõe de diversos diagramas que são independentes de linguagem, plataforma ou processo, que combinam conceitos de várias metodologias, que abrangem todas as etapas da construção de um sistema, como: modelagem do negócio, requisitos, análise, desenho, implementação, testes e implantação.

Um projeto que não possui uma arquitetura bem definida e documentada, geralmente enfrenta dificuldades na hora de sua manutenção ou incorporação de mudanças, pois é perdido muito tempo para entender o que faz o que e onde dentro do sistema, além de que qualquer alteração pode causar algum efeito colateral, causando frequentemente problemas e até mesmo dificultando a correção de um erro.

2.5 Reutilização de software

Grande parte dos sistemas construídos por uma empresa de desenvolvimento de software é semelhante, possuindo características ou propósitos iguais, dessa forma, ao construir um novo sistema, ao invés de fazê-lo do zero, podemos considerar sistemas anteriores, avaliando se seus componentes podem ser adaptados ou reutilizados por completo neste novo sistema.

Segundo Oliveira (2001), a adoção de técnicas que permitam a reutilização de código, resulta em aumento da produtividade e da qualidade, pois: “sistemas bem construídos têm mais partes reutilizáveis. E quando o sistema é bem especificado e projetado, gasta-se menos tempo em testes e “emendas” para atender ao usuário.”

Pfleeger (2004) afirma que existem basicamente duas formas de se proceder com a reutilização, que são: a reutilização caixa-preta e a reutilização caixa-branca. A reutilização caixa-preta consiste na reutilização de componentes inteiros, sem modificá-los, por exemplo, uma biblioteca de funções matemáticas. Já a reutilização caixa-branca consiste em adaptar um componente de acordo com as necessidades, para que ele se encaixe ao software como, por

exemplo, a utilização de um *framework*, que é modificado e estendido de acordo com as necessidades do software.

Desenvolver softwares não é uma tarefa fácil, pois eles atuam em problemas complexos, devem ser robustos e o mercado exige que eles sejam construídos e alterados num curto prazo. Para atender a essas necessidades, empresas de desenvolvimento de software devem adotar práticas que estimulam o aumento da produtividade de seus softwares. Uma forma de fazer isso é desenvolver seus sistemas de uma forma que se possa reutilizar parte desse software em outros que possuam requisitos semelhantes.

A reutilização de software, ou partes dele, além de garantir o aumento da produtividade, promove, também, o aumento da qualidade, pois os processos que foram reutilizados, geralmente, já foram testados e validados em outros sistemas, garantindo, assim, o seu funcionamento.

Para viabilizar a reutilização de software, devem-se desenvolver sistemas, aplicando algumas das práticas vistas anteriormente na definição da arquitetura, que são criar componentes com baixo acoplamento, alta coesão e o princípio da responsabilidade única, tornando-os mais robustos.

A tentativa de criar e reutilizar código e componentes em sistemas permite que softwares sejam criados com um mínimo de desenvolvimento, pois, com a reutilização, consegue-se aumentar a produtividade, reduzindo custos tanto do desenvolvimento quanto da manutenção. A reutilização diminui o volume de codificação, simplifica a compreensão e aumenta a qualidade dos sistemas.

2.6 UML

A Linguagem Modelada Unificada (UML - *Unified Modeling Language*) é uma linguagem visual utilizada para modelar e especificar a construção de um software, como também para documentar os artefatos do sistema.

UML é definida como:

> Uma família de notações gráficas, apoiada por um metamodelo único, que ajuda na descrição e no projeto de sistemas de software, particularmente daqueles construídos utilizando o estilo orientado a objetos. (FOWLER, 2004, p. 25)

Um dos pontos chave da UML é o fato de ela ser utilizada como uma padronização para representar graficamente vários aspectos de um sistema, permitindo representar tanto as partes relativas à implementação, através de seus diagramas de classes e de sequência, como a arquitetura do sistema e modelos de implantação com os diagramas de componentes, pacotes e implantação.

Os diagramas da UML têm como objetivo apresentar um sistema sob diferentes óticas, cujo conjunto destes diagramas compõe um único modelo. Com um modelo UML, consegue-se especificar o que um sistema deve fazer, porém não define como implementar esse sistema.

A UML é muito utilizada, pois permite documentar e ou especificar sistemas de uma maneira mais simples e genérica, pois seus diagramas não são específicos para uma determinada linguagem de programação. Por exemplo, um diagrama de classes que foi especificado em UML pode ser implementado em qualquer linguagem, como C++, Java ou C#.

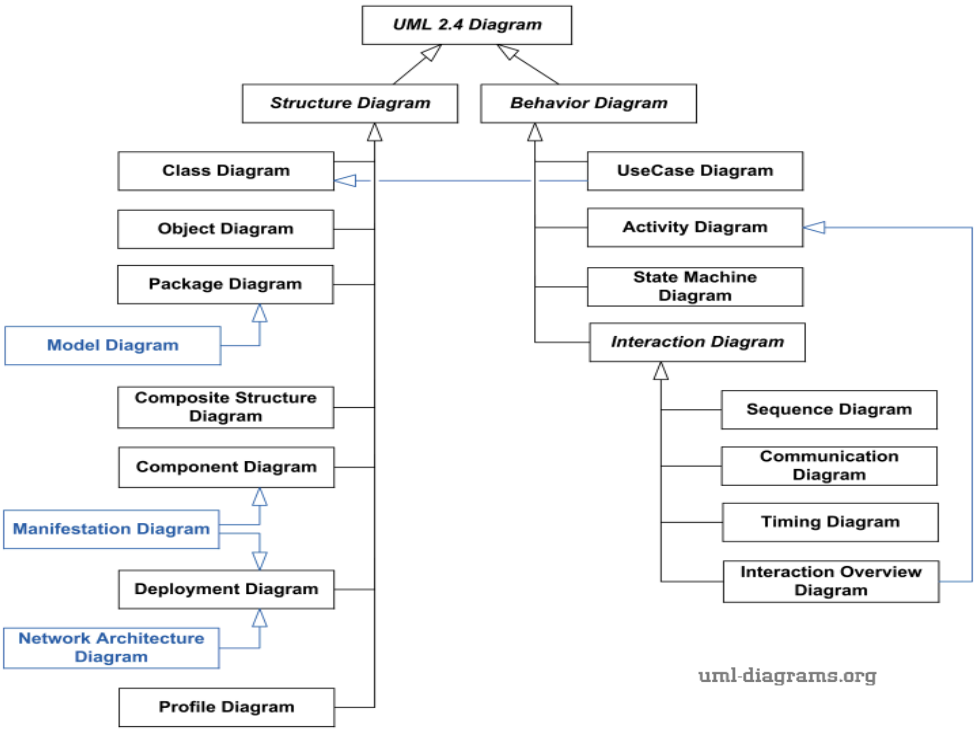
> Diagramas nos ajudam a ver ou explorar mais do panorama e relacionamentos entre os elementos de análise ou software, ao mesmo tempo em que nos permitem ignorar ou ocultar detalhes desinteressantes. (LARMAN, 2008, p. 42)

Outro aspecto importante da UML é o fato de poder visualizar e identificar facilmente as soluções elaboradas em outros sistemas que possam ser reutilizadas, além de servir como um modelo padrão, auxiliando na comunicação entre pessoas e equipes envolvidas em um projeto.

Conforme Guedes (2009), a UML teve sua origem na união de outros três métodos de modelagem: *Booch*, *Object Modeling Tecnique* (OMT) e o *Object-Oriented* Software *Engineering* (OOSE). Até meados da década de 1990, esses três métodos eram os mais utilizados pelos profissionais de desenvolvimento de software. Em 1996, foi lançada a primeira versão da UML que foi evoluindo com aprimorações para melhorar e ampliar a linguagem, sendo adotada, em 1997, como linguagem padrão para modelagem pelo *Object Management Group* (OMG). Em julho de 2005, foi lançada oficialmente a versão 2.2, sendo que, atualmente, encontra-se na versão 2.4.1.

A Figura 5 apresenta os diferentes tipos de diagramas definidos pela UML:

Figura 5 - Diagramas da UML



Fonte: Retirado da página *UML-Diagrams*[2].

Dentre os diversos diagramas da UML apresentados acima, apenas alguns destes serão detalhados a seguir, os quais serão utilizados no presente trabalho.

2.6.1 Diagrama de Casos de Uso

Larman (2008) define casos de uso como uma narrativa em texto, amplamente utilizada para descobrir e registrar requisitos. Já Guedes (2009), aborda que o diagrama de casos de uso é um diagrama mais geral e informal, que é utilizado durante todo o processo de modelagem do software, sendo elaborado nas fases de análise e levantamento de requisitos.

O digrama de casos de uso tem o objetivo de identificar os atores e as funcionalidades de um sistema, assim como a relação de quais atores podem executar quais funcionalidades. Os atores são a representação de usuários, outros sistemas, algum hardware ou serviço que interaja com alguma funcionalidade do software. (GUEDES, 2009)

A Figura 6 apresenta um exemplo de diagrama de casos de uso. Nele, pode-se verificar que um determinado caso de uso possui dois atores, o vendedor – que pode acessar as

[2] Disponível em: <http://www.uml-diagrams.org/uml-24-diagrams.html>. Acesso em maio 2013.

funcionalidades de consulta de produto e consulta de estoque, e o ator cliente que apenas pode acessara funcionalidade de consulta de produtos.

Figura 6 - Exemplo de diagrama de Casos de Uso

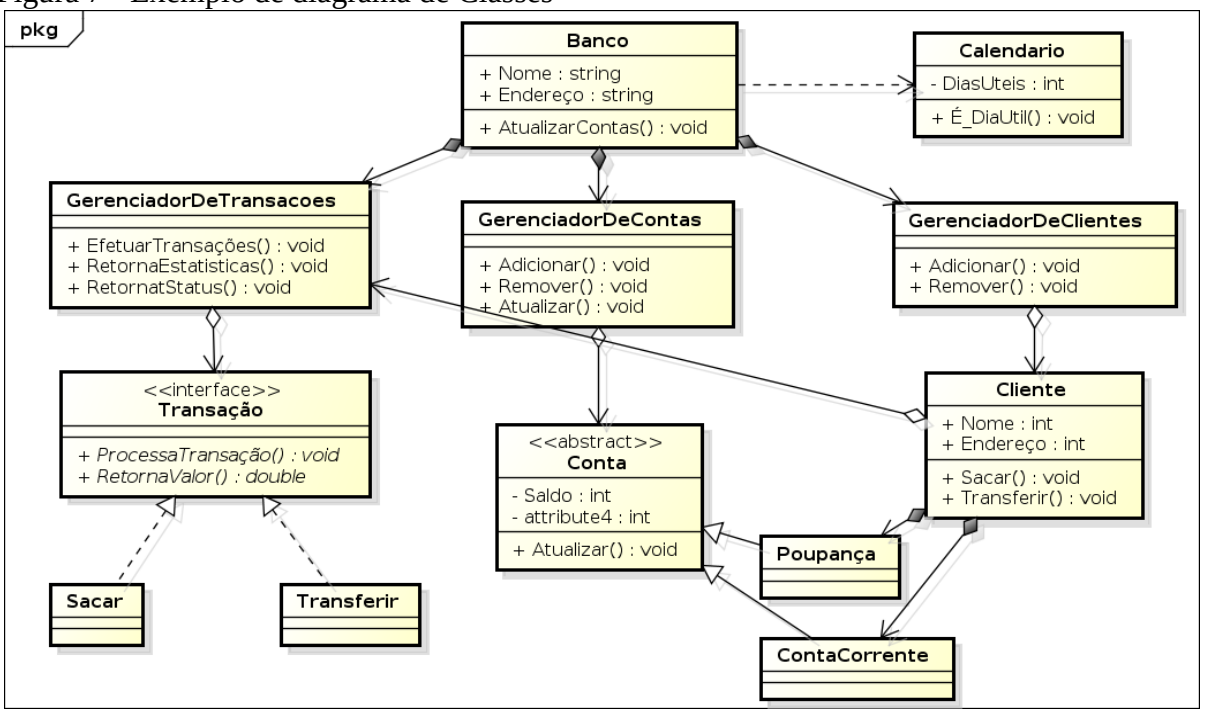
Fonte: Do autor.

2.6.2 Diagrama de Classes

Segundo Guedes (2009), o diagrama de classe é provavelmente o mais utilizado e é um dos mais importantes. Ele serve de apoio para os demais diagramas, tem, por princípio, definir a estrutura de classes utilizadas nos sistemas, além de estabelecer como estas se relacionam e trocam informações entre si.

A Figura 7 apresenta um exemplo de diagrama de classes.

Figura 7 - Exemplo de diagrama de Classes



Fonte: Do autor.

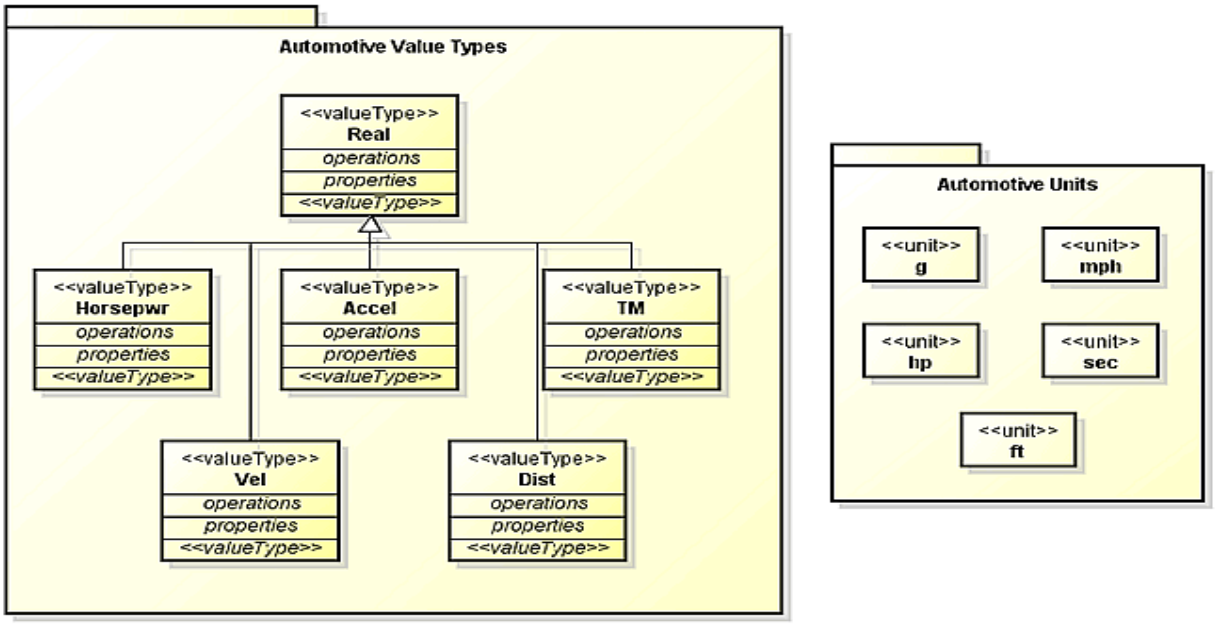
2.6.3 Diagrama de Pacotes

O diagrama de pacotes tem o objetivo de representar a arquitetura de um sistema, apresentando os módulos e subsistemas englobados no sistema, com as partes que o compõem e suas ligações. (GUEDES, 2009)

Fowler (2004) define o diagrama de pacotes como uma construção que agrupa elementos em unidades de um nível mais alto, sendo que um elemento pode ser qualquer construção UML. A aplicação mais comum desse diagrama é com o agrupamento de classes.

Na Figura 8, pode-se ver a estrutura de um diagrama de pacotes.

Figura 8 - Exemplo de um diagrama de Pacotes



Fonte: Retirado da página *Astah SysML Tutorial*[3].

2.6.4 Diagrama de Componentes

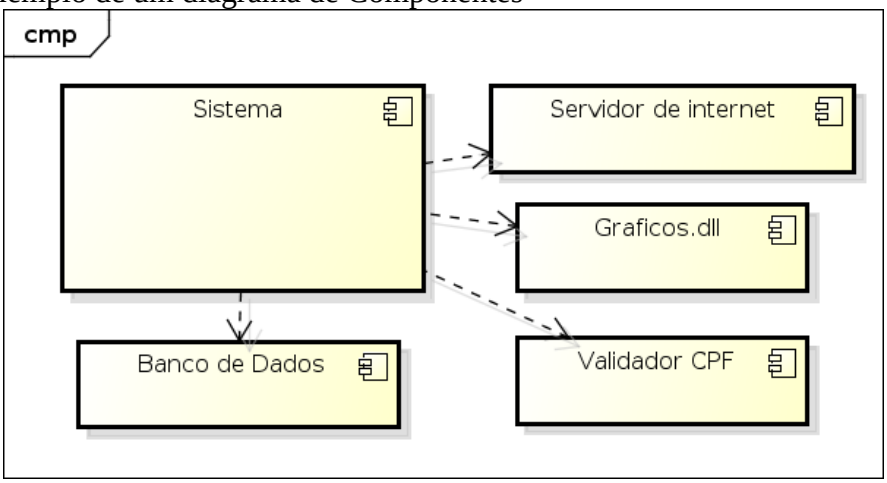
O diagrama de componentes busca apresentar os componentes do sistema que serão implementados, representando como estarão estruturados e como irão interagir os módulos de código fonte, bibliotecas, *Dynamic-link library* (DLL - Biblioteca de Vínculo Dinâmico), formulários e módulos executáveis para que o sistema funcione. (GUEDES, 2009)

[3] Disponível em: <http://astah.net/tutorials/sysml/block-definition>. Acesso em maio 2013.

Fowler (2004) afirma que o diagrama de componentes pode ser utilizado para apresentar a decomposição dos módulos de um sistema numa estrutura de mais baixo nível. Sendo que esses módulos ou componentes podem ser acrescidos e atualizados independentemente ao sistema.

A Figura 9 apresenta um diagrama de Componentes.

Figura 9 - Exemplo de um diagrama de Componentes



Fonte: Do autor.

2.7 Padrões de Projeto

No Capítulo 2, verificou-se que um bom software apresenta algumas características, como possuir componentes com baixo grau de acoplamento, alta coesão, implementados com abstrações, sendo bem organizados e estruturados, a fim de possibilitar um fácil entendimento, além de permitir atribuir mudanças para a evolução do software.

Uma maneira de garantir e atribuir as características descritas acima a um software é com a aplicação e a utilização de *Design Patterns*, ou Padrões de Projetos, ou seja, usando formas e modelos para padronizar a estrutura e a codificação do software, pois isso visa a garantir a aplicação de boas práticas da Engenharia de Software, que estão incorporadas nos padrões de projetos.

A definição de padrão de projeto segundo Gamma *et al*. (2008) é:

> Um padrão de projeto sistematicamente nomeia, motiva e explica um design geral que endereça um problema recorrente de design em sistemas orientados a objeto. Ele

> descreve o problema, a solução, quando aplicar a solução e suas consequências. Ele também dá dicas de implementação e exemplos. A solução é um arranjo geral de objetos e classes que resolvem o problema. A solução é customizada e implementada para resolver o problema em um contexto em particular. (GAMMA *et al*, 2008, p. 333)

Sommerville (2007) afirma que a utilização de padrões de projetos possibilita construir softwares e componentes que viabilizam a reutilização de forma eficiente, pois padrões são uma descrição do problema com a essência da sua solução, o que possibilita a reutilização dessa solução em diferentes sistemas, pois eles descrevem a experiência e o conhecimento que foram acumulados em uma solução comprovada para um problema em comum.

Os padrões de projeto tiveram sua origem com base nos padrões identificados pelo arquiteto e matemático Christopher Alexander, que teve a ideia de documentar os pontos em comum identificados em projetos de arquitetura e urbanismo, que seriam utilizados posteriormente em novos projetos.

Alexander é considerado o criador dos primeiros conceitos de padrões de projeto, o mesmo afirma que “cada padrão descreve um problema no nosso ambiente e o cerne de sua solução, de tal forma que se possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira”. (ALEXANDER, 1977 apud GAMMA et al., 2008)

Com base nos padrões de projetos arquitetônicos de Alexander, os autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides propuseram, no início da década 1990, o conceito de padrões de projetos para a área de software com a publicação do livro *Design Patterns: Elements of Reusable Object-Oriented* Software. Devido ao amplo reconhecimento obtido com a publicação desse livro, esses autores passaram a ser reconhecidos como a “Gangue dos Quatro” (GoF), sendo que esse livro é, até hoje, a principal referência sobre o assunto.

A obra publicada pela Gangue dos Quatro teve como objetivo, além de introduzir o conceito de padrões de projetos, descrever uma estrutura para catalogar um total de 23 padrões. Um ponto importante é que os autores não são responsáveis pela criação desses padrões, eles apenas documentaram padrões que foram identificados por eles.

Gamma et al. (2008) estabeleceu um formato para descrever e documentar um padrão de projeto, no qual deve ser descrito, em um esquema, um conjunto de elementos essenciais:

- O nome: Um padrão deve ter um nome a fim de identificá-lo, o que facilita nas discussões e trocas entre programadores;

- O problema: é a motivação do padrão, deve ter uma descrição da situação e do contexto no qual o padrão é aplicado;

- A solução: é a estrutura do padrão, envolve um número de classes e descreve as relações entre as suas instâncias, responsabilidades e colaborações. Define como resolver o problema;

- As consequências: resultados da análise das vantagens e desvantagens da aplicação do padrão, assim como as determinadas circunstâncias em que pode ser aplicado.

Fowler (2003) afirma que "Um padrão é uma ideia útil em um contexto prático e que provavelmente será útil em outro". Afinal, um padrão não é um algoritmo ou a implementação de um código que resolve um determinado problema, mas sim uma descrição abstrata de uma solução para resolver um determinado problema, no qual foi identificado o que se repete em diversas situações, sendo que essa solução pode ser utilizada várias vezes e de maneiras diferentes.

Os padrões de projeto devem ser vistos como uma forma de trazer clareza para um software, pois, com eles, consegue-se ter uma ideia de o que o software está fazendo e não apenas uma descrição dos detalhes do código. Outro ponto importante é que a aplicação deles permite melhorar o software, porque promovem a extensibilidade e a reusabilidade, além de serem altamente estruturados, estando documentados em um modelo que identifica o que é necessário para entender o problema e a sua solução. (SHALLOWAY e TROTT, 2004)

De acordo com Shalloway e Trott (2004) os padrões de projeto têm como princípio a reutilização de ideias que foram bem sucedidas, reaproveitando não apenas o código, mas também a experiência adquirida, resultando numa solução confiável, mais robusta, reduzindo a complexidade e simplificando a manutenção.

Os padrões também auxiliam a estabelecer um vocabulário em comum entre os desenvolvedores, melhorando a comunicação, a documentação e o aprendizado deles, além de servir como um referencial que pode ser seguido. E, ainda, fornece uma perspectiva de mais

alto nível do problema e do processo, sendo que aplicam os conceitos básicos da orientação a objetos como encapsulamento, herança e polimorfismo, o que melhora a capacidade de modificações no código fonte, tornando-o mais elegante, estruturado e consequentemente mais simplificado.

Pressman (2010) afirma que a utilização e aplicação de padrões de projeto, auxiliam na obtenção da qualidade de um software, contribuindo não só para a redução dos custos de manutenção envolvidos, como também na diminuição do tempo de aprendizado de novas bibliotecas ou funções pela equipe, com soluções mais simples e mais flexíveis, sem grandes complexidades, numa linguagem de alto nível.

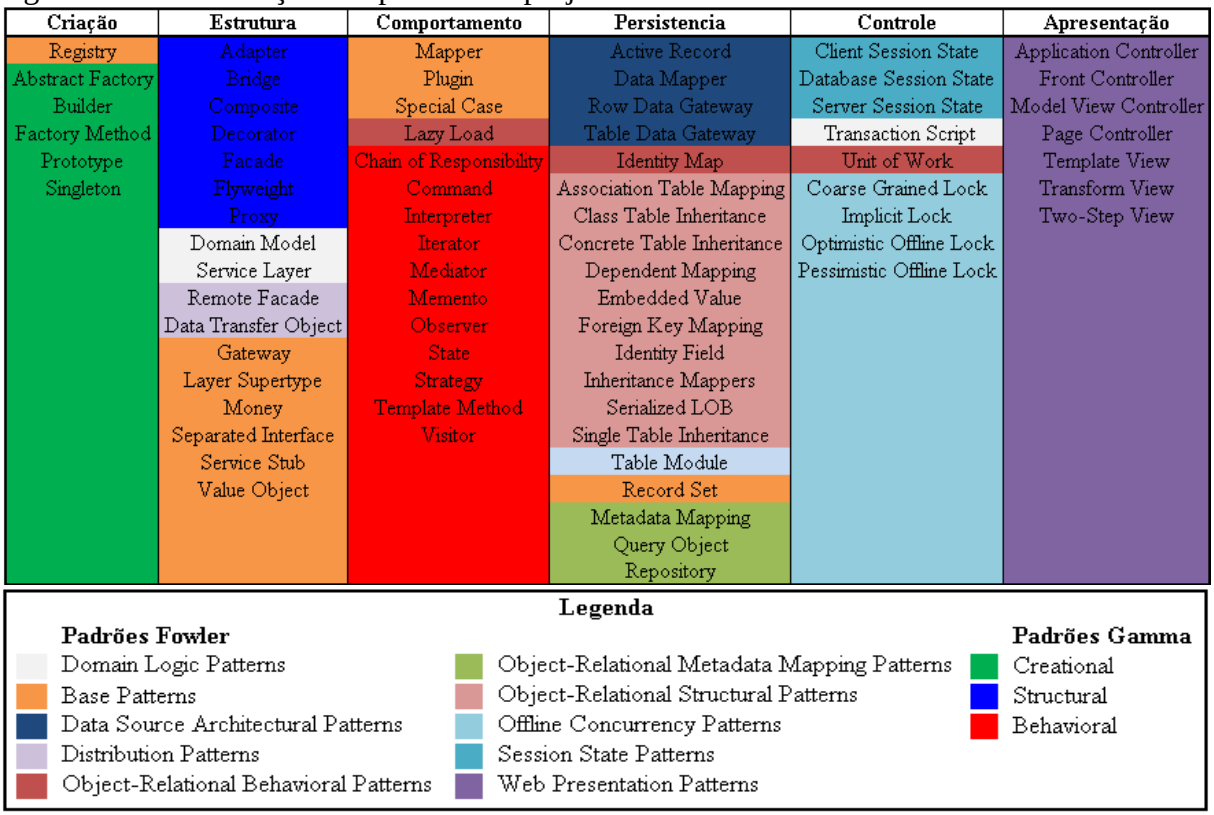
A fim de melhorar o entendimento dos padrões de projeto, os autores buscam classificá-los e dividi-los em famílias de acordo com a relação e características que cada padrão possui, numa forma de torná-los mais organizados.

Dentre os diversos autores que estudaram os padrões de projetos, o presente trabalho fará referência aos padrões de projeto documentados e catalogados por Gamma *et al.* (2008), que catalogou um total de 23 padrões no livro intitulado *Design Patterns: Elements of Reusable Object-Oriented Software*, em conjunto com os 51 padrões catalogados por Martin Fowler (2003), no livro intitulado *Patterns of enterprise application architecture*.

Na Figura 10, podemos observar os padrões de Gamma e de Fowler, classificados em seis grupos distintos, sendo: padrões de criação, padrões estruturais, padrões comportamentais, padrões de persistência, padrões de controle e padrões de apresentação. A partir da legenda abaixo da figura, podemos ver a classificação que cada autor utilizou em seu livro.

Após a Figura 10, que apresenta os padrões de Gamma e Fowler, será detalhada e exemplificada cada uma das seis classificações abordadas.

Figura 10 - Classificação dos padrões de projeto



Fonte: Do auto

2.7.1 Padrões de criação

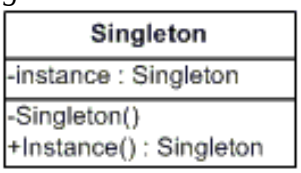
Os padrões de projeto deste grupo estão relacionados aos mecanismos que lidam com a criação e instanciação de objetos e classes.

Esses padrões podem ser subdivididos em padrões de criação de classes e padrões de criação de objetos. Os padrões de criação de classes sugerem utilizar o mecanismo de herança da orientação a objetos para padronizar e controlar instanciação de classes para subclasses, já os padrões de criação de objetos sugerem que a criação dos objetos deve ser delegada a outros objetos. Em suma, esses padrões sugerem um modelo que facilite e padronize a forma como é efetuada a criação e a instanciação dos objetos e classes. (GAMMA et al., 2008)

Como exemplo de padrão de criação, podemos citar o *Singleton*, que tem o objetivo de garantir que somente exista uma instância de uma determinada classe, fornecendo um método global para acessar esta única instância. (GAMMA et al., 2008)

A Figura 11 demonstra o diagrama do padrão *Singleton*:

Figura 11 - Diagrama do padrão *Singleton*



Fonte: Gamma et al. (2008, p. 130).

Para um melhor entendimento do funcionamento desse padrão, segue um exemplo de código da implementação e utilização dele, sendo detalhado na Listagem 1:

Listagem 1 - Exemplo da implementação do padrão *Singleton*

```
public class Conexao
{
    private static Conexao _instance;

    public static Conexao RetornaInstancia()
    {
        if(_instance ==null)
            _instance =new Conexao();
        return _instance;
    }
}
//Exemplo de uso
Conexao conn = Conexao.RetornaInstancia();
```

Fonte: Do autor.

Como podemos observar no exemplo acima, o método `RetornaInstancia()` tem como objetivo garantir que sempre seja retornada a mesma instância da classe, pois ele certifica-se de que a classe tenha sido instanciada apenas uma vez, retornando sempre uma referência para a única instância. Pelo fato de esse método ser público e estático, garante que seu acesso seja global em toda a aplicação, de uma forma simples e prática.

Esse padrão, geralmente, é utilizado em classes que retornam a conexão ao banco de dados, ou nas em classes que controlam a infraestrutura de *logs* de uma aplicação, pois são casos em que precisamos ter apenas uma única instância desses objetos dentro de uma mesma execução da aplicação.

2.7.2 Padrões estruturais

Todos os padrões de projeto relacionados a esse grupo sugerem como devem ser estruturados e modelados os objetos e as classes. Esses padrões também sugerem a criação de

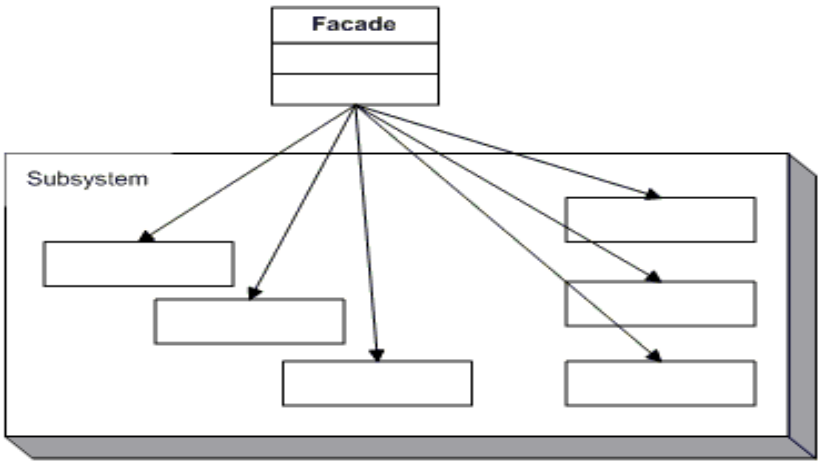
interfaces que serão herdadas pelos objetos, definindo, assim, um modelo de estrutura para a criação das funcionalidades nos objetos.

Esses padrões nos levam a identificar quais são as relações entre as entidades de uma maneira mais simples, facilitando e simplificando o entendimento da aplicação, num código mais limpo e claro.

Um exemplo de padrão estrutural é o *Facade,* que tem como objetivo fornecer uma interface unificada para um conjunto de interfaces de um subsistema. Com ele podemos definir uma interface de um nível mais elevado para um conjunto de subsistemas, facilitando, assim, sua utilização. (GAMMA et al., 2008)

A Figura 12 apresenta o diagrama do padrão *Facade.*

Figura 12 - Diagrama do padrão *Facade*



Fonte: Gamma et al. (2008, p. 181).

A Listagem 2, apresenta um exemplo da implementação e utilização do padrão *Facade*.

Listagem 2 - Exemplo da implementação do padrão *Facade*

```
public class Facade
{
    //Relação de SubSistemas
    SubSistemaUm obj1;
    SubSistemaDois obj2;
    SubSistemaTres obj3;
    SubSistemaQuatro obj4;
    public Facade()
    {
        //Instancia os SubSistemas
        obj1 =new SubSistemaUm();
        obj2 =new SubSistemaDois();
        obj3 =new SubSistemaTres();
        obj4 =new SubSistemaQuatro();
    }
```

```
        public void MetodoA()
        {
            Console.WriteLine("MetodoA -----");
            obj1.MetodoUm();
            obj2.MetodoDois();
            obj3.MetodoTres();
            obj4.MetodoQuatro();
        }
        public void MetodoB()
        {
            Console.WriteLine("MetodoB -----");
            obj2.MetodoDois();
            obj3.MetodoTres();
        }
    }

    //Exemplo de uso
    Facade facade =new Facade();
    facade.MetodoA();
    facade.MetodoB();
```

Fonte: Do autor.

Na Listagem 2, podemos observar a utilização do padrão, no qual foi criada uma classe chamada `Facade`, que tem, como objetivo, fornecer uma interface que simplifica o uso de um conjunto de subsistemas que, no exemplo, são o `SubSistemaUm`, `SubSistemaDois`, `SubSistemaTres` e `SubSistemaQuatro`. Como podemos verificar, o padrão *Facade* cria uma camada de fachada que busca simplificar, agregar e facilitar o uso de um conjunto de subsistemas.

Com a utilização desse padrão, podemos reduzir o acoplamento de um sistema a seus subsistemas, pois, assim, as alterações que possam ocorrer neste subsistema afetarão apenas o *Facade* e não toda a aplicação. Além disso, esse padrão ajuda a fornecer uma interface simples para um sistema mais complexo.

Esse padrão, geralmente, é utilizado para estruturar um sistema em camadas, no qual podemos aplicar o *Facade* para definir uma interface para cada camada e, dessa forma, podemos automatizar os testes de cada camada da aplicação, pois ela estaria isolada e desacoplada das demais camadas, por exemplo.

2.7.3 Padrões comportamentais

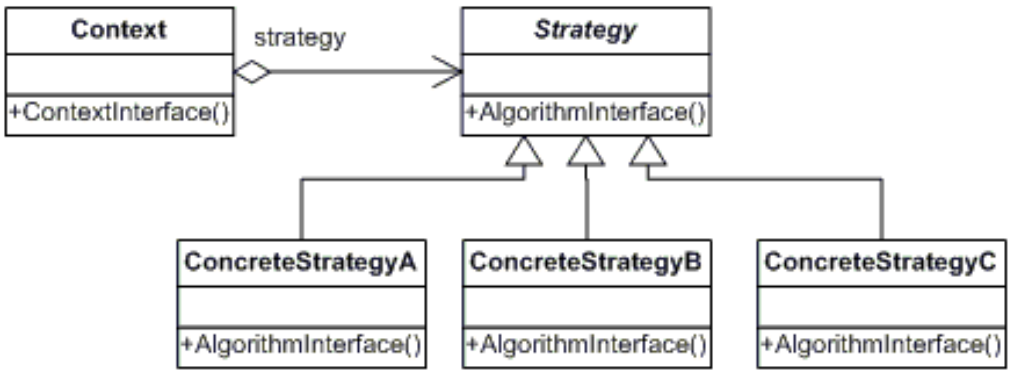
Os padrões de projeto deste grupo sugerem modelos específicos de como deve ocorrer a comunicação entre os objetos e as classes, além da atribuição das responsabilidades.

Esses padrões ajudam a identificar os modelos em comum na comunicação dos objetos e como esta comunicação ocorre (os fluxos de controle e as cooperações), tornando a comunicação muito mais flexível.

O padrão *Strategy* é um exemplo de padrão comportamental, pois ele permite que o algoritmo de uma classe varie, independentemente dos clientes que a utilizem. Ele define uma família de algoritmos, fazendo o encapsulamento de cada um, permitindo que eles sejam intercambiáveis.

A Figura 13 apresenta o digrama do padrão *Strategy*.

Figura 13 - Diagrama do padrão *Strategy*



Fonte: Gamma et al. (2008, p. 294).

Na Listagem 3, podemos ver um exemplo da utilização do padrão *Strategy*:

Listagem 3 - Exemplo da implementação do padrão *Strategy*

```
interface IStrategy
{
    void Converter();
}

public class ConversorBaixaQualidade : IStrategy
{
    public void Converter()
    {
        ConversorDeArquivo conversor =new ConversorDeArquivo("BAIXA");
        conversor.ConverteArquivo();
    }
}

public class ConversorAltaQualidade : IStrategy
{
    public void Converter()
    {
        ConversorDeArquivo conversor =new ConversorDeArquivo("ALTA");
        conversor.ConverteArquivo();
    }
}
```

```
public class ConversorDeArquivo
{
    string qualidade;
    public ConversorDeArquivo(string qualidade)
    {
        this.qualidade = qualidade;
    }
    public void ConverteArquivo()
    {
        if(qualidade =="BAIXA")
            Console.WriteLine("Convertendo arquivo na qualidade BAIXA");
        elseif(qualidade =="ALTA")
            Console.WriteLine("Convertendo arquivo na qualidade ALTA");
    }
}

//Exemplo de uso
IStrategy qualidadeSelecionada =null;

Console.WriteLine("Informe a qualidade: 1-Baixa, 2- Alta");
int op = Console.Read();
if(op ==1)
    qualidadeSelecionada = new ConversorBaixaQualidade();
elseif(op ==2)
    qualidadeSelecionada = new ConversorAltaQualidade();

if(qualidadeSelecionada != null)
    qualidadeSelecionada.Converter();
```

Fonte: Do autor.

Nesse exemplo, podemos ver que a classe `ConversorDeArquivo` permite converter um arquivo através do método `ConverteArquivo()`, porém o algoritmo desse método varia de acordo com a opção de qualidade informada.

Com esse padrão, podemos ganhar flexibilidade no software, definindo um conjunto de variações de um método, que são compatíveis entre si, sem criar uma dependência entre as classes envolvidas. Pode-se notar que, no exemplo, a conversão de arquivos não depende de um método de conversão específico, mas sim de um que implemente a interface `IStrategy`. Isso permite que as alterações nas classes que variam o comportamento do método não impactem no método em si.

Esse padrão pode ser aplicado em casos nos quais existem mais de uma "estratégia" para resolver um problema como, por exemplo: num *e-commerce*, aonde, ao efetuar uma compra, podemos selecionar mais de uma estratégia para calcular o frete como, por exemplo, Sedex, PAC, transportadora X. Assim, definiu-se apenas um interface `CalculaFrete`, que pode ter suas diferentes implementações de acordo com a opção selecionada.

2.7.4 Padrões para persistência

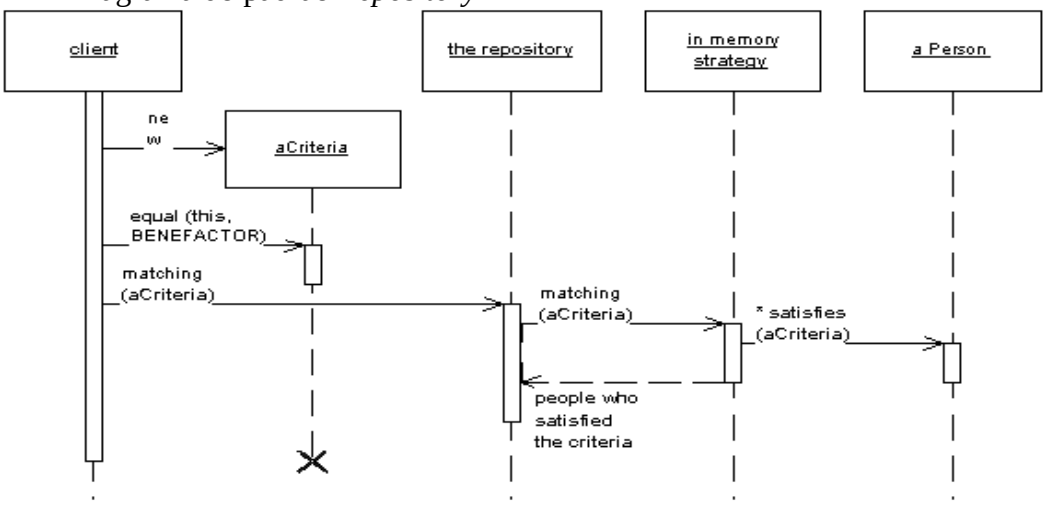
Os padrões de projeto para persistência de dados têm o objetivo de criar uma camada, que é responsável pela comunicação entre as demais camadas do software, com o banco de dados.

Esse grupo de padrões compreende a maneira pela qual a lógica do domínio da aplicação irá trocar informações com o banco de dados, disponibilizando uma interface que centralize tudo que for necessário para efetuar a persistência dos dados, simplificando a forma pela qual a aplicação "conversa" com o banco de dados.

Como exemplo de padrão de persistência, podemos citar o *Repository*, que tem o objetivo de isolar a lógica de acesso a dados de qualquer outra lógica do software, criando uma camada que é responsável pela mediação entre a camada de mapeamento de dados com a camada de domínio da aplicação. O *Repository* cria uma camada na qual os objetos persistidos são tratados num formato de coleção e busca concentrar o código de construção de consultas de acesso a dados. (FOWLER, 2003)

A Figura 14 apresenta o digrama do padrão *Repository*.

Figura 14 - Diagrama do padrão *Repository*



Fonte: Fowler (2003, p. 322).

Na Listagem 4, podemos ver um exemplo da implementação do padrão *Repository*.

Listagem 4 - Exemplo da implementação do padrão *Repository*

```
public interface IRepositorio<TEntity> where TEntity :class
{
    TEntity Salvar(TEntity item);
    void Excluir(TEntity item);
    TEntity RetornaPeloId(object key);
}
public class Repositorio<TEntity>: IRepositorio<TEntity> where TEntity :class
{
    public TEntity Salvar(TEntity item)
    {
        //código para salvar item...
    }
    public void Excluir(TEntity item)
    {
        //código para excluir item....
    }
    TEntity RetornaPeloId(object key)
    {
        //código para retornar o item do id...
    }
}

publicclass Cliente
{
    public int Id { get; set;}
    public string Nome { get; set;}
    public bool Inadimplente { get; set;}
}

public interface IClienteRepositorio : IRepositorio<Cliente>
{
    IEnumerable<Cliente> BuscaClientesInadimplentes();
}
public class ClienteRepositorio : IClienteRepositorio
{
    IEnumerable<Cliente> BuscaClientesInadimplentes()
    {
        //código para retornar clientes inadimplentes...
        return (from cli in db.Clientes
            where  cli.Inadimplente == False
            select cli).ToList();
    }
}

//Exemplo de uso
IClienteRepositorio repCli = new ClienteRepositorio();

Cliente cli = new Cliente();
cli.Nome ="João";
cli.Inadimplente = False;

//Salva o cliente no BD
repCli.Salvar(cli);

//retorna lista de clientes inadimplentes
IEnumerable<Cliente> listaCliInadimplentes =
repCli.BuscaClientesInadimplentes();
```

Fonte: Do autor.

No exemplo acima, podemos verificar a utilização do padrão *Repository*, que fornece uma interface para manipular a persistência de dados de uma maneira mais orientada a

objetos, em que cada tabela ou conjunto de tabelas é tratado como se fosse uma coleção de dados em memória, permitindo inserir, alterar e excluir dados.

Um aspecto importante desse padrão é que ele auxilia na separação da camada de mapeamento de dados da camada de domínio, diminuindo a dependência entre elas, permitindo desenvolver um sistema, abstraindo a forma como esse irá se comunicar com o banco de dados, podendo, assim, desenvolver a camada de mapeamento de dados separadamente, assim, informando apenas uma interface ao *Repository*, seria permitido trocar a forma como o *Repository* acessa os dados.

Fowler (2003) sugere a aplicação desse padrão a sistemas grandes e complexos que geralmente possuem muitos objetos de domínio e demandam muitas consultas, pois, com ele, podemos reduzir a quantidade de códigos de consulta, concentrando-os em um único lugar, minimizando a duplicidade de códigos delas, fazendo isso de uma forma encapsulada e orientada a objetos.

Outro ponto importante é que podemos especificar as consultas do sistema, de uma forma totalmente orientada a objetos, em que se cria objetos de critérios para filtrar os registros, sem a necessidade de escrever consultas SQL.

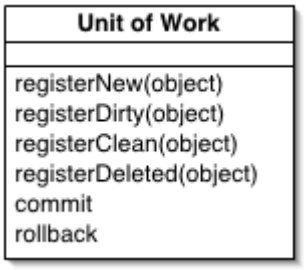
2.7.5 Padrões para controle

Este grupo de padrões de projeto tem como objetivo lidar com questões relacionadas com o controle da aplicação, ou seja, eles buscam gerenciar e controlar a forma pela qual o fluxo das informações deve ocorrer na aplicação.

Como exemplo de um padrão de projeto utilizado para o controle, podemos citar o padrão *Unit of Work* que tem como principal objetivo manter uma lista de objetos os quais foram afetados por uma transação comercial na aplicação, então o *Unit of Work* coordena a gravação desse conjunto de alterações, a fim de tratar a resolução de possíveis problemas de concorrência.

A seguir podemos ver o digrama do padrão:

Figura 15 - Diagrama do padrão *Unit of Work*



Fonte: Fowler (2003, p. 184).

Como podemos observar no diagrama acima, esse padrão busca encapsular as alterações que possam ocorrer com os objetos do domínio em uma transação, a fim de permitir que, caso ocorra algum problema durante um determinado processo desse sistema, seja possível restaurar o estado anterior desses objetos, garantido que o processo seja apenas finalizado caso ocorra por completo e, com a ocorrência de qualquer problema, seja restaurado o estado anterior do sistema.

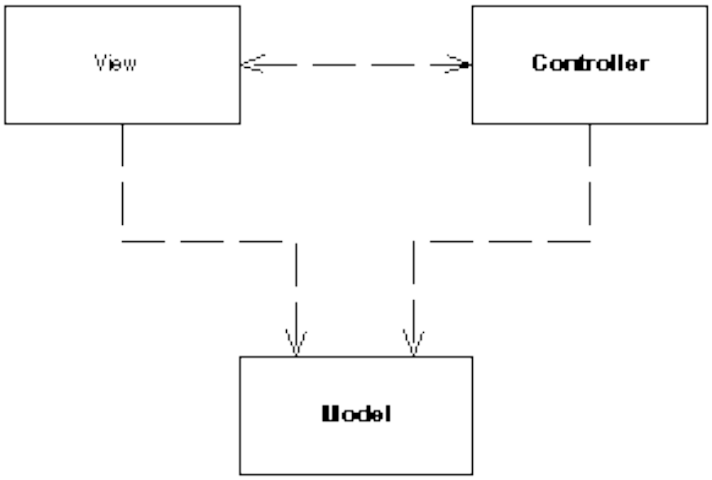
2.7.6 Padrões para apresentação

Este grupo de padrões de projeto tem o objetivo de definir a forma pela qual as informações serão apresentadas para o usuário, sendo utilizado para separar a camada de interface do restante da aplicação, permitindo que a aplicação fique independente da camada de apresentação.

Um ótimo exemplo de um padrão de apresentação é o padrão *Model-View-Controller* (MVC) que tem como objetivo isolar a interface de apresentação da regra de negócios da aplicação, dividindo esta interação em três papéis distintos: *Model*, *View* e *Controller*. (FOWLER, 2003)

A Figura 16 apresenta o diagrama do padrão *Model-View-Controller*.

Figura 16 - Diagrama do padrão *Model-View-Controller*



Fonte: Fowler (2003, p. 330).

O *Model* é responsável por representar a camada de domínio da aplicação, contendo os dados e o comportamento, mas não conhecendo detalhes da visualização desses dados. O *View* é a camada de interface que tem como objetivo apresentar as informações do modelo aos usuários, sendo que qualquer alteração nessas informações será direcionada ao *Controller*. Este, por sua vez, recebe as informações de entrada do usuário, faz as manipulações necessárias no modelo e, após isso, atualiza as informações na camada *View*.

Esse padrão permite que seja separada a camada de interface da camada do modelo de domínio da aplicação. Com isso podemos desenvolver a camada de interface, levando em conta apenas aspectos relacionados à interface com o usuário, assim como desenvolver o núcleo da aplicação, levando em conta apenas questões ligadas às políticas do negócio, reduzindo as dependências da aplicação, em que a interface dependa do modelo de domínio, porém este não dependa da interface.

O padrão MVC é largamente utilizado nas aplicações atuais, devido ao benefício de separar a camada de apresentação e interface do usuário, da camada do modelo de domínio, pois é bastante comum termos uma mesma aplicação que funcione tanto na WEB, como em dispositivos móveis, possuindo as mesmas funcionalidades em ambos os meios.

3 METODOLOGIA

A fim de possibilitar o desenvolvimento deste trabalho, foi necessário definir e seguir sua metodologia. Goldenberg (2000) afirma que "Metodologia Científica é muito mais do que algumas regras de como fazer uma pesquisa. Ela auxilia a refletir e propicia um 'novo' olhar sobre o mundo: um olhar científico, curioso, indagado e criativo". Ainda, segundo Goldenberg (2000), "a pesquisa científica exige criatividade, disciplina, organização e modéstia, baseando-se no confronto permanente entre possível e impossível, entre o conhecimento e a ignorância".

Segundo Wainer (2007), uma pesquisa em Ciência da Computação[4] envolve novidades, as quais geralmente tratam da criação de um programa, de um modelo, de um algoritmo ou de um sistema novo. Porém existem casos em que um sistema, um modelo ou um algoritmo novo não é suficiente para constituir uma pesquisa, então se faz necessário avaliá-lo de uma forma metodológica em busca de algum conhecimento.

O objetivo do presente trabalho consistiu em buscar soluções técnicas (bibliotecas, estruturas, classes, componentes) que trabalhem conjuntamente para a formação de uma arquitetura de software em camadas, de maneira coerente com os principais *Design Patterns* conhecidos. Para tanto, foi necessária uma pesquisa com o objetivo de explorar soluções técnicas. Dessa forma, este trabalho se caracteriza, conforme Santos (1999), segundo seu objetivo, como pesquisa exploratória de natureza qualitativa.

[4] Ciência da computação é a ciência que faz o estudo dos algoritmos, de suas aplicações e de suas implementações na forma de software.

> Explorar é tipicamente a primeira aproximação de um tema e visa criar maior familiaridade a um fato ou fenômeno. Quase sempre busca-se essa familiaridade pela prospecção de materiais que possam informar ao pesquisador a real importância do problema, o estágio em que se encontram as informações já disponíveis a respeito do assunto, e até mesmo, revelar ao pesquisador novas fontes de informação. (SANTOS, 1999, p. 26)

As soluções técnicas que foram exploradas, ou seja, pesquisadas para a obtenção da arquitetura de software foram baseadas em técnicas e modelos que foram consagrados em ideias bem sucedidas, obtidas com a experiência. Para tanto, foi preciso buscar embasamento para justificar quais técnicas e modelos foram bem sucedidos. Assim, a pesquisa bibliográfica deve ser utilizada para buscar informações importantes sobre o assunto.

> A Pesquisa bibliográfica é fundamentada nos conhecimentos de biblioteconomia, documentação e bibliografia; sua finalidade é colocar o pesquisador em contato com o que já se produziu a respeito do seu tema de pesquisa. (PÁDUA, 2004, p. 55)

O trabalho não teve o objetivo de efetuar uma pesquisa para quantificar soluções, de forma a gerar dados estatísticos que possam ser apresentados, mas sim de efetuar um estudo aprofundado de cada solução, buscando qualificar o uso de cada uma, descrevendo-as e documentando-as. Portanto, este estudo se trata de uma pesquisa qualitativa que, segundo Wainer (2007), é definida como:

> Métodos qualitativos diferem de métodos quantitativos, porque se ocupam de variáveis que não podem ser medidas, apenas observadas. Essa é uma dicotomia muito simplista. Métodos qualitativos vêm das ciências sociais, em oposição aos métodos quantitativos que derivam das ciências naturais. Essa diferença na origem já é suficiente para que visões diferentes sobre o que é ciência, e como se faz ciência, tornem definições sucintas sobre o que é um ou outro método muito difícil. De um modo geral, métodos qualitativos em Ciência da Computação são métodos que se caracterizam por ser um estudo aprofundado de um sistema no ambiente onde ele está sendo usado, ou, em alguns casos, onde se espera que o sistema seja usado. Métodos qualitativos sempre envolvem pessoas, e na maioria das vezes sistemas. (WAINER, 2007, p.27)

Dessa forma, como foram procuradas soluções que aplicam *Design Patterns* devido ao fato de que elas visam a aplicar as boas práticas da orientação a objetos, foi necessária a avaliação de cada uma das soluções que forem utilizadas.

Toda a especificação da arquitetura que foi construída para a solução proposta, foi documentada utilizando os diagramas da linguagem UML, pois ela possibilita documentar e especificar a estrutura de um software de forma simples, por se tratar de uma linguagem visual que nos permite ter uma visão do software como um todo, além de ser uma linguagem padrão para a modelagem de softwares.

Para efetuar essa avaliação, foi necessário utilizar o conceito de verificação e de validação. A verificação tem como objetivo avaliar se o que foi planejado realmente foi realizado, ou seja, se os requisitos, funcionalidades e performance documentados foram implementados. Já, a validação tem o objetivo de avaliar se o que foi entregue atende às expectativas. De uma forma ampla, avaliação é o processo de verificar para que serve e o quanto serve. (WAINER, 2007)

A validação da proposta sugerida se deu sobre um protótipo. Para Pfleeger (2004): “um protótipo para avaliação da viabilidade nos permite descobrir, no estágio de projeto, se a solução que propomos realmente resolverá o problema em questão".

O protótipo elaborado atua sobre um conjunto de requisitos não funcionais de software, levando em conta questões ligadas ao desempenho, qualidade, manutenibilidade, interface e não a itens relacionados a requisitos funcionais que tratam de questões ligadas às funcionalidades específicas do software, devido ao fato de que este trabalho tem o objetivo de propor uma arquitetura e não a construção de um determinado software.

Como instrumento para a coleta de informações para a avaliação de cada solução, foi adotado o procedimento de Estudo de Caso. Um Estudo de Caso consiste no estudo amplo e detalhado de um ou de poucos objetos que, neste trabalho, serão as soluções, descrevendo e explorando a situação do contexto em que é feita a investigação de cada solução.

O estudo de caso se realizou devido à necessidade da evolução dos softwares da empresa ABC Informática LTDA, que necessita acompanhar a evolução tecnológica que o mercado de trabalho demanda, para tanto, o mesmo servirá de base para estudar diversas alternativas e soluções existentes no mercado, buscando associar as que melhor se encaixam nas necessidades da empresa, de forma a garantir a qualidade dos produtos entregues.

Para Wainer (2007), a literatura sobre métodos qualitativos não é comum, porém é uma literatura muito rica. Wainer (2007) ainda afirma que:

> A área de Sistemas de Informação tem um caráter mais aplicado, e estuda desde como resolver problemas tecnológicos práticos no uso e desenvolvimento de sistemas de informação, até os impactos econômico/financeiros e sociais desses sistemas nas organizações, e até os problemas na adoção ou desenvolvimento de novos sistemas. (WAINER, 2007, p.28)

Portanto, o presente trabalho visou a explorar e qualificar soluções técnicas que, a partir da documentação e do estudo das mesmas, possibilitem disseminar e agregar

informações importantes que possam auxiliar outras pessoas na busca de soluções para uma arquitetura de software.

4 ESTUDO REALIZADO

Neste capítulo, estão contidas as informações referentes à apresentação da solução da arquitetura que será proposta para o sistema de gestão empresarial, com o objetivo estabelecido neste trabalho. A mesma é descrita e documentada no decorrer deste capítulo.

4.1 Contextualização

O propósito geral do estudo realizado no presente trabalho é analisar um conjunto de soluções técnicas, verificando a aderência dessas soluções a padrões de projeto, para elaborar um modelo de arquitetura para um sistema de gestão para a empresa ABC Informática LTDA.

4.1.1 A empresa

Primeiramente, é necessário relembrar que a empresa na qual foi desenvolvido o presente trabalho, solicitou que sua identidade fosse preservada, para tanto ela sempre foi referida pelo nome fictício de ABC Informática LTDA.

A ABC Informática é uma empresa que está no mercado há mais de 20 anos, atuando no mercado de Tecnologia da Informação (TI), tendo como foco do negócio o desenvolvimento de sistema de gestão empresarial, consultoria de processo na área de TI, desenvolvimento de sistemas personalizados para *desktops*, dispositivos móveis e WEB,

buscando proporcionar a seus clientes uma vantagem competitiva através da correta aplicação da tecnologia da informação.

A empresa situa-se no Vale do Taquari e possui uma vasta carteira de clientes, atendendo, principalmente, ao estado do Rio Grande do Sul, possuindo, também, clientes nos estados de Santa Catarina, Paraná, São Paulo, Rio de Janeiro e Espírito Santo.

A ABC Informática tem, como principal produto, um sistema completo de gestão empresarial, o qual está dividido em módulos que contemplam os seguintes departamentos:

- Faturamento;
- Financeiro;
- Estoque;
- Expedição;
- Suprimentos;
- Prestação de Serviços e Contratos;
- Transporte;
- *Customer Relationship Management* (CRM - Gestão de Relacionamento com o Cliente);
- Planejamento e Controle da Produção (PCP);
- *Business Intelligence.*

Outro ponto importante do sistema de gestão empresarial da ABC Informática é a conformidade com as legislações, como emissão de NF-e, NFS-e, CT-e, SPED, conformidade com o PAF-ECF, entre outros. Além disso, o sistema é amplamente personalizável, permitindo, assim, adaptar-se melhor às necessidades de seus clientes.

4.1.2 A equipe de desenvolvimento

A ABC Informática possui, atualmente, um equipe de desenvolvimento bem qualificada e diversificada, sendo que seus integrantes possuem seus papéis e responsabilidades bem definidas, ocorrendo casos em que um integrante assuma mais de um papel na equipe.

Na Tabela 1, podemos visualizar os diferentes papéis existentes atualmente na equipe de desenvolvimento da empresa.

Tabela 1 - Papéis da Equipe

Papel	Qtde	Responsabilidades
Analista	3	Levantamento e especificação de requisitos, reuniões com clientes, analise do negócio
Arquiteto de Software	1	Definir a arquitetura do software, definir as tecnologias, garantir e manter a evolução tecnológica dos produtos
Especificador Técnico	1	Definir e detalhar os requisitos aos desenvolvedores, conhecer modelagem de Banco de Dados, UML
Desenvolvedor C#	3	Conhecer a linguagem C#, Orientação a Objetos, *Frameworks*, SQL Server, MySQL, PostgreSQL
Desenvolvedor VB	4	Conhecer a linguagem Visual Basic, conhecimento básico de Banco de Dados, lógica de programação
Desenvolvedor WEB	2	Conhecer HTML, CSS, Ajax, JQuery
Gerente de Projetos	1	Gerenciar e coordenar projetos e equipe de desenvolvimento

Fonte: do autor.

A partir do levantamento dos papéis existentes na equipe de desenvolvimento, são listados os membros da equipe, junto da atribuição dos papéis por eles assumidos, conforme pode ser visto na Tabela 2:

Tabela 2 - Membros da Equipe

Quem	Papéis
Evandro	Gerente de Projetos
Douglas	Analista, Arquiteto de Software, Especificador Técnico, Desenvolvedor C# e VB
Leandro	Analista
Marcos	Analista
Ivan	Desenvolvedor C# e VB
Cléverton	Desenvolvedor C#, WEB
João	Desenvolvedor VB, WEB
Tailini	Desenvolvedor VB

Fonte: do autor.

Com isso, pode-se perceber que a equipe de desenvolvimento é bastante diversificada e, tratando-se de uma equipe pequena, parte de seus membros possui mais de um papel. Isto é resultado da necessidade imposta pelo mercado de trabalho, de que um profissional da área da tecnologia da informação, necessite buscar, continuamente, a sua capacitação, de modo que possibilite a adequação e o acompanhamento das evoluções tecnológicas que ocorrem na informática.

4.1.3 O projeto

A ABC Informática tem, como foco principal, o desenvolvimento e a manutenção do seu sistema de gestão empresarial, assim como o desenvolvimento de algumas soluções personalizadas de acordo com as solicitações e demandas de seus clientes.

Para possibilitar acompanhar a evolução tecnológica e atender às novas demandas, tanto de sistemas personalizados, quanto da manutenção e evolução de seu sistema de gestão empresarial, surge a necessidade de se adotar boas práticas de desenvolvimento, como a definição de uma estrutura sólida nos seus produtos, de modo que esta comporte a evolução do sistema para melhor atender às necessidades de seus clientes. Necessidades estas que podem ser específicas de clientes, como também as impostas pelas legislações vigentes, como NF-e, CT-e, SPED, PAF-ECF, entre outros.

Sendo assim, o presente estudo buscou analisar e estudar um conjunto de soluções técnicas que se fazem fundamentais para a construção de uma boa arquitetura de um sistema, para que este possa contemplar as mudanças que venham a surgir com o tempo, além de garantir uma série de requisitos básicos - requisitos funcionais e requisitos não funcionas - que, de um modo geral, contribuam para o ganho de produtividade da equipe como um todo, como também auxilie na garantia da qualidade dos produtos.

4.2 Requisitos

Os requisitos estabelecidos no presente estudo visam a garantir a qualidade da solução de arquitetura que foi proposta, com também a adoção e aplicação de boas práticas da Engenharia de Software que foram relacionadas e detalhadas anteriormente.

Segue uma lista dos requisitos não funcionais que se julgam necessários na arquitetura que foi proposta para a utilização da ABC Informática em seu software de gestão empresarial:

- O sistema deve ser desenvolvido em camadas;
- O sistema deve ter comunicação com mais de um banco de dados;
- O sistema deve possuir interfaces portáveis para a plataforma Linux;
- O sistema deve suportar interfaces WEB;
- O sistema deve ter uma estrutura que viabilize sua manutenção;
- O sistema deve ter componentes reusáveis;
- O sistema deve fornecer alguns serviços SOAP para integrações;
- O sistema deve ter um bom desempenho, com operações de cadastros não superiores a 15 segundos;
- O sistema deve permitir automatizar testes;
- O sistema deve possibilitar sua personalização.

4.3 Definições quanto à Tecnologia

A ABC Informática tem a necessidade de que este sistema de gestão empresarial seja desenvolvido utilizando a linguagem C#, porque, inicialmente, o sistema será destinado à plataforma Windows, sendo executado sobre o *framework*.NET. Porém, futuramente, a empresa tem a intenção de portar parte de seus sistemas, como telas de frente de caixa, para a

plataforma Linux, com a utilização do *framework* Mono, o qual se compara ao *framework* .NET da Microsoft, pois isso permite a redução de custos dos clientes na compra de licenças de aplicativos, com a adoção de soluções livres.

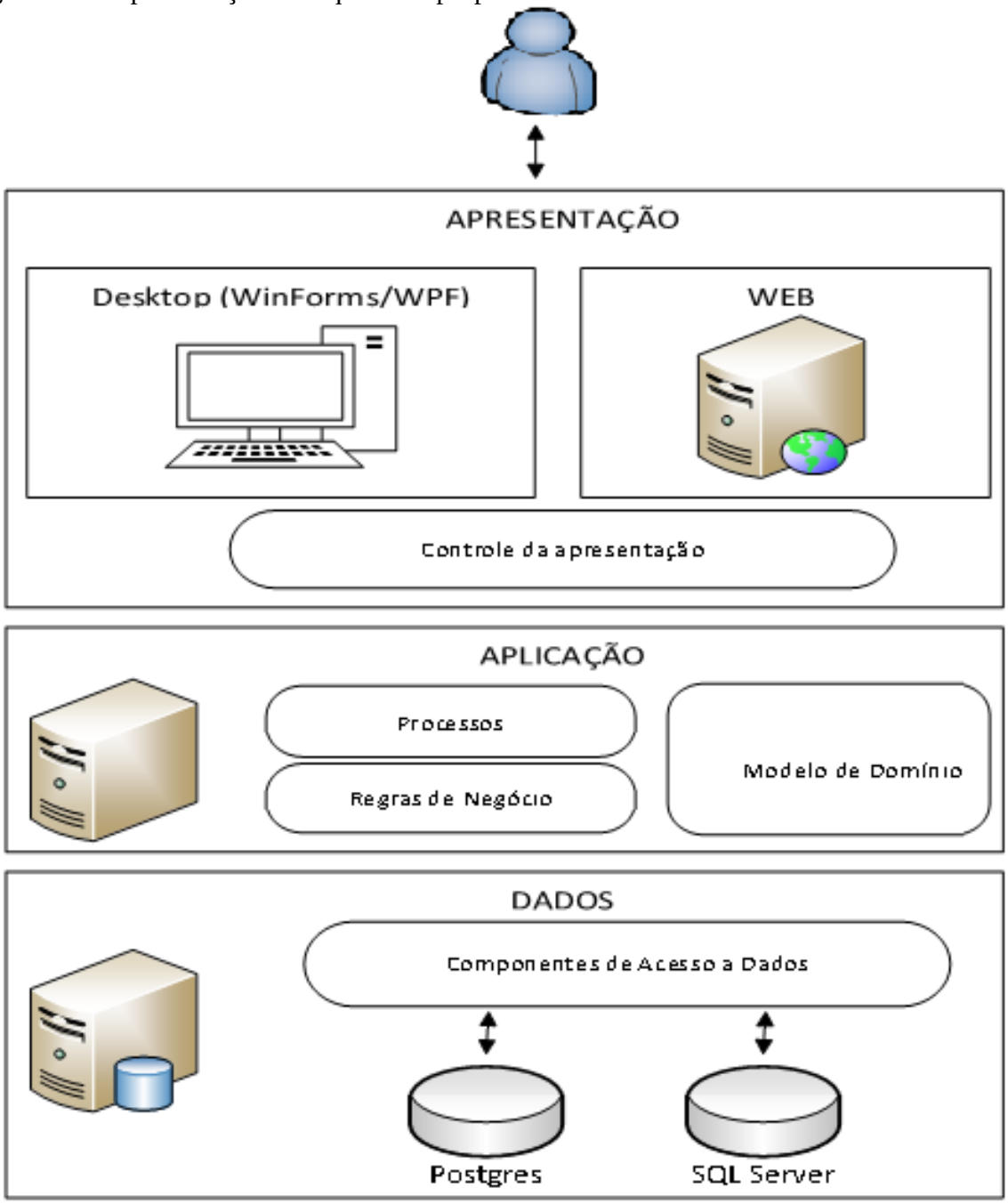
Em busca de também reduzir os custos com a infraestrutura necessária para execução dos aplicativos da ABC Informática, a empresa busca desenvolver aplicações que suportem mais de um banco de dados, devendo ter uma arquitetura de software que permita funcionar com o banco de dados *SQL Server* da Microsoft e com a alternativa *Open Source*, *PostgreSQL*.

4.4 Definições gerais da Arquitetura

Esta seção tem como objetivo apresentar a disposição da organização da arquitetura que foi proposta.

Na Figura 17, podemos ver um diagrama da visão geral da arquitetura, no qual está ilustrada a organização da mesma.

Figura 17 - Representação da arquitetura proposta



Fonte: Do autor.

Para elaborar o desenho da arquitetura apresentada na figura acima, foi levado em consideração um conjunto de definições que foram estabelecidas, a fim de garantir que todos os objetivos e requisitos sejam atendidos.

A seguir é listado o conjunto de camadas, nas quais a arquitetura foi dividida, a fim de melhor organizá-la:

- **Apresentação**: esta camada tem o objetivo de separar as informações relativas à apresentação dos dados, possuindo, como dependência, uma interface que deve ser fornecida pela camada de aplicação, permitindo que esta camada se preocupe apenas com sua responsabilidade: apresentar os dados;

- **Aplicação**: tem o objetivo de concentrar todas as regras e processos do negócio envolvidos na aplicação, permitindo fornecer uma interface à camada de apresentação a qual forneça todas as informações necessárias para as interações dos usuários, sendo responsável por gerir os dados, aplicando regras e executando as atividades necessárias;

- **Dados**: esta camada tem como objetivo fornecer uma interface às demais camadas que possibilite a persistência e recuperação de dados, concentrando tudo o que for necessário para o acesso aos dados.

A divisão da arquitetura, nestas três camadas, tem o objetivo de isolar e melhor organizar as partes básicas que compõem um software de gestão, que são a interface pela qual o usuário interage com o sistema, o processamento das informações que o usuário insere e requisita do sistema e o armazenamento das informações.

Com essa separação, é possível obter uma melhor organização da aplicação, pois se torna possível isolar as responsabilidades do sistema, facilitando a manutenção do sistema e permitindo adicionar novas funcionalidades a ele, sem que sua estrutura seja afetada.

4.5 Soluções avaliadas

Nesta seção são apresentadas as soluções avaliadas para o atendimento dos requisitos, e nela é listado um conjunto de soluções técnicas que possam ser adotadas para solucionar um determinado conjunto de requisitos e objetivos, estando descritos alguns detalhes da solução e seus objetivos propostos.

4.5.1 Persistência de dados

Com o objetivo de facilitar e abstrair a camada de acesso a dados da arquitetura proposta, foram avaliados dois dos mais conhecidos *frameworks* para o Mapeamento Objeto-Relacional (*Object-Relational Mapping* – ORM) em .NET, que são o *Entity Framework* e o *NHibernate*.

Um *framework ORM* tem como objetivo abstrair o acesso ao banco de dados através da orientação a objetos, pois ele cria uma ponte entre o modelo relacional (banco de dados) e o modelo orientado a objetos (as classes da aplicação). Com a utilização de um *framework ORM*, é possível diminuir a necessidade de se escrever consultas *SQL* para manipular informações num banco de dados, pois, geralmente, estas consultas são geradas automaticamente pelo *framework*, permitindo criar uma aplicação com um código mais elegante e, consequentemente, ampliando a facilidade de posteriores manutenções.

A utilização de um *framework ORM* para persistir as informações de uma aplicação para o banco de dados facilita e auxilia a reduzir o tempo de construção de uma aplicação, pois todo o serviço de "converter" as informações armazenadas no banco de dados para as classes do domínio da aplicação é realizado pelo *framework*. Sendo assim, consegue-se criar uma aplicação totalmente orientada a objetos, sem muito esforço, pois os desenvolvedores lidam com os dados, usando a mesma linguagem utilizada para construir a lógica de negócio da aplicação.

4.5.1.1 Entity Framework

O *Entity Framework* é um *framework* de persistência de dados que foi desenvolvido pela Microsoft a qual o disponibilizou com o *Visual Studio* e o *.NET Framework*.

A primeira versão do *Entity* foi disponibilizada em 2008, com o *Visual Studio* 2008 e o *.NET Framework* 3.5 SP1. Em 2010, foi lançada a versão 4.0 junto ao *Visual Studio* 2010 e o *.NET Framework* 4.0. Em 2012, com o lançamento do *Visual Studio* 2012 e o .NET *Framework* 4.5 veio o *Entity* 5.0. Atualmente, o *Entity Framework* se encontra na versão 6

beta, sendo que, nesta versão, esse projeto se tornou *Open Source*, estando licenciado sob a licença *Apache v*2.

Com o *Entity Framework* pode-se gerar o modelo Orientado a Objetos através de um banco de dados existente ou elaborar o modelo Orientado a Objetos e gerar a estrutura do banco de dados através dele. Outra maneira pela qual o *Entity* permite fazer o mapeamento entre o Modelo OO e o ER é com o *Code First*, que consiste em fazer o mapeamento através da codificação e criação das classes que representam o banco de dados, porém esta funcionalidade foi inserida nas últimas versões.

Outro ponto importante a destacar é que o *Entity* possui suporte para diversos bancos de dados como *SQL Server, Oracle, PostgreSQL, MySQL, DB2*, porém, para estes outros produtos que não são da Microsoft, é necessário utilizar um *provider* que é fornecido por terceiros, e efetuar o mapeamento específico do banco de dados utilizado.

4.5.1.2 NHibernate

O *NHibernate* é um *framework ORM* para a persistência de dados em .NET. Ele é um projeto *Open Source* que teve sua origem em 2005, junto a JBoss, pois se tratava, inicialmente, de uma versão portada do *Hibernate* para o Java, que é considerado um dos primeiros *frameworks ORM*. Em 2006, a JBoss abandonou o projeto e, desde então, ele é mantido inteiramente pela sua comunidade de usuários, possuindo funcionalidades específicas para .NET.

Apesar de o *NHibernate* ser um projeto *Open Source*, existe uma grande comunidade envolvida nesse projeto que, atualmente, encontra-se na versão 3.3. E, juntamente com essa grande comunidade do *NHibernate*, existe uma grande variedade de bibliotecas e extensões para ele que visam a prover uma série de funcionalidades e facilidades, como o *Fluent NHibernate*, que é uma biblioteca a qual fornece uma maneira de fazer o mapeamento entre o modelo relacional e o Orientado a Objetos via codificação e não com o tradicional e confuso arquivo XML do *NHibernate*.

Com o esse *framework*, é possível acessar diversos bancos de dados, como *SQL Server, Oracle, DB2, PostgreSQL, MySQl, Firebird*, entre outros, de forma nativa, sendo

necessário apenas mudar a configuração do *framework*, especificando o tipo de banco de dados e a *string* de conexão.

4.5.2 *Framework de Logs*

As bibliotecas de *Logs* têm como objetivo auxiliar a gravação de informações de *logs* que possam ser úteis, a fim de agilizar a identificação e correção de erros e *bugs* de um software através do rastreamento dessas informações gravadas, além de permitir monitorar e analisar a aplicação, em tempo real, depurando-as durante sua execução, facilitando encontrar soluções para possíveis problemas relatados pelos usuários.

Outra aplicação desse tipo de ferramenta é a gravação de informações relativas ao que um determinado usuário fez no sistema como, por exemplo, quais cadastros manipulou, a fim de fornecer informações de auditoria.

4.5.2.1 Log4net

O *Log4net* é uma biblioteca *Open-Source* mantida pela *Apache* e trata-se de uma versão portada para o .NET do *Log4j* do Java.

O diferencial do *Log4net* é que existe uma grande quantidade de *frameworks* que possuem uma interface para o acoplamento dessa biblioteca de uma maneira muito simples. Outros aspectos importantes é que toda a configuração dos formatos e sessões dos *logs* são feitos através de um arquivo de configuração XML, tornando muito fácil alterar alguma configuração do funcionamento da biblioteca.

A última versão disponibilizada do *log4net* é a 1.2.12, sendo disponibilizada em 16/07/2003.

4.5.2.2 NLog

O *NLog* é uma biblioteca *Open-Source* criada especificamente para a plataforma .NET, sendo mantida por sua comunidade de usuários.

É uma biblioteca bastante flexível, pois permite que sua configuração seja feita tanto via programação, como via um arquivo de configuração XML, fornecendo uma interface simples, pela qual serão gravadas as informações de *log*.

A última versão disponibilizada do *NLog* é a 2.0, sendo disponibilizada em 19/07/2011.

4.5.2.3 Logging Application Block

A *Logging Application Block* é uma ferramenta de *log* fornecida junto com a biblioteca *Microsoft Enterprise Library*, a qual possui um conjunto de várias ferramentas fornecidas pela Microsoft que visam a fornecer um conjunto de padrões e práticas da Microsoft. O *Enterprise Library*, além de fornecer uma ampla documentação destas ferramentas, disponibiliza, também, o código fonte, acompanhado de exemplos e testes que visam a auxiliar e a orientar a utilização dessas ferramentas.

O funcionamento do *Logging Application Block* consiste na utilização de um arquivo de configuração XML, no qual são feitas as definições de formatos de saída, *layout* dos *logs*, que serão gravados a partir de uma interface fornecida que permite que sejam gravadas as informações de *Trace* e *log* da aplicação.

A última versão disponibilizada do *Logging Application Block* é a 6.0, sendo disponibilizada em 21/05/2013.

4.5.3 *Framework* de Injeção de Dependência

A Injeção de Dependência é uma das formas de se conseguir e aplicar a Inversão de Controle que, de forma sucinta, tem o objetivo de reduzir o acoplamento entre as classes, objetos e componentes.

Com o auxílio de *Frameworks* de Injeção de Dependência é possível criar uma arquitetura com um bom *Design*, permitindo que a mesma não tenha problemas e dificuldades para evoluir, pois seus componentes possuem poucas dependências.

4.5.3.1 NInject

O *NInject* é um *framework Open-Source* desenvolvido para a plataforma .NET que tem como objetivo facilitar a utilização e aplicação da Injeção de dependência através de um componente simples e de fácil utilização.

Esse *Framework* está repleto de extensões que têm como objetivo facilitar a aplicação da injeção de dependência, pois toda a configuração da injeção de dependência pode ser feita via código. Atualmente se encontra na versão 3.0.

4.5.3.2 Spring.NET

O *Spring.NET* é um *framework* de injeção de dependência *Open-Source*, sendo que sua origem foi baseada no *framework Spring* do Java.

Esse *framework* também está repleto de extensões que auxiliam na injeção de dependência, e um dos seus diferencias é que todo o mapeamento da injeção de dependência é efetuado via um arquivo de configuração XML.

4.5.3.3 Unity

O *Unity* é um pequeno *framework* de injeção de dependência, que é mantido pela Microsoft, distribuído em conjunto com o *Enterprise Library*.

Um dos diferencias deste *framework* é o fato de ter uma ótima documentação, possuindo muitos exemplos que detalham e explicam muito bem seu funcionamento. Ele permite injetar dependência em métodos, propriedades e construtores, sendo bastante extensível e flexível.

4.5.3.4 StructureMap

O *StructureMap* é um *framework* de injeção de dependência e inversão de controle desenvolvido para .NET, sendo uma ferramenta *Open Source*.

O funcionamento do *StructureMap* é bastante flexível, podendo ser configurado tanto via arquivo de configuração XML, como via programação ou ainda via atributos de marcação, que “decoram” as próprias classes.

4.5.3.5 Simple Injector

O objetivo do *Simple Injector* é proporcionar aos desenvolvedores de .NET uma ferramenta simples, fácil e flexível para a aplicação da injeção de dependência.

O *Simple Injector* funciona com base nas definições feitas via código na aplicação, com o objetivo de simplificar e facilitar a injeção de dependência e inversão de controle da aplicação. O seu diferencial é sua grande simplicidade e a possibilidade da sua execução sobre o *framework* MONO.

4.5.4 *Framework* de Validações

Este tipo de *framework* tem o objetivo de auxiliar e facilitar a implementação de regras de negócio no modelo de domínio da aplicação, sendo fácil a implementação das regras que garantem que um objeto tenha todas suas propriedades essenciais devidamente informadas, evitando que o objeto fique num estado inválido, permitindo que, num processamento futuro desse objeto, seja criado um erro devido à falta ou mau preenchimento de alguma informação.

Através das validações podem-se especificar as restrições que a aplicação venha a ter em seu modelo de domínio, porém elas serão verificadas em várias outras camadas da aplicação, como na camada de apresentação, camada de acesso a dados, evitando que essas regras sejam duplicadas no código da aplicação, o que dificultaria sua manutenção.

4.5.4.1 NHibernate Validator

O *NHibernate Validator* é uma extensão nativa do *NHibernate*. Ela permite que sejam criadas regras e restrições no modelo de domínio de uma aplicação, no qual é possível especificar regras que serão validadas antes das entidades serem persistidas ao banco de dados.

A especificação dessas regras e validações pode ser feita através de arquivos de configuração XML, ou de atributos de marcação que “decoram” as próprias classes do domínio da aplicação.

Esse projeto possui seu código aberto e teve sua origem com base no *Hibernate Validator* do Java, porém, atualmente, ele está além de ser apenas um versão portada do projeto em Java, pois conta com uma série de melhorias e novas funcionalidades, estando na versão 1.3.2, disponibilizada em 29/06/2012.

4.5.4.2 FluentValidation

O *FluentValidation* é uma pequena biblioteca de validação, é *Open Source* e permite que seja criada um conjunto de regras e restrições no modelo de domínio da aplicação.

O funcionamento do *FluentValidation* consiste na definição de regras que são definidas e especificadas no código, com o auxílio de expressões *lambda.*

O *FluentValidation* encontra-se, atualmente, na versão 4.0 que foi disponibilizada em 18/07/2013.

4.5.5 WCF

O WCF ou *Windows Communication Foundation* é uma plataforma que tem como objetivo efetuar a comunicação entre dois aplicativos distribuídos. O WCF surgiu com a união de um conjunto de várias tecnologias de programação distribuídas, sendo concentradas dentro de um único modelo, baseando-se na arquitetura orientada a serviços (SOA).

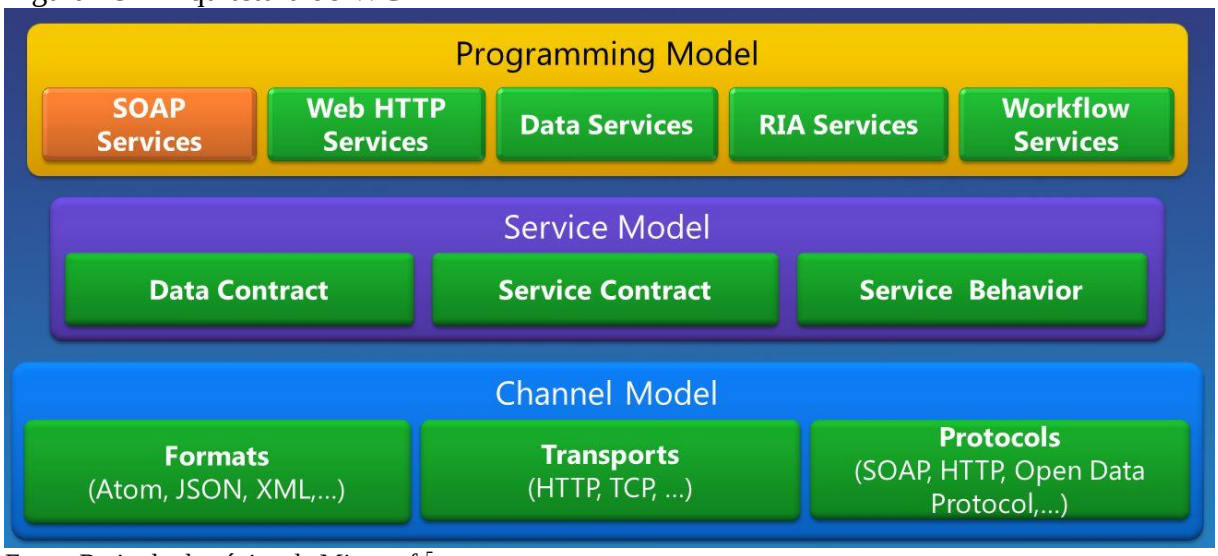
Essa plataforma torna a criação e o consumo de um serviço muito fácil, pois toda a infraestrutura necessária fica totalmente desacoplada das regras de negócios que serão expostas pela aplicação.

Com o WCF pode-se criar uma aplicação no modelo cliente/servidor, sendo que um cliente pode consumir vários serviços e um serviço pode ter vários clientes, permitindo criar uma aplicação, aplicando os princípios da arquitetura orientada a serviços.

Um dos benefícios do WCF é que ele torna possível a criação de uma aplicação que se comunique com outras aplicações, efetuando esta comunicação independente de sistema operacional e ou linguagem, pois, com o WCF, pode-se configurar uma aplicação para se comunicar com outra de várias maneiras, através de vários protocolos.

Na Figura 18, é possível verificar um digrama que apresenta os principais componentes que compõem a arquitetura do WCF.

Figura 18 - Arquitetura do WCF



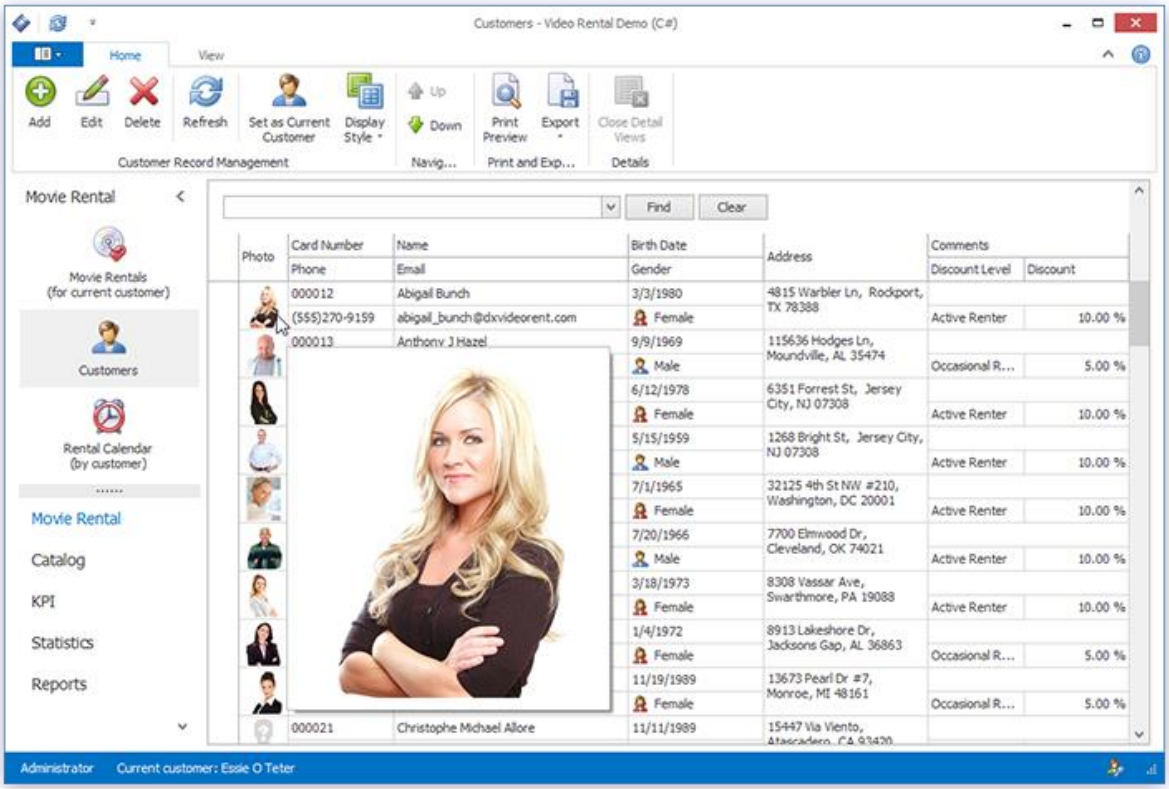
Fonte: Retirado da página da Microsoft[5].

4.5.6 *DevExpress*

A *DevExpress* é uma biblioteca de componentes visuais que tem como objetivo facilitar e auxiliar a criação de telas de aplicações as quais garantam aos usuários uma boa experiência durante sua utilização. Outro detalhe é que seus componentes possuem um design bem inovador, no estilo dos componentes da suíte Office da Microsoft, garantindo que a aplicação tenha um belo visual e seja muito funcional, pois esta biblioteca possui uma grande variedade de componentes que tornam mais fácil e mais produtivo a criação de uma tela mais elaborada, porque seus componentes são de fácil utilização e são muito customizáveis.

[5] Disponível em: <http://msdn.microsoft.com/en-us/library/vstudio/ms733128(v=vs.100).aspx>. Acesso em outubro de 2013.

Figura 19 - Exemplo de uma interface criada com os componentes *DevExpress*



Fonte: Retirado da página de DevExpress[6].

Na figura, pode-se observar um exemplo de uma interface criada com os componentes da *DevExpress*. Essa biblioteca possui uma grande variedade de componentes como diversos, *Grids, TreeViews, ListViews, Combos*, etc.

Seus componentes ajudam a criar uma aplicação com uma maior usabilidade e melhoram o visual da aplicação criada, sem falar que possui propriedades em seus componentes que facilitam a validação, exibição e mascaramento dos dados.

4.6 Avaliação das soluções

Esta seção tem como objetivo, exibir uma matriz de soluções escolhidas para a definição da arquitetura elaborada. Nela estão listadas as soluções, comparadas com um conjunto de critérios elaborados como, por exemplo, a aderência a padrões de projetos,

[6] Disponível em: < https://www.devexpress.com/Products/NET/Controls/WinForms/demos.xml>. Acesso em outubro de 2013.

quantos padrões aplica, a quais objetivos e requisitos atende, questões relacionadas à interoperabilidade, reutilização, interface limpa e bem definida, entre outros.

4.6.1 Persistência de dados

Para efetuar a persistência dos dados da aplicação a um banco de dados, foram avaliados o *Entity Framework* e o *NHibernate*. Na tabela a seguir, é realizada uma comparação entre as duas soluções, com o objetivo de comparar algumas das funcionalidades e dos diferencias que cada *framework* de persistência possui:

Tabela 3 - Comparativo de funcionalidades

Funcionalidade	**Entity**	**NHibernate**
Suporte a operações em lote	Não	Sim
Suporte a *Lazy Load*	Sim	Sim
Suporte a Cache de 2º nível	Não	Sim
Extensibilidade	Não	Sim
Portabilidade de Plataforma	Não	Sim
Suporte a diversos Banco de Dados	Sim, com provider de terceiros.	Sim, nativo
Suporte a tipos personalizados e Enums	Sim, mas com algumas restrições.	Sim
Facilidade de mudar o tipo de banco de dados	Complexo	Fácil
Single request operation	Não	Sim

Fonte: Do autor.

A seguir são detalhados os quesitos avaliados na tabela acima:

- Suporte a operações em lote: possibilidade de efetuar um conjunto de operações que serão executadas em um único comando ao banco de dados, evitando o acesso demasiado ao banco de dados;

- Suporte a *Lazy Load*: possibilidade de mapear atributos em uma entidade que somente serão carregados sob demanda como, por exemplo, no cadastro de uma classe pessoa, tem-se a referência da cidade desta pessoa e os dados desta entidade referenciada serão carregados somente quando este atributo for acessado;

- Suporte a *Cache* de 2º nível: possibilidade de efetuar *cache* para todas as sessões, compartilhado entre os processos e *threads*, podendo utilizar diferentes ferramentas como, por exemplo, o banco de dados *NoSQL Redis*

- Extensibilidade: possibilidade de estender funcionalidades do *framework*, customizando suas funcionalidades, como a adição de gatilhos disparados antes ou após o acesso ao banco de dados;

- Portabilidade de Plataforma: possibilidade de executar o *framework* em mais de um sistema operacional, como o suporte ao *Mono Framework* que permite executar a aplicação em ambiente UNIX;

- Suporte a diversos Bancos de Dados: possibilidade de executar uma mesma aplicação em diferentes bancos de dados, como *Microsoft SQL Server, Oracle, DB2, MySQL, PostgreSQL*, entre outros;

- Facilidade de mudar o tipo de banco de dados: possibilidade de alterar apenas os atributos de configuração e conexão, para que a aplicação funcione em outro banco de dados;

- Suporte a tipos personalizados e *Enums*: possibilidade de mapear campos customizados e/ou tipo de dados complexos, como o de campos que representem uma lista fixa, para *Enums* no sistema;

- *Single Request Operation*: possibilidade de retornar do banco de dados, vários objetos através da execução de uma única consulta.

Em relação à aderência de padrões de projetos destas duas soluções, pode-se afirmar que, em ambas as soluções, é possível empregar o uso de diversos padrões como, por exemplo:

- *Singleton*: podemos utilizar o padrão *Singleton* para garantir que seja aberta apenas uma conexão ao banco de dados por cada instância da aplicação;

- *Repository*: o *Repository* pode ser utilizado para simplificar a forma como são persistidos os registros ao banco de dados, em que cada tabela é manipulada como se fosse uma coleção de dados;

- *Data Access Object* (DAO): com o DAO podemos abstrair os métodos de persistência nas classes do domínio, em que cada objeto teria os métodos Salvar, Incluir, Excluir, BuscarPeloCodigo, por exemplo;

- *Data Mapper*: esse padrão tem por objetivo armazenar o mapeamento entre dois modelos como, por exemplo, mapear o modelo orientado a objetos ao modelo relacional de um sistema;

- *Unity Of Work*: o padrão *Unity of Work* pode ser empregado como uma maneira de controlar as transações da aplicação, quando um processo deve ser executado por completo, ou, em caso de algum erro, a aplicação deve voltar ao seu estado anterior;

- *Query Object*: esse padrão tem como objetivo permitir que sejam executadas consultas sobre os objetos, sendo manipulados de uma maneira semelhante às consultas SQL.

A partir da comparação que foi realizada entre o *Entity Framework* e o *NHibernate,* foi possível perceber que as duas ferramentas são muito semelhantes, tanto em questões relativas a funcionalidades quanto à performance, porém a que mais se destacou foi o *NHibernate.*

Entre os pontos de destaque do *NHibernate* estão o suporte ao Cachê de 2º nível que possibilita a implementação de cachê na camada de acesso ao banco de dados, o que pode vir a resultar numa melhora significativa de performance de uma aplicação; o suporte a tipos personalizados e *Enum* que permite o mapeamento de estruturas complexas do banco de dados.

A extensibilidade do *NHibernate* também chamou a atenção, porque, com esse recurso, pode-se implementar facilmente uma pequena camada de auditoria na camada de acesso a dados, pois, para isso, basta utilizar a grande variedade de eventos de interceptação que o *framework* disponibiliza, permitindo interceptar operações antes e depois da execução das consultas ao banco de dados, permitindo, também, criar tratamentos especiais antes de incluir ou excluir um determinado registro, por exemplo.

Ambos os *frameworks* possuem suporte a diversos bancos de dados, porém o suporte a estes banco de dados, no *NHibernate* é nativo, já, no *Entity,* depende de *providers* fornecidos por terceiros, o que pode vir a causar alguns problemas com o passar do tempo e com a evolução das funcionalidades do *framework.*

Mais um benefício do *NHibernate* é que ele oferece suporte para o *Mono framework* que se trata do *.NET Framework* para rodar aplicações C# em Linux, permitindo criar uma aplicação que rode em mais de uma plataforma de sistema operacional.

4.6.2 *Framework de Log*

Para a comparação das ferramentas de *Log,* foi criada uma tabela que compara e relaciona as funcionalidades de cada *framework* analisado. Segue a tabela comparativa:

Tabela 4 - Comparativo de funcionalidades dos *frameworks* de gravação de *log*

Informações de Log	**Log4net**	**NLog**	**Enterprise Library**
Prioridade	Não	Não	Sim
Informações do Processo/Thread	Sim	Sim	Sim
Informações ASP.NET	Sim	Sim	Sim
Informações de Trace	Sim	Sim	Sim
Exceptions	Sim	Sim	via exception block
Listeners	**Log4net**	**NLog**	**Enterprise Library**
ASP.NET Trace	Sim	Sim	Não
Colored Console	Sim	Não	Não
Console	Sim	Sim	Não
Base de Dados	Sim	Sim	Sim
Debug	Sim	Sim	Não
Log de Eventos	Sim	Sim	Sim
Arquivo	Sim	Sim	Sim
Email	Sim	Sim	Sim
Memória	Sim	Sim	Não
MSMQ	Não	Sim	Sim
Net Send	Sim	Não	Não
Remoting	Sim	Não	Não
Rolling File	Sim	Sim	Sim
Syslog (unix)	Sim	Não	Não
Telnet	Sim	Não	Não
UDP	Sim	Sim	Não

Fonte: Do autor.

Na tabela acima, estão relacionadas as funcionalidades suportadas por cada *framework*. Na parte superior da tabela, no grupo Informações de *Log*, estão listadas as funcionalidades relativas ao suporte à gravação de informações de depuração, que são úteis para identificar onde está ocorrendo um determinado erro. Já, o grupo de informações *Listeners* contém a relação de possíveis saídas que possamos dar ao *logs*, ou seja, são as

opções em que os framework *de log* podem ser configurados para armazenar e/ ou destinar os *logs* capturados, para que possam ser consultados posteriormente.

Como podemos ver na Tabela 5, no quesito de funcionalidades, os três *frameworks* são muito parecidos: por suporte a funcionalidades, eles ficariam classificados na seguinte ordem, *Log4Net, NLog e Enterprise Library*.

A fim de conhecer um pouco melhor o funcionamento e a configuração de cada ferramenta, foi criado um pequeno aplicativo com o intuito de simular a gravação de informações de *log* de uma aplicação, buscando medir a performance de cada ferramenta.

Essa aplicação de teste constituiu de um simples aplicativo que possui uma função que é responsável em efetuar a gravação de um determinado número de *logs* informado por parâmetro, medir o tempo de execução dessa tarefa e apresentá-la como saída. A aplicação de teste foi configurada igualmente para cada ferramenta para armazenar os *logs* registrados em um arquivo texto, com a informação da data e da hora do evento.

A seguir é apresentada uma tabela com os dados obtidos, comparando as ferramentas e apresentado o tempo que cada um levou para gravar um determinado número de *logs:*

Tabela 5 - Tempo de gravação de log

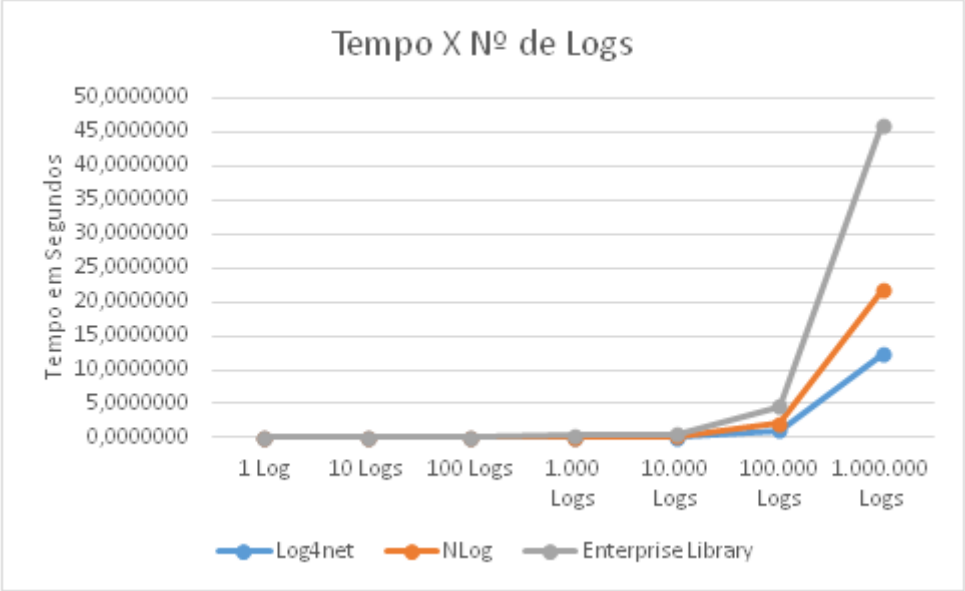
Framework	**1 Log**	**10 Logs**	**100 Logs**	**1.000 Logs**	**10.000 Logs**	**100.000 Logs**	**1.000.000 Logs**
Log4net	0,0000440	0,0001256	0,0008501	0,0807890	0,0770603	1,1114703	12,2322137
Nlog	0,0000612	0,0003041	0,0195200	0,0185645	0,1673430	2,1614581	21,6842321
Enter. Library	0,0001018	0,0006045	0,0469390	0,4097330	0,4347006	4,6949313	46,0052795

Fonte: Do autor.

Vale ressaltar que, para a execução desses testes, foi utilizado o mesmo ambiente para a execução do aplicativo de testes, sendo que cada teste foi executado três vezes e foi utilizada a média obtida entre as três execuções.

Abaixo segue o gráfico do desempenho do *framework*, no qual foi avaliado o tempo, em segundos, para a gravação de um dado número de *logs*:

Figura 20 - Comparativo de gravação de *logs*



Fonte: Do autor.

Como podemos observar no gráfico acima, em relação à performance o *Log4net* foi superior, seguido pelo *NLog* e por último ficou o *Enterprise Library*. Porém, em questão de facilidade de configuração o *NLog* foi muito superior, visto que o mesmo pode ser configurado via código, o que se torna muito mais simples quando comparado ao *Log4net* e ao *Enterprise Library* cuja configuração é por via de arquivo de configuração XML.

Um ponto negativo do *Enterprise Library*, além da performance, é a necessidade se referenciar mais dependências ao projeto para permitir o seu funcionamento, enquanto que no *Log4net* e no *NLog*, é preciso referenciar apenas uma única DLL que contém tudo que é necessário para o funcionamento.

Diante dos resultados obtidos, definiu-se a adoção pelo *Log4net*, visto que foi o mais rápido nos testes, assim como o que apresentou mais possibilidades de configuração como meio de saída dos *logs*.

Outro ponto que auxiliou para a sua escolha, mesmo sendo um pouco mais difícil de configurá-lo em comparação ao *NLog*, foi a possibilidade da fácil integração dele ao *NHibernate*, que possui uma interface de integração para capturar os *logs* gerados pelo *framework* de persistência, o que permite monitorar o que está acontecendo no acesso ao banco de dados.

4.6.3 *Framework* de Injeção de Dependência

Para a escolha do *framework* de injeção de dependência e inversão de controle, foi utilizado um estudo comparativo entre alguns *frameworks* de injeção de dependência o qual foi realizado por Palme (2013) e divulgado em seu blog[7]. Seu estudo consiste, basicamente, no desenvolvimento de um pequeno aplicativo o qual aplica a injeção de dependência em vários cenários distintos, sendo avaliado o tempo gasto para efetuar a injeção de dependência em um milhão de vezes, mensurando-se o tempo gasto em milissegundos. No blog de Palme (2013), é possível efetuar o *download* dos projetos elaborados para efetuar os testes.

Dentre os diversos *frameworks* de dependências avaliados por Palme (2013), encontram-se os cinco *frameworks* avaliados no presente trabalho, portanto os dados utilizados para a avaliação dos mesmos contemplaram apenas esses cinco *frameworks.*

A seguir é apresentada uma tabela que busca comparar as funcionalidades de cada *framework:*

Tabela 6 - Comparativo de funcionalidades dos *frameworks* de injeção de dependência

Framework	Performance	Configuração			Funcionalidades		
		Code	XML	Auto	Autowiring	Custom lifetimes	Interception
NInject	Lento	Sim	Sim	Sim	Sim	Sim	Não
SimpleInjector	Rápido	Sim	Não	Sim	Sim	Sim	Sim
Spring.NET	Muito Lento	Não	Sim	Não	Sim	Não	Sim
StructureMap	Médio	Sim	Sim	Sim	Sim	Sim	Sim
Unity	Médio	Sim	Sim	Sim	Sim	Sim	Sim

Fonte: Adaptado de *IoC Container Benchmark - Performance comparison*[8].

A seguir consta o detalhamento dos quesitos relacionados na tabela acima:

- Performance: o quesito performance é um conceito que foi atribuído a cada *framework* no qual foi avaliado o seu desempenho;
- Configuração: diferentes tipos de suportes para a configuração do *framework;*

[7] Blog do Daniel Palme: www.palmmedia.de/blog/.
[8] Dados utilizados foram adaptados do estudo realizado por DDD. Disponível em <http://www.palmmedia.de/blog/2011/8/30/ioc-container-benchmark-performance-comparison>. Acessado em julho de 2013.

- *Autowiring*: com esta funcionalidade podemos “decorar” uma classe com algum atributo de marcação, para que o *framework* de injeção de dependência saiba que ali deve ser efetuada a injeção de uma dependência especificada;

- *Custom lifetimes*: é a possibilidade de definir o ciclo de vida do objeto instanciado como, por exemplo, realizar uma nova instanciação do objeto para cada nova *thread*;

- *Interception:* esta funcionalidade permite que sejam definidos métodos, os quais com o auxílio do *framework* de injeção de dependência intercepte a execução de um determinado método na aplicação, permitindo, assim, tratar de alguma informação necessária.

Com o objetivo de avaliar o desempenho de cada ferramenta, a seguir é apresentada a tabela que contém a avaliação do desempenho de cada ferramenta em cenários distintos:

Tabela 7 - Comparativo de desempenho por cenário

Framework	**Singleton**	**Transient**	**Combined**	**Property**
NInject	8997	18530	50983	131647
SimpleInjector	112	109	117	186
Spring.NET	1641	16664	40305	98173
StructureMap	2178	2009	5984	19103
Unity	2915	4009	11660	34445

Fonte: Fonte: Adaptado de *IoC Container Benchmark - Performance comparison*[9].

A seguir segue a explicação de cada cenário avaliado na tabela de comparação de desempenho:

- *Singleton*: esse cenário visa à aplicação do padrão *Singleton*, que consiste em ter apenas uma instância de um determinado objeto na aplicação;

- *Transient*: nesse cenário é realizada uma nova instanciação do objeto a cada chamada;

- *Combined*: esse cenário consiste de uma interface que combina os dois cenários anteriores, com um objeto *Singleton* e um *Transient*;

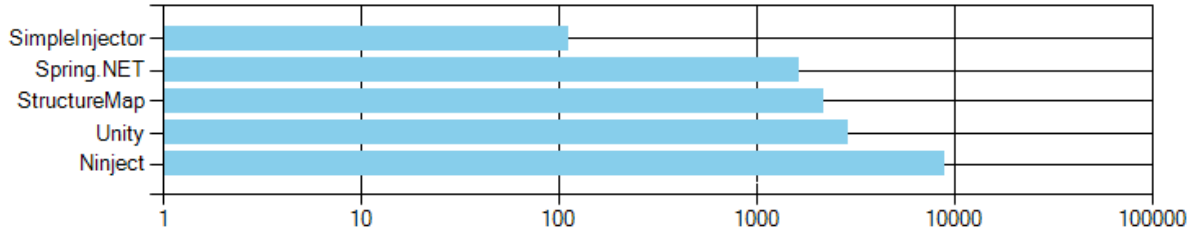
[9] Dados utilizados foram adaptados do estudo realizado por DDD. Disponível em <http://www.palmmedia.de/blog/2011/8/30/ioc-container-benchmark-performance-comparison>. Acessado em julho de 2013.

- *Property*: esse cenário consiste na injeção de dependência em um objeto a partir de uma propriedade do mesmo.

A Tabela 8 visa a mostrar a relação do desempenho de cada *framework* em diferentes situações nas quais é efetuada a injeção de dependência, em que é apresentado o tempo, em segundos, gasto para a execução de 1.000.000 de interações.

Abaixo segue um gráfico comparativo na resolução da injeção de dependência *Singleton*, ou seja, quando é realizada uma única instanciação do objeto.

Figura 21 - Gráfico de comparação de desempenho com *Singleton*



Fonte: Fonte: Adaptado de *IoC Container Benchmark - Performance comparison*[10].

Com base nas tabelas e gráficos acima, é possível verificar que o *Simple Injector* foi a ferramenta que obteve os melhores resultados de performance.

Outro quesito decisivo na escolha foi a facilidade de utilização e configuração das ferramentas e, nesse ponto o *Simple Injector* e o *StructureMap* se saíram muito próximos, visto que ambos possuem uma forma de utilização muito semelhante na configuração via código.

Diante dos fatos, optou-se pela utilização do *Simple Injector*, visto que foi um dos que melhor se saiu no quesito de performance e porque possui uma interface de configuração e utilização muito intuitiva e simples.

[10] Dados utilizados foram adaptados do estudo realizado por DDD. Disponível em <http://www.palmmedia.de/blog/2011/8/30/ioc-container-benchmark-performance-comparison>. Acessado em julho de 2013.

4.6.4 *Framework* de Validações

Para a escolha do *framework* de validação, foi realizado um conjunto de testes e pesquisas para determinar qual das duas ferramentas melhor se encaixaria na arquitetura proposta.

A seguir está uma pequena tabela que compara algumas funcionalidades de cada ferramenta:

Tabela 8 - Comparativo das funcionalidades dos *frameworks* de validação

Framework	**Configuração**			**Interceptors**	**Validação Condicional**
	XML	**Atributos**	**Code**		
NHibernate Validator	Sim	Sim	Não	Não	Não
FluentValidation	Não	Não	Sim	Sim	Sim

Fonte: Do autor.

Entre as possibilidades de configuração das regras, no *NHibernate Validator* à possível fazê-las a partir de atributos que irão decorar as classes, ou através de especificação das regras via arquivo XML, enquanto que no *FluentValidation* a especificação das regras ocorre via código.

Outro aspecto em que o *FluentValidation* se diferencia, é que ele suporta a criação de regras de validação condicionais, permitindo uma dinamicidade maior, além de que ele também suporta a criação de *Interceptors* que consistem na criação de eventos que podem ser disparados antes e após a validação das regras ser executada.

Optou-se pela escolha do *FluentValidation* para *framework* de validação, visto que, durante os testes das duas ferramentas, ele apresentou maior facilidade na sua utilização, pois como as regras de validação são escritas via código, ficou mais natural o entendimento e a definição das mesmas.

Outro ponto a favor do *FluentValidation* foi o fato de as regras de validação se concentrarem fora das classes de domínio da aplicação e não nelas, como ocorre com o *NHibernate Validator*, pois, com essa separação, não existe o acoplamento entre as regras de validação e o domínio.

4.6.5 Outras decisões tomadas

Esta seção tem como objetivo descrever outras decisões que foram tomadas na elaboração da arquitetura elaborada.

4.6.5.1 WCF

A utilização do WCF foi decidida baseada nos benefícios que ele vem a fornecer à arquitetura de uma aplicação, visto que um dos objetivos do trabalho é propor uma arquitetura para futuros projetos da ABC Informática LTDA, que venham a ser aplicações de pequeno e médio porte.

Com o WCF fornecendo toda a infraestrutura necessária para distribuir a aplicação e, aliado à diversidade de possibilidades que se tem com a configuração do WCF, é possível dimensionar a aplicação de acordo com as necessidades que ela venha a ter, ou seja, pode-se configurar facilmente o WCF para tratar de cada requisição como um único processo, a cada requisição feita por um serviço consumido, como pode ser configurado para que, a cada requisição, seja instanciado um novo processo, permitindo escalar a aplicação.

Outro ponto importante é que, com o WCF, é possível fornecer um conjunto de serviços através de um aplicativo *self-host* que consiste na criação de um executável o qual é responsável por fornecer um conjunto de serviços de uma forma simples e sem necessidade da configuração de uma complexa infraestrutura necessária. Ainda é possível publicar esses serviços a um servidor *Internet Information Services* (IIS), que é uma ferramenta semelhante ao Apache, responsável por disponibilizar serviços para WEB e, dessa forma é possível disponibilizar a aplicação para acesso a uma Internet, ou a uma rede corporativa, onde existe toda uma infraestrutura para configuração e gerenciamento de serviços que venham a necessitar de uma demanda maior.

Mais um fator importante é que, com o WCF, podemos configurar uma variedade de protocolos sobre os quais a aplicação venha a se comunicar, sendo necessário, basicamente, especificar o formato em que as mensagens serão trocadas (Atom, XML, JSON), o meio de transporte (HTTP, TCP, Remote) e o protocolo (SOAP, HTTP, Open Data Protocol).

Outro ponto importante sobre o WCF é que podemos configurá-lo de diferentes formas de acordo com o tipo de aplicação em que será utilizado, e de acordo com as necessidades do processo a que é destinado. O WCF possui basicamente três tipos de configuração para dimensionar o processamento das chamadas recebidas no servidor que está hospedando os serviços, que são:

- *Per call*: nesse modo, a cada chamada do serviço, é instanciado um novo serviço, sendo indicado para casos em que há um consumo alto de memória e um controle maior das conexões ativas, possibilitando ter uma escalabilidade maior, porém esta configuração não permite que o serviço tenha um estado para o controle de transações, pois é *stateless*;

- *Per session*: com esta configuração podemos controlar o estado do *service*, possibilitando criar transações entre as chamadas do serviço, pois o controle da instanciação do serviço é por sessão, permitindo ter uma escalabilidade maior;

- *Single*: nesse modo, é realizada uma única instanciação do serviço e todas as chamadas são atendidas por essa instância, sendo geralmente utilizada em casos nos quais é necessário compartilhar informações entre as chamadas e nos quais a escalabilidade não é necessária.

4.6.5.2 DevExpress

Optou-se pela utilização da suíte de componentes *DevExpress*, visto que a mesma fornece um grande conjunto de componentes que podem ser utilizados para a criação de telas que visem a uma melhor experiência ao usuário que vier a utilizar a aplicação.

Com os componentes da *DevExpress*, é possível criar telas que tenham uma aparência diferenciada, além da maior usabilidade que seus componentes fornecem, pois permitem definir facilmente máscaras para a entrada de dados, validar as informações e possuem diversos eventos para capturar e tratar melhor as interações dos usuários.

Um ponto importante a destacar é que os componentes da *DevExpress* são pagos, porém a ABC Informática LTDA possui a licença, pois já havia adquirido essa suíte mediante os benefícios fornecidos por ela, que já foram utilizados em outros projetos da empresa.

4.7 Especificação da arquitetura

Nesta seção, constam os detalhes finais da arquitetura definida, sendo apresentados maiores detalhes de toda a arquitetura, como o digrama de componentes que visa a exibir como os componentes, que compõem a arquitetura, relacionam-se e quais as suas dependências.

O objetivo é de detalhar o máximo possível a arquitetura elaborada, documentando todos seus aspectos, a fim de possibilitar o desenvolvimento de aplicações com base nessa arquitetura, auxiliando e servindo de guia para esclarecer e informar os detalhes de cada parte que compõe a arquitetura.

4.7.1 Visão geral da arquitetura

Com o objetivo de definir uma arquitetura para um sistema que possa evoluir facilmente, cujas manutenções não impactem sobre todo o sistema, gerando efeitos colaterais, buscou-se dividir cada camada em pequenos componentes que visam a isolar as responsabilidades para aumentar a coesão do sistema.

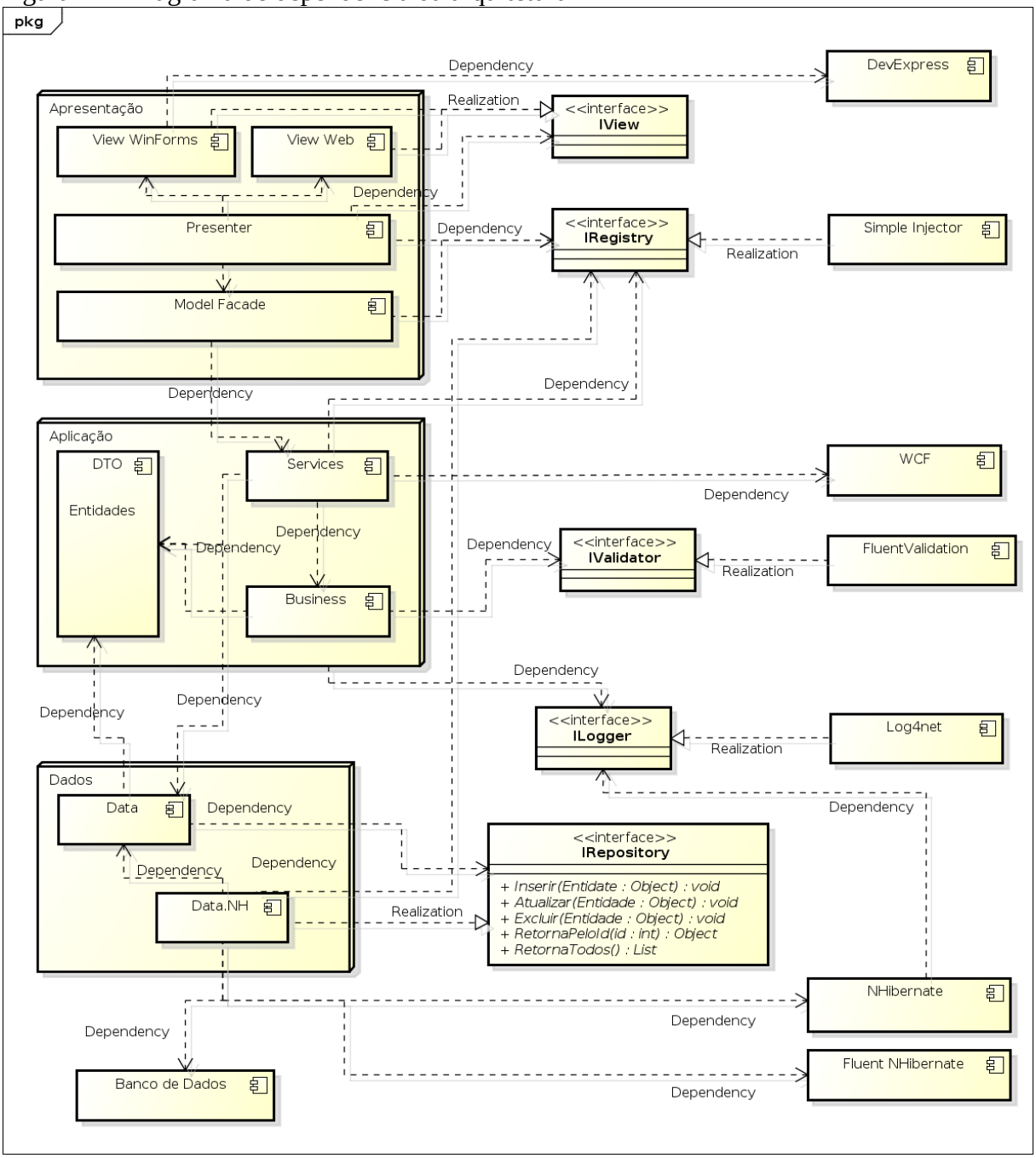
Para diminuir o acoplamento entre os componentes que irão compor o sistema, decidiu-se que seriam criados conjuntos de interfaces que têm o objetivo de funcionar como contratos nos quais os componentes baseiam as relações entre os componentes. Dessa forma, dois componentes estão relacionados mediante a interface, permitindo separar a implementação de cada componente e injetar as dependências necessárias com o uso de um *framework* de injeção de dependência.

Com a divisão da arquitetura em componentes menores, que se relacionem a partir de suas interfaces, é possível isolá-los de modo que uma alteração em um determinado elemento

não interfira no funcionamento dos demais. Outro benefício desse isolamento é a possibilidade da criação de testes de unidade e de integração, a fim de automatizar os testes de cada componente e/ou de um conjunto relacionado de componentes, o que permite assegurar a qualidade da aplicação.

A seguir é apresentado um digrama de componentes que busca apresentar uma visão geral da arquitetura proposta, exibindo os detalhes de como se dá a sua separação nas três camadas, assim como apresentar as dependências e as relações entre seus componentes.

Figura 22 - Diagrama de dependência da arquitetura



Fonte: Do autor.

Como é possível observar na Figura 22, a organização da arquitetura se dá da seguinte forma: na camada de apresentação estão concentradas as informações relativas às telas do sistema pelas quais o usuário irá interagir. Essa camada possui uma dependência para a camada de Aplicação a qual concentra toda a lógica do sistema como regra de negócios, processos e o modelo de domínio da aplicação.

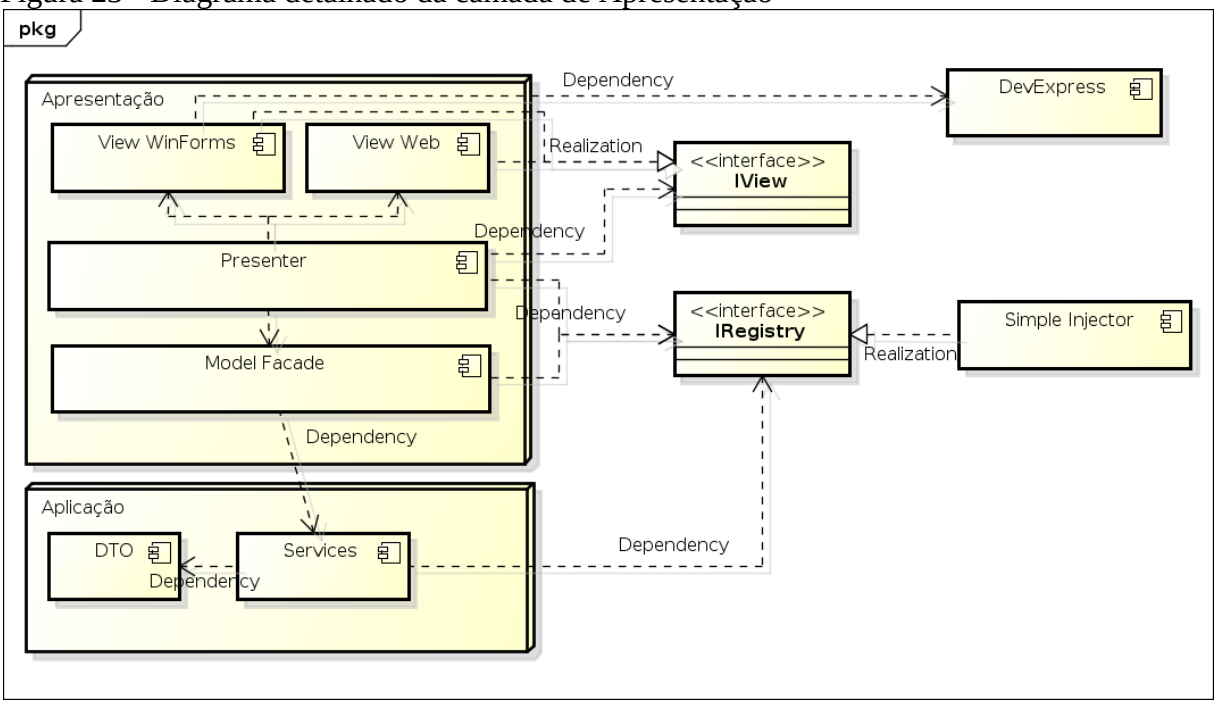
Já, a camada de Aplicação possui uma dependência para a camada de Dados, pois, na camada de Dados, é que ficarão concentradas as rotinas que irão fazer acesso ao banco de dados.

A seguir é detalhado o funcionamento de cada camada, explicando melhor seus detalhes e características.

4.7.1.1 Camada de Apresentação

A camada de Apresentação é a camada na qual vão estar concentrados todos os detalhes relacionados às telas e a lógica que trata as interações do usuário com o sistema. A Figura 23 apresenta o digrama de componentes que detalha o funcionamento e as dependências dessa camada.

Figura 23 - Diagrama detalhado da camada de Apresentação



Fonte: Do autor.

Como podemos verificar na Figura 23, a camada de apresentação está subdividida em basicamente mais quatro componentes, o *Model Facade*, *Presenter, View WinForms* e *View* WEB. O objetivo dessa separação é permitir que a camada de apresentação possa estar desacoplada do restante do sistema, pois um dos objetivos dessa arquitetura é permitir que ela possa ser aplicada a diferentes tipos de aplicações, tanto WEB como *Desktop*, ou até mesmo as duas opções em conjunto.

Para permitir que o sistema tenha a dinamicidade de ser acessado tanto na WEB, quanto no *Desktop*, sem que seja necessário manter toda a lógica que trata as interações do usuário duplicadas no sistema, foi aplicado o padrão *Model View Presenter* (MVP) que consiste de uma variação do padrão *Model View Controller*.

Com o MVP, é possível que seja isolado o código que trata das interações do usuário, de uma forma que possa ser utilizado indiferentemente da tecnologia aplicada na apresentação. Por esse motivo, na Figura 23, constam os componentes *View WinForms* e *View Web* que, na verdade, representam a camada *View* do padrão MVP.

A camada *Presenter* é representada pelo componente de nome homônimo, nele estão definidos as interfaces *IView* que devem ser implementadas pelas *Views* (telas) do sistema, isso permite que a camada *Presenter* do MVP trate as interações do usuário com as telas do sistema sem que o *Presenter* tenha conhecimento de como se dá a implementação das telas, resultando no desacoplamento das telas, permitindo que, por exemplo, um desenvolvedor cuide apenas da forma como uma tela seja construída, cuidando de questões da usabilidade e da aparência, enquanto outro trabalhe na lógica de como deverá funcionar a tela.

O componente *Presenter* vai controlar o fluxo das informações do sistema, pois os usuários vão interagir com as *Views* que solicitam as informações ao *Presenter*, e este irá tratar dessas solicitações e buscar o que precise no *Model*, para, então, retornar para as *Views* as informações.

Já, o componente *Model Facade* representa a camada *Model* do MVP, que são as regras do negócio, processos e acesso a dados do sistema, porém, na arquitetura que está sendo proposta, esse modelo está concentrado em outra camada: na Aplicação que disponibilizará os serviços que tratam dessas questões. Por esse motivo, foi criado o componente *Model Facade* que aplica o padrão *Facade,* que tem como objetivo criar uma fachada, agrupando os serviços que serão acessados na camada Aplicação.

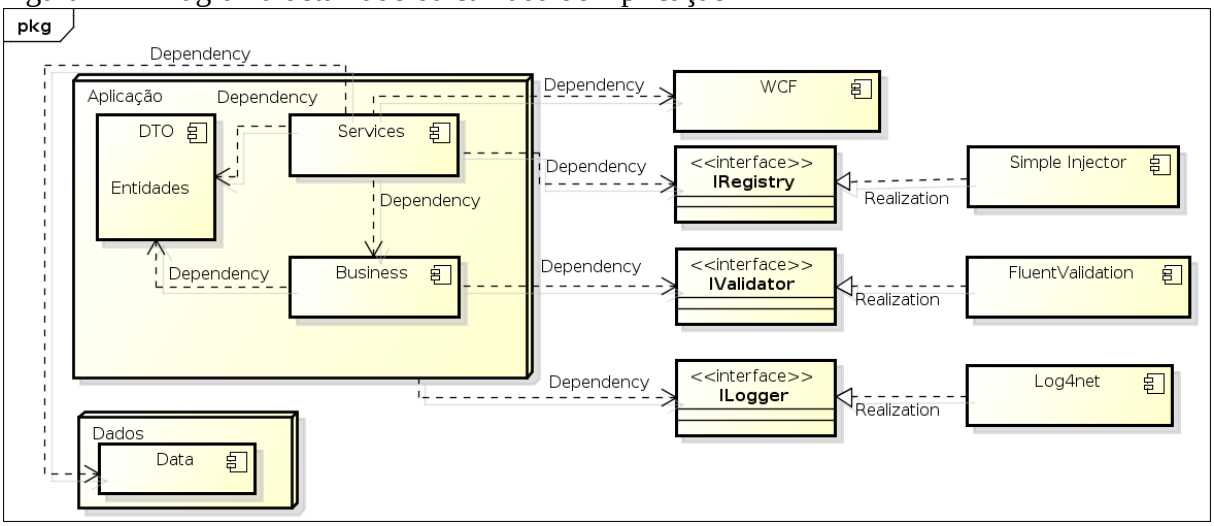
As telas para *Desktop* do sistema possuem uma dependência para o componente *DevExpress* o qual possui os diversos componentes que são adicionados às telas como botões, *Grids*, *Combos*, entre outros.

Já o componente *Simple Injector* é utilizado para realizar e injetar as dependências dos serviços no componente *Model Facade*, que são disponibilizados pela camada Aplicação e para realizar a instanciação desses serviços no componente *Presenter* que é quem os utiliza, portanto, o *Simple Injector* é utilizado para a injeção de dependência e para o *Factory* dos serviços utilizados.

4.7.1.2 Camada de Aplicação

Na camada de Aplicação, está concentrado todo o núcleo do sistema como o modelo de domínio, as regras de negócio e os serviços disponibilizados na camada de Apresentação.

Figura 24 - Diagrama detalhado da camada de Aplicação



Fonte: Do autor.

Como podemos observar na Figura 24, o componente DTO busca concentrar o modelo de domínio da aplicação, no qual estarão as entidades da aplicação. O componente *Business* irá conter as regras de negócio da aplicação e o componente *Services* os serviços que serão consumidos pela camada de Apresentação.

O componente *Business* utiliza as entidades que estão definidas no componente DTO, para efetuar o tratamento das regras de negócio da aplicação. No componente *Business*, é que

também vão estar as validações do sistema, que são implementadas com a utilização da *interface IValidator*, que são executadas pelo *FluentValidation* que gera exceções para a camada de apresentação, caso alguma das regras definidas não seja válida, essa deverá ser tratada e exibida ao usuário pela camada de Apresentação.

O componente *Services* utiliza o componente WCF para realizar a implementação dos serviços que serão disponibilizados para a Apresentação. Sendo assim, um serviço tem o objetivo de disponibilizar alguma funcionalidade com base nas Entidades do DTO, as quais são validadas pelo componente Business e são persistidas pela camada Dados.

Com o objetivo de desacoplar a camada que fará a persistência dos dados da camada Aplicação, a camada aplicação apenas conhece as interfaces que são disponibilizadas pelo componente Data da camada Dados, o que reduz o acoplamento entre os dois componentes e isola os diferentes comportamentos de cada camada.

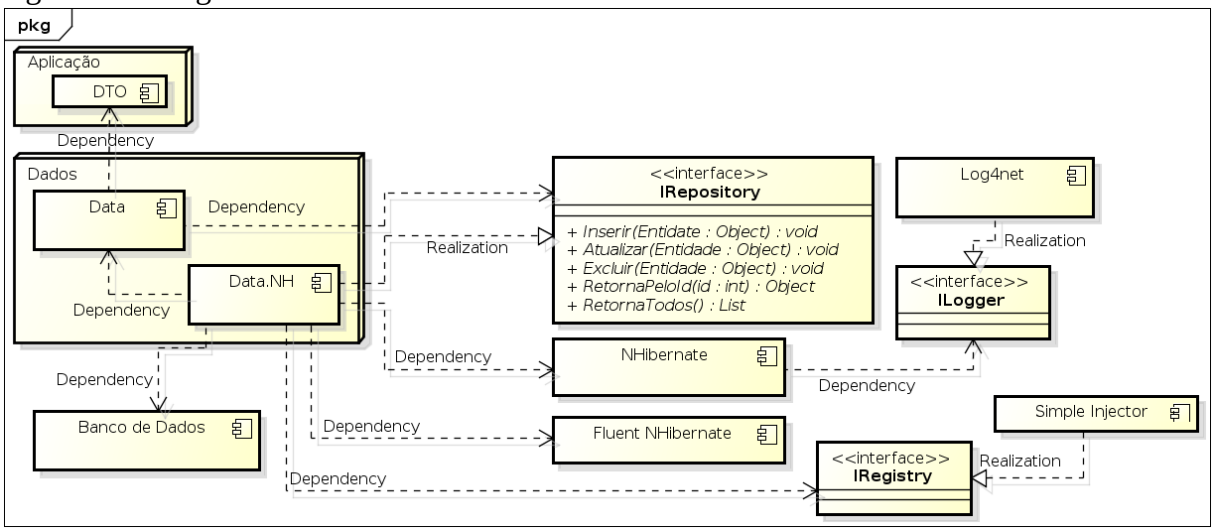
A interface *ILogger* disponibiliza os métodos para efetuar a gravação de diversos *logs* que estarão dispostos em vários locais da camada Aplicação, com o objetivo de facilitar a depuração da aplicação durante sua execução. O componente *Log4net* é o responsável por disponibilizar a implementação da interface *ILogger*, que trata da gravação dos *logs*.

O Componente *Simple Injector* é utilizado na camada de aplicação para realizar a injeção de dependências estabelecidas com a *interface IRegistry* e como construtor (*Factory)* dos objetos que realizam a implementação de interfaces como as definidas no componente Data, que tratam da persistência dos dados, ou de outras interfaces como a *ILogger* e a *IValidator*. Dessa forma, podemos substituir facilmente um componente do sistema, pois basta especificar para o *Simple Injector* qual vai ser a classe que deve ser instanciada quando for necessária uma determinada interface.

4.7.1.3 Camada de Dados

É a camada de Dados que irá concentrar todos os detalhes da persistência das informações para a aplicação. Na Figura 25, há um diagrama de componentes que detalha esta camada e apresenta seus componentes e suas dependências.

Figura 25 - Diagrama detalhado da camada de Dados



Fonte: Do autor.

Como podemos observar na Figura 25, a camada de dados é composta basicamente pelo componente Data, que tem o objetivo de disponibilizar as interfaces *IRepository* as quais serão utilizadas pela camada Aplicação.

Uma *interface IRepository* visa a estabelecer os detalhes para que uma determinada entidade do domínio, que consta no componente DTO da Aplicação, sejam persistidos ao banco de dados com a utilização do padrão *Repository.*

Já, o componente Data.NH é responsável por efetuar a implementação da *interface IRepository*. O Data.NH também faz o uso do componente *NHibernate*, que é um *framework* ORM que será responsável pela comunicação do sistema com o banco de dados, permitindo assim que seja utilizado diferentes banco de dados na mesma aplicação.

O componente *Fluent NHibernate* é utilizado pelo componente Data.NH para efetuar o mapeamento do modelo Orientado a Objetos ao modelo Relacional do banco de dados, através de classes que especificam esse mapeamento via código, o que torna mais fácil a manutenção e a especificação do banco de dados do sistema.

O *Simple Injector* também é utilizado nessa camada, para efetuar a injeção de dependências, de forma que, se um dia, a empresa ABC Informática fosse utilizar outro *framework* ORM, por exemplo, bastaria criar um novo componente que se implementa às *interfaces IRepository* para o acesso ao banco de dados e, então, informar ao *Simple Injetor* que injete, agora, o componente novo ao invés do Data.NH para as *interfaces IRepository* e,

então, teríamos trocado o componente de persistência dos dados sem necessitar alterar mais nada na aplicação.

O componente *Simple Injector* também é utilizado como *Factory* para os objetos do *NHibernate* que irão abrir e manter a conexão ao banco de dados e, com o objetivo de cada execução da aplicação ter somente uma conexão, é utilizado o padrão *Singleton* a partir do componente *Simple Injector*.

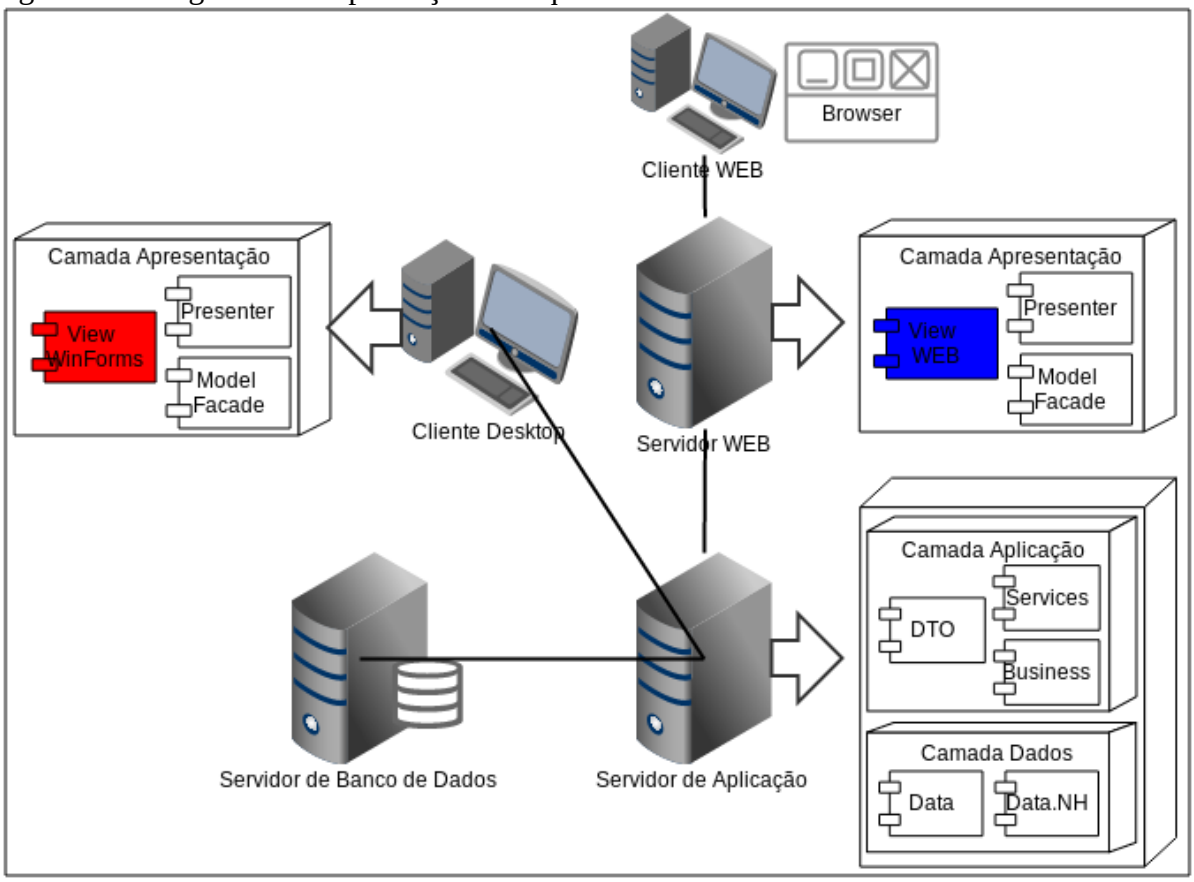
Em conjunto ao *NHibernate* é utilizado o componente *Log4net* que fará a gravação dos *logs* das consultas executadas pelo *NHibernate* no banco de dados.

4.7.2 Visão de implantação da arquitetura

Esta seção tem o objetivo de detalhar como se dá a implantação da arquitetura proposta, documentando a maneira de como os componentes são implantados e como ocorre a comunicação entre estas três camadas.

Basicamente, a arquitetura proposta necessita de um servidor de banco de dados, um servidor para a aplicação, os clientes e, opcionalmente, um servidor WEB, caso seja criado uma *interface* WEB para a aplicação. O diagrama da Figura 26 apresenta maiores detalhes de como os componentes estão distribuídos e como se comunicam.

Figura 26 - Diagrama de implantação da arquitetura



Fonte: Do autor.

Como podemos observar na Figura 26, o servidor de aplicação contém os componentes da camada de Dados que tem o objetivo de acessar o banco de dados, os componentes da camada de aplicação, os quais contém a camada de Aplicação, que visam a disponibilizar os serviços que serão acessados pela camada de Apresentação.

A camada de Apresentação pode ser tanto *Desktop* como WEB, isso depende dos requisitos da aplicação que serão desenvolvidos, porém, como podemos verificar na Figura 26, é possível ter um servidor WEB que contém a camada de Apresentação WEB, que é acessada pelos clientes com o *browser*, ou podemos ter outro cenário em que os clientes executam a camada de Apresentação para *Desktop*. Em ambos os casos, a camada de Apresentação se comunica com a camada Aplicação que estaria no servidor de Aplicação.

Um ponto importante a ressaltar é que, indiferentemente, de a aplicação ser *Desktop* ou WEB, ela irá utilizar os mesmos componentes na camada de Apresentação, apenas mudando os componentes que implementam a *interface* e que estão destacados em vermelho e azul na Figura 26.

A comunicação entre a camada de Apresentação (tanto WEB como *Desktop*) é a responsável por tratar e gerenciar as interações do usuário, pois a camada Aplicação é realizada pelo WCF que é responsável pela abstração de toda a comunicação entre as duas camadas.

O WCF faz todo o trabalho de disponibilizar as funcionalidades que são expostas no lado Servidor, que seria a camada Aplicação, para o lado cliente, cuidando e gerenciando toda a troca de informações, fazendo todo o trabalho de gerenciar e controlar a forma de como é realizada a conexão entre os dois pontos, como o serviço de serializar e desserializar os objetos que trafegam de um lado para o outro.

Um aspecto muito interessante do WCF é que, com ele, podemos definir várias formas de como se dará a comunicação entre o servidor que disponibiliza os serviços e os clientes que consomem o serviço. É possível configurar vários protocolos de comunicação no WCF, como HTTP, TCP, Named Pipes, MSMQ, além de suportar diversos formatos para as mensagens transportadas, sendo: SOAP, non-SOAP XML, RSS, JSON e formatos binários.

Outro detalhe do WCF é que os serviços podem ser publicados no (IIS), que se compara com um servidor Apache para o PHP, ou podemos criar aplicativos *sefl-host* que irão disponibilizar os serviços. A melhor forma a ser utilizada irá depender do porte da aplicação, pois, quanto maior for a aplicação, a melhor escolha seria publicar os serviços no IIS, pelo fato de ser mais escapável, permitindo expandir a aplicação com a alocação de mais recursos ao servido.

Portanto, a forma de como será distribuída a implantação da arquitetura proposta pode variar com as necessidades da aplicação em que for utilizada, pois permite suportar aplicações de pequeno porte, mais simples e aplicações maiores que demandam uma infraestrutura maior.

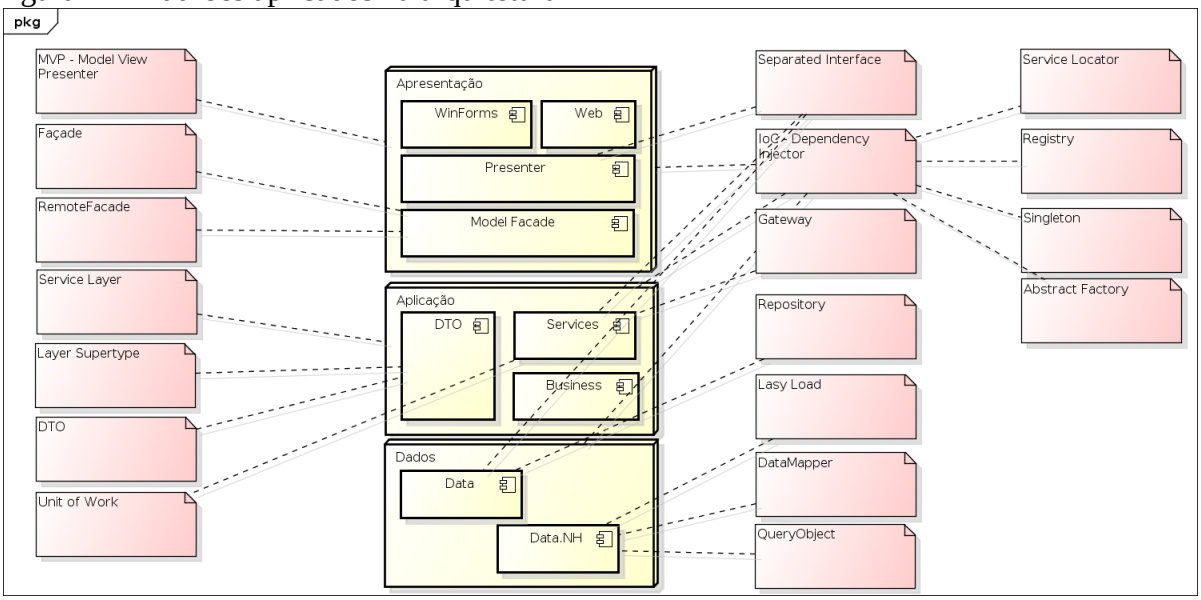
4.7.3 Padrões de Projetos aplicados

Esta seção tem o objetivo de detalhar a arquitetura em relação a que padrões de projetos foram utilizados e a forma de sua aplicação dentro da arquitetura, visto que o

objetivo do presente trabalho é propor um modelo de arquitetura que aplique padrões de projetos.

Na Figura 27, são apresentados os padrões utilizados na arquitetura relacionados com o local em que foi aplicado:

Figura 27 - Padrões aplicados na arquitetura



Fonte: Do autor.

Como podemos verificar na Figura 27, foram aplicado um conjunto de 18 padrões de projetos em diferentes locais na arquitetura. A Tabela 9 a seguir, tem o objetivo de detalhar estes padrões utilizados, apresentando uma breve descrição de cada padrão, a camada onde foram aplicados e alguns detalhes referente à sua utilização na arquitetura.

Tabela 9 - Padrões utilizados na arquitetura.

Padrão	**Objetivo do padrão**	**Camada**	**Detalhes da utilização**
Registry	Um objeto conhecido utilizado para localizar outros objetos.	Apresentação, Aplicação e Dados.	É utilizado a partir do componente *Simple Injector* em todas as camadas da arquitetura, pois é informada uma interface ao *Simple Injector* e ele localiza a sua implementação, efetuando a instanciação do objeto que implementa esta interface.
Facade	Fornece uma interface unificada para um conjunto de subsistema.	Aplicação	Utilizado na camada Aplicação, no componente *Services*, com o objetivo de disponibilizar um serviço que é composto por outros serviços a partir de uma única interface.
Remote Facade	Promove uma interface unificada para um conjunto de subsistemas que são acessados via rede.	Apresentação	Utilizado na camada Apresentação, no componente *Model Facade*, tendo como objetivo disponibilizar uma interface única, que agrupar serviços que são consumidos pela camada de Apresentação e são disponibilizados pela camada Aplicação.
Service Layer	Define o limite de uma aplicação com uma camada de serviços que disponibilizam um conjunto de operações e coordena a resposta a cada operação.	Aplicação	O padrão *Service Layer* foi aplicado na camada de Aplicação, a fim de criar uma camada que concentra todas as funcionalidades e regras de negócio, disponibilizando um conjunto de serviços que possam ser consumidos por outras aplicações e ou camada de Apresentação pela qual o usuário interage.
Model View Presenter	Divide a aplicação em três regras distintas, a fim de isolar as diferentes responsabilidades em *Model*, *View* e *Presenter*.	Apresentação	A utilização deste padrão permite isolar as regras de negócio e acesso a banco de dados em uma camada *Model*, isola o controle do fluxo das informações na camada *Presenter* e isola a interface do usuário na camada *View*, permitindo assim desenvolver cada parte separadamente, isolada das demais, sendo utilizado na camada Apresentação.
Layer Supertype	Um tipo que atua como um supertipo para os demais componentes de uma camada.	Aplicação	Este padrão foi utilizado na camada Aplicação no componente *DTO*, onde foi definida uma classe *Entidade*, da qual todas as demais classes do domínio herdam.
Data Transfer Object	Este padrão tem o objetivo de definir objetos que trafegam entre os processos da aplicação, sobre os quais são efetuadas as operações do sistema.	Aplicação	Foi utilizado no componente *DTO*, o qual é utilizado em todas as camadas da aplicação, pois são sobre estes objetos que são realizados as operações do sistema.
Separated Interface	Define a interface de uma classe em um componente diferente de suma implementação.	Apresentação, Aplicação e Dados.	Este padrão foi utilizado em todas as camadas da aplicação, pois foram definidas as interfaces em um componente, e sua implementação foi realizada em outro componente, desta forma a implementação é realizada com base na interface de uma classe, e é realizada a injeção de dependência na execução da aplicação, permitindo substituir facilmente um componente por outro.
Gateway	Este padrão tem o objetivo de encapsular em um objeto o acesso a algum recurso ou sistema externo.	Apresentação	O funcionamento do WCF tem como base o padrão *Gateway* para encapsular a comunicação entre os serviços e seus clientes.
Unit of Work	Um objeto que contém a relação de objetos que foram afetados por uma transação comercial.	Aplicação e Dados	Este padrão foi utilizado nas camadas Aplicação e Dados, com o objetivo de controlar as transações dos processos do sistema, garantindo que um processo não tenha terminado e tenha alterado o estado de um ou mais objetos, mesmo que um erro tenha ocorrido.
Query Object	Um objeto que represente uma consulta do banco de dados.	Dados	Este padrão é aplicado no componente *NHibernate*, que encapsula as consultas SQL do banco de dados em objetos, permitindo efetuar consultas ao banco de dados somente com código orientado a objetos, sem consultas SQLs. Sendo utilizado na camada de Dados.
Data Mapper	Este padrão tem o objetivo de isolar o mapeamento entre o modelo relacional do banco de dados ao modelo orientado a objetos.	Dados	É aplicado com o componente *Fluent NHibernate*, que disponibiliza uma interface para efetuar o mapeamento entre o banco de dados e o modelo orientado a objetos de um sistema, sendo aplicado na camada de Dados no componente Data.NH.

Padrão	**Objetivo do padrão**	**Camada**	**Detalhes da utilização**
Repository	Disponibiliza uma interface para manipular dados num formato de colação de dados.	Dados	Este padrão foi utilizado na camada Dados, para isolar a manipulação de objetos que são consultados e armazenados no banco de dados, permitindo disponibilizar uma interface para a aplicação que seja indiferente da forma como os dados são armazenados.
Lazy Load	Um objeto que não contém todos os dados necessários, mas sabe como buscá-los.	Dados	É utilizado em conjunto com o *NHibernate*, o qual implementa uma forma de mapear relacionamentos entre dois objetos, que só são retornados do banco de dados quando for acessada a propriedade deste relacionamento.
Singleton	Este padrão tem o objetivo de manterá existência de apenas uma instancia de um objeto, mantendo um ponto global de acesso.	Apresentação, Aplicação e Dados	O *Singleton* é utilizado em vários lugares na arquitetura, como no da sessão do *NHibernate*, no objeto validador de cada classe do *Fluent Validation*, no objeto que fornece a interface para a gravação dos *logs* do *Log4Net*.
Abstract Factory	Permite a criação de objetos por meio de uma única interface, sem que a classe concreta seja especificada.	Apresentação, Aplicação e Dados	Este padrão é utilizado pelo componente *Simple Injector*, pois ele disponibiliza uma interface que faz a construção e instanciação de determinados objetos do sistema.
Dependency injection	É utilizado para manter um baixo acoplamento entre os diferentes componentes de um sistema.	Apresentação, Aplicação e Dados	O uso do *Simple Injector*, permite injetar facilmente as dependências de um objeto de forma automática, pois ele é capaz de detectar estas dependências e injeta-as no objeto.
Service Locator	Este padrão visa encapsular e abstrair os processos envolvidos na obtenção de um objeto.	Apresentação, Aplicação e Dados	Este padrão é utilizado pelo componente Simple Injector, pois a partir dele é possível localizar a instância de um objeto a partir de sua interface, pois o *Simple Injector* localiza a implementação desta classe em tempo de execução, o que permite substituir algum componente do sistema facilmente, pois basta informar a ele qual o componente que contém a implementação daquela interface.

Fonte: Do autor.

5 VALIDAÇÃO DA PROPOSTA

A fim de validar a solução proposta, foi desenvolvido um pequeno protótipo, o qual contém um conjunto básico de funcionalidades que tem como objetivo validar a arquitetura que foi proposta, verificando se todos os requisitos, objetivos, funcionalidades e estruturas, foram plenamente atendidos através das soluções técnicas que foram definidas e devidamente testadas, mostrando-se aderentes à utilização e aplicação de padrões de projeto.

5.1.1 Objetivos

O objetivo do protótipo foi desenvolver um sistema que permita gerir as informações de uma clínica de saúde, aplicando os conceitos da arquitetura proposta no presente trabalho, a fim de permitir avaliar a arquitetura proposta, num cenário que poderia se aproximar de um caso real, porém em dimensões menores.

As necessidades desse sistema consistem basicamente em gerenciar informações dos pacientes que frequentam a clínica, dos médicos que lá trabalham, das especialidades, dos agendamentos e sobre consultas realizadas, de possíveis tratamentos indicados e de possíveis medicamentos receitados durante a consulta.

5.1.2 Requisitos

De acordo com Pressman (2010),

> Uma compreensão completa dos requisitos de software é fundamental para um bem-sucedido desenvolvimento de software. Não importa quão bem projetado ou quão bem codificado seja um programa mal analisado e especificado desapontará o usuário e trará aborrecimentos ao desenvolvedor. (PRESSMAN, 2010, p.231)

Assim, a seguir estão descritos os requisitos funcionais e casos de uso que devem ser contemplados pela ferramenta, os quais proverão a compreensão sobre as funcionalidades, restrições, interfaces e outros elementos que o sistema deve compreender.

Tabela 10 - Descrição do Requisito Funcional 1

Requisito	RF01	**Nome**	Permitir manter pacientes
Descrição			
O sistema deve permitir manter uma relação de pacientes que frequentam a clínica, armazenando informações do nome, endereço completo, data de nascimento, sexo, CPF, RG, telefone e email.			

Fonte: Do autor.

Tabela 11 - Descrição do Requisito Funcional 2

Requisito	RF02	**Nome**	Permitir manter médicos
Descrição			
O sistema deve permitir manter uma relação dos médicos que trabalham na clínica, armazenando informações do nome, endereço completo, data de nascimento, sexo, CPF, RG, telefone, email e especialidade do médico.			
Restrições			
Um médico só pratica uma determinada especialidade.			

Fonte: Do autor.

Tabela 12 - Descrição do Requisito Funcional 3

Requisito	RF03	**Nome**	Permitir manter especialidades
Descrição			
O sistema deve permitir manter uma relação de especialidades que são atendidas na clínica.			

Fonte: Do autor.

Tabela 13 - Descrição do Requisito Funcional 4

Requisito	RF04	**Nome**	Permitir manter medicamentos
Descrição			
O sistema deve permitir manter uma relação de medicamentos que possam ser indicados a um paciente em uma consulta.			

Fonte: Do autor.

Tabela 14 - Descrição do Requisito Funcional 5

Requisito	RF05	**Nome**	Permitir manter tratamentos
Descrição			
O sistema deve permitir manter uma relação de tratamentos que possam ser indicados a um paciente em uma consulta.			

Fonte: Do autor.

Tabela 15 - Descrição do Requisito Funcional 6

Requisito	RF06	**Nome**	Permitir manter especialidades
Descrição			
O sistema deve permitir manter uma relação de especialidades que atende na clínica.			

Fonte: Do autor.

Tabela 16 - Descrição do Requisito Funcional 7

Requisito	RF07	**Nome**	Permitir manter agendamentos de consultas
Descrição			
O sistema deve permitir manter uma relação de consultas agendadas de cada paciente para um determinado médico numa data e hora especificadas.			
Restrições			
Uma consulta agendada para um determinado médico, não pode conflitar com outro agendamento para o mesmo médico data e hora.			

Fonte: Do autor.

Tabela 17 - Descrição do Requisito Funcional 8

Requisito	RF08	**Nome**	Permitir agendar
Descrição			
O sistema deve permitir manter uma relação de consultas realizadas pelo paciente, que consiste em armazenar informações que são preenchidas pelo médico no ato da consulta, como vincular possíveis tratamentos e medicamentos relacionados à consulta.			

Fonte: Do autor.

Tabela 18 - Descrição do Requisito Funcional 9

Requisito	RF09	**Nome**	Permitir manter ficha de informações
Descrição			
O sistema deve permitir manter uma ficha de informações por especialidade para cada paciente, que deve ser atualizada sempre que um médico da especialidade observar esse paciente.			

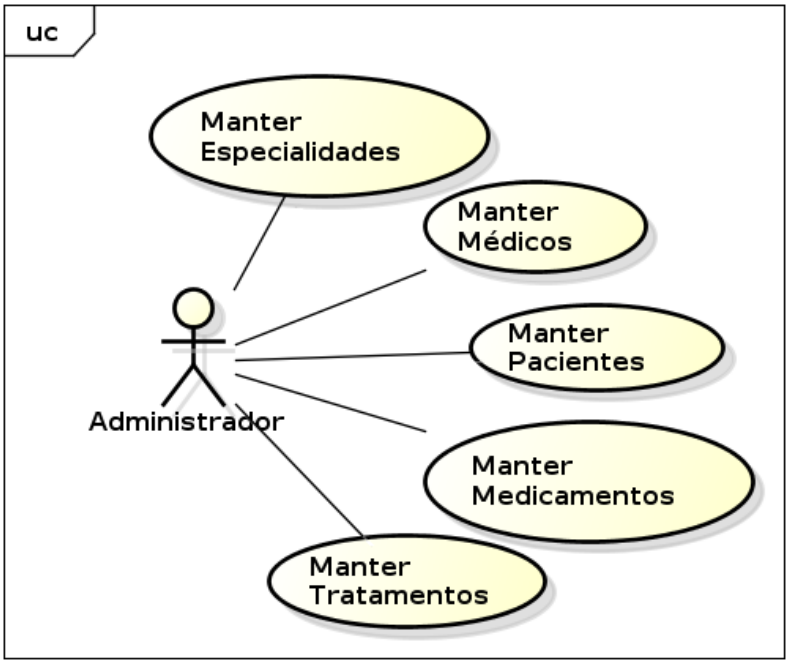
Fonte: Do autor.

A partir do levantamento dos requisitos listados acima, foram identificados os casos de uso e os atores envolvidos no protótipo.

O funcionamento do protótipo se dá basicamente pelos seguintes atores: o administrador, a secretária, os médicos e os pacientes.

A seguir é apresentado o Caso de Uso do Administrador do sistema:

Figura 28 - Caso de Uso do Administrador



Fonte: Do autor.

Na Tabela 19, é apresentada a descrição do Caso de Uso envolvendo o Administrador do sistema.

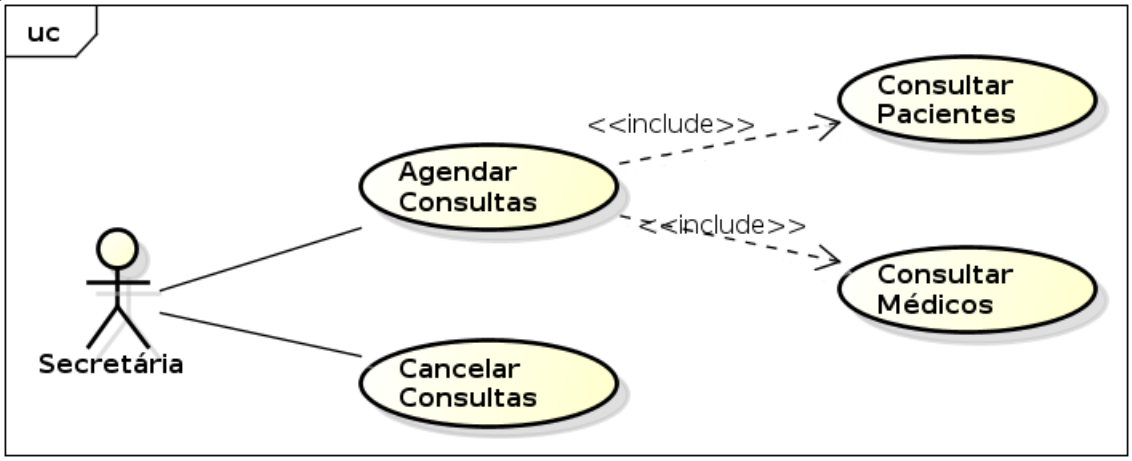
Tabela 19 - Descrição do Caso de Uso do Administrador

Caso de Uso	Descrição
Manter Especialidades	Incluir, alterar e excluir Especialidades. Neste caso de uso, o administrador deve cadastrar as especialidades que a clínica de saúde atende.
Manter Médicos	Incluir, alterar e excluir Médicos. Neste caso de uso, o administrador deve cadastrar os médicos que atuam na clínica de saúde.
Manter Pacientes	Incluir, alterar e excluir a relação de pacientes que frequentam a clínica de saúde.
Manter Medicamentos	Incluir, alterar e excluir Medicamentos. Neste caso de uso, o administrador deve cadastrar os possíveis medicamentos que possam ser registrados por um médico em uma consulta.
Manter Tratamentos	Incluir, alterar e excluir Tratamentos. Neste caso de uso, o administrador deve cadastrar os possíveis tratamentos que possam ser registrados por um médico em uma consulta.

Fonte: Do autor.

Na Figura 29, é exibido o Caso de Uso da Secretária:

Figura 29 - Caso de Uso da Secretária



Fonte: Do autor.

A seguir, na Tabela 20, está descrito o detalhamento do Caso de Uso da Secretária.

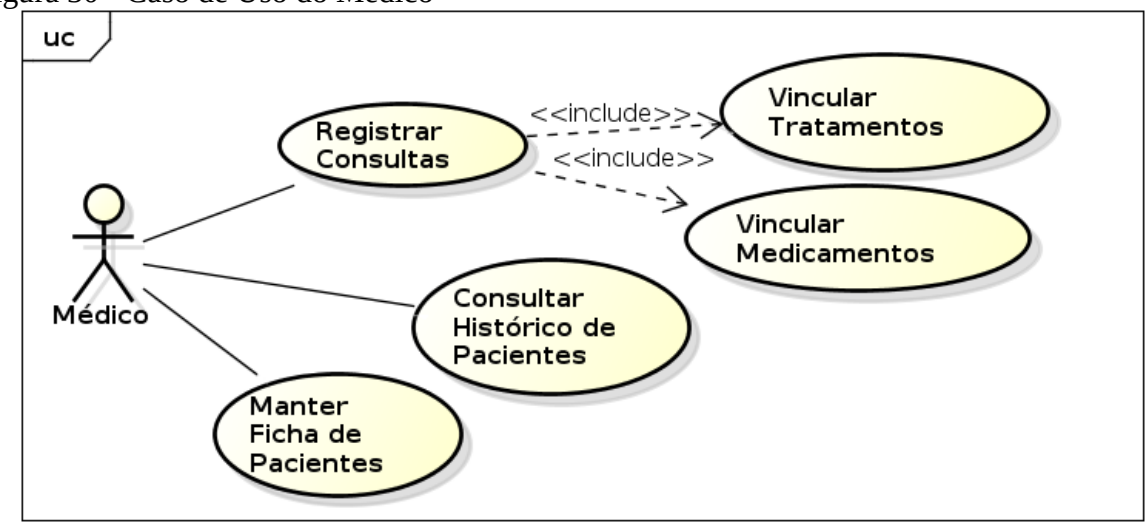
Tabela 20 - Descrição do Caso de Uso da Secretária

Caso de Uso	Descrição
Agendar Consultas	Neste caso de uso, a secretária efetua o cadastro do agendamento de consultas de um paciente para um determinado médico em uma data e hora especificada, a qual o sistema deverá validar para que não haja conflitos.
Cancelar Consultas	Neste caso de uso, a secretária poderá fazer o cancelamento de uma consulta agendada anteriormente pelo paciente.

Fonte: Do autor.

A seguir, é possível observar o Caso de Uso que envolve os Médicos:

Figura 30 - Caso de Uso do Médico



Fonte: Do autor.

Segue abaixo, a descrição do Caso de Uso dos Médicos:

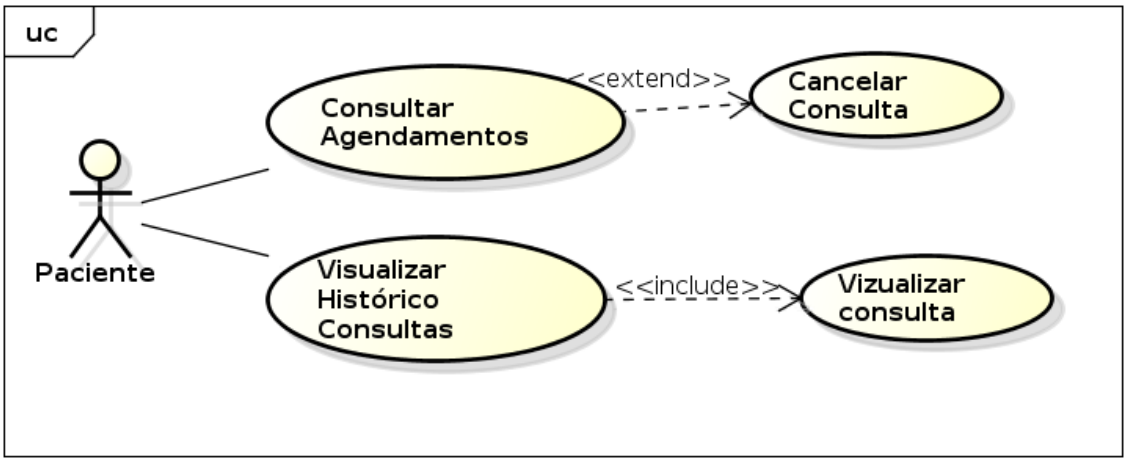
Tabela 21 - Descrição do Caso de Uso do Médico

Caso de Uso	Descrição
Registrar Consultas	Neste caso de uso, o médico efetua o registro de uma consulta agendada, na qual podem ser vinculados possíveis tratamentos e medicamentos, preenchendo observações relativas à consulta.
Consultar Histórico de Pacientes	Permitir que o médico visualize o histórico de consultas do paciente, podendo verificar os detalhes de consultas efetuadas anteriormente.
Manter Ficha de Pacientes	Aqui o médico deve manter atualizada uma ficha de observações do paciente, a qual contém detalhes relacionados à especialidade que o médico atua.

Fonte: Do autor.

E, por último, segue o Caso de Uso do Paciente:

Figura 31 - Caso de Uso do Paciente



Fonte: Do autor.

Abaixo, na Tabela 21, podemos verificar a descrição do Caso de Uso do Paciente:

Tabela 22 - Descrição do Caso de Uso do Paciente

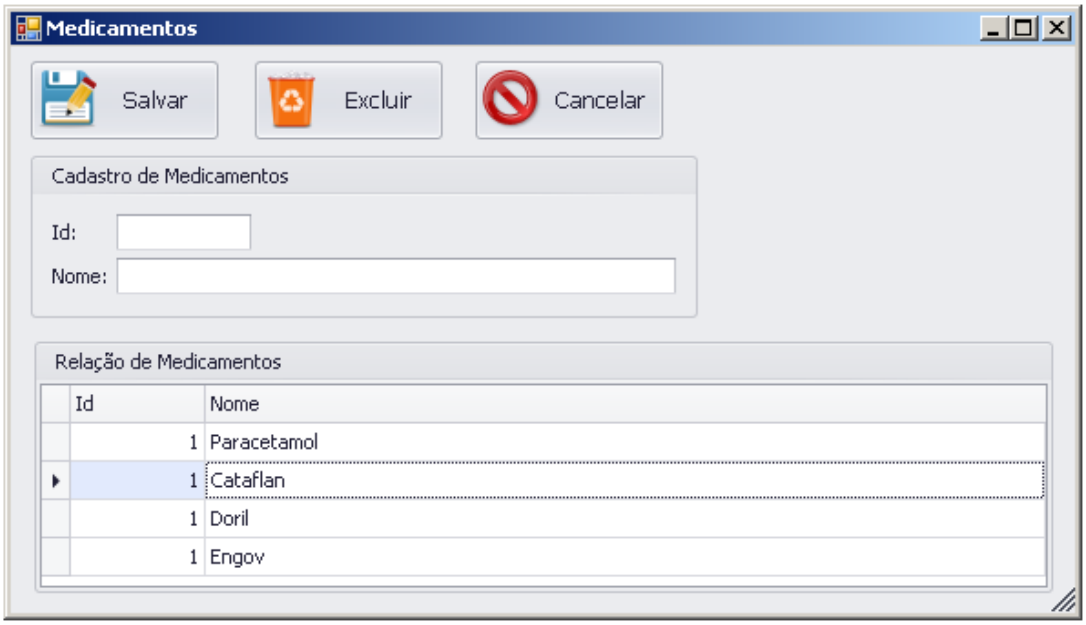
Caso de Uso	Descrição
Consultar Agendamentos	Neste caso de uso, o paciente pode visualizar a relação de suas consultas agendadas, podendo efetuar o cancelamento das mesmas.
Visualizar histórico de Consultas	Aqui, o paciente pode consultar seu histórico de consultas realizadas, podendo verificar os detalhes de cada consulta, como os medicamentos e tratamentos indicados em cada consulta.
Cancelar Consultas	Neste caso de uso, o paciente pode fazer o cancelamento de uma consulta agendada.

Fonte: Do autor.

A partir do detalhamento dos casos de uso, em conjunto da especificação dos requisitos, foram elaboradas as telas do sistema. A seguir são listadas algumas das telas do protótipo.

Na Figura 32, podemos verificar um exemplo de uma tela de cadastro simples, como é o caso da tela de cadastro de medicamentos, tratamentos e especialidades, que contém apenas um campo para o código e um para o nome.

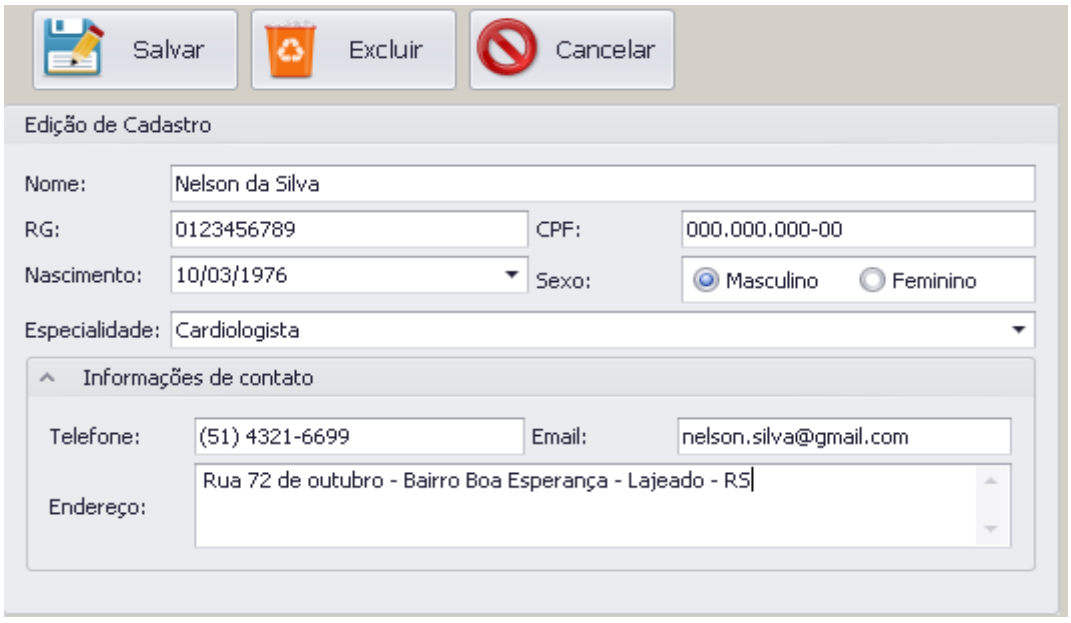
Figura 32 - Cadastro de Medicamentos



Fonte: Do autor.

Já, na Figura 33, podemos observar a tela de cadastro de médicos, que é uma tela um pouco mais elaborada, visto que possui um número maior de informações. A tela de cadastro de pessoas é muito semelhante a de cadastro de médico, visto que a tela de pessoas apenas não tem o campo especialidade, portanto foi utilizado o mesmo modelo.

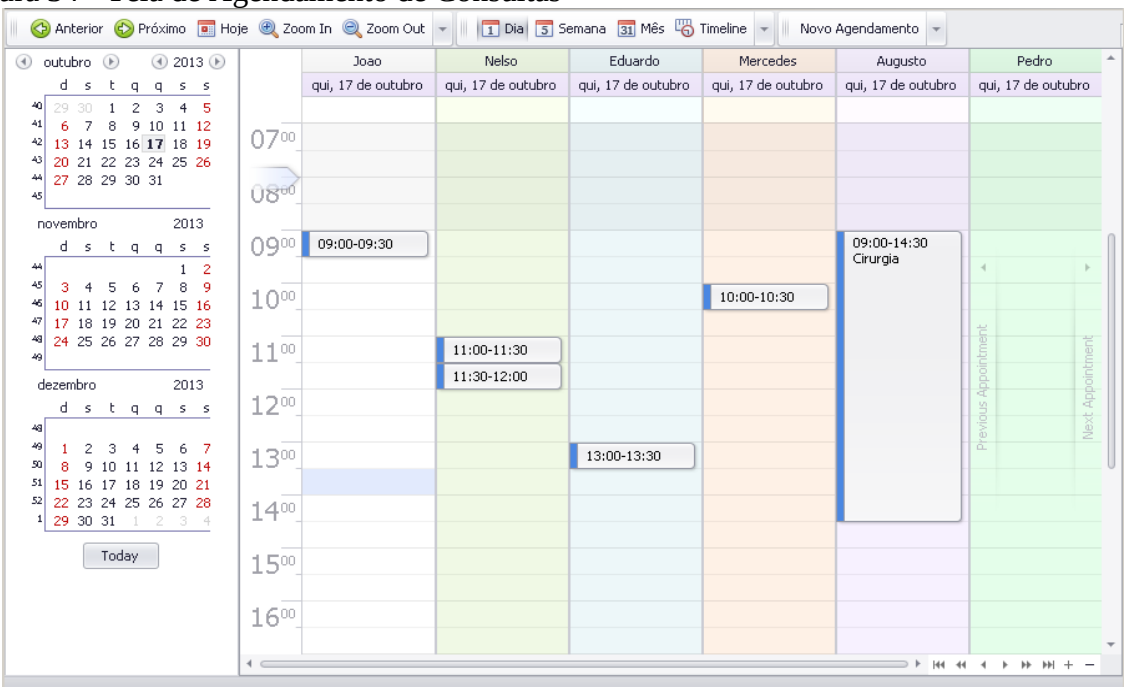
Figura 33 - Cadastro de Médicos



Fonte: Do autor.

A seguir podemos verificar uma das telas mais elaboradas do sistema, que é a tela de agendamentos de consultas. Mediante essa tela é que a secretária fará os agendamentos das consultas para os pacientes que frequentam a clínica, portanto, como podemos observar, a tela apresenta uma espécie de agenda a qual contém a relação dos médicos da clínica e seus respectivos compromissos. Um detalhe importante dessa tela é que a secretária pode alterar o modo de visualização do calendário, podendo ser por dia, semana, mês ou *timeline*.

Figura 34 - Tela de Agendamento de Consultas



Fonte: Do autor.

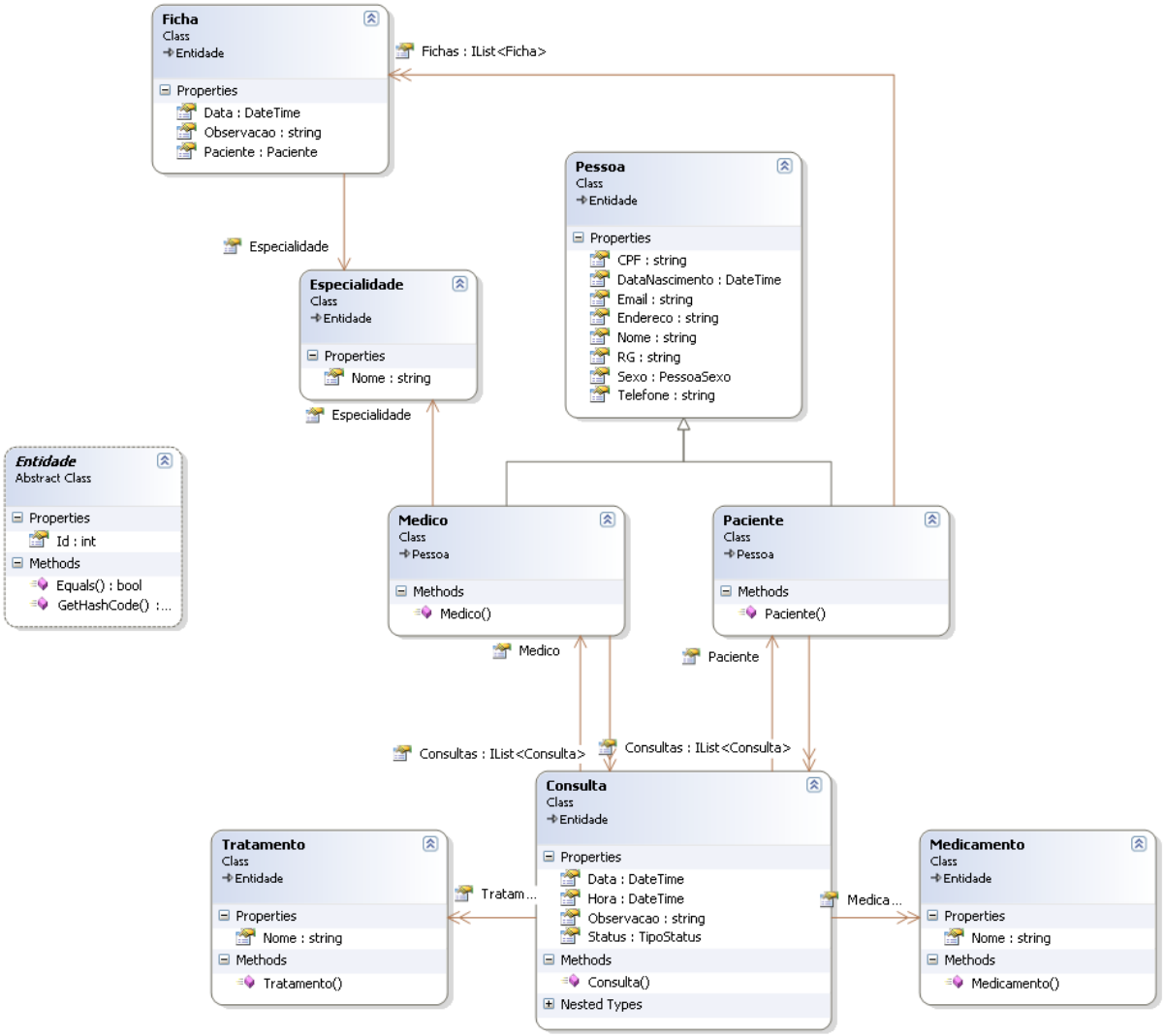
Como podemos observar nas imagens das telas acima, todas elas foram desenvolvidas utilizando os componentes *DevExpress*, pois eles permitem a criação de telas mais elaboradas, pois, sem esses componentes, seria muito difícil criar uma tela de agendamento com essa aparência e dinamicidade, pois, muitos dos detalhes que auxiliam a usabilidade de uma tela como máscaras de preenchimento, validações e formatações customizadas, são facilmente implementadas na aplicação com os componentes da *DevExpress*.

5.1.3 Estrutura

Nesta seção está descrito a forma de como foi implementado o protótipo em relação à arquitetura, tendo como objetivo validar a viabilidade da arquitetura, além de apresentar maiores detalhes de como ocorre a utilização da arquitetura.

A seguir podemos verificar o diagrama de classes, o qual visa a apresentar as entidades que compõem as classes do domínio do protótipo.

Figura 35 - Diagrama de classes do protótipo



Fonte: Do autor.

As classes que compõe o domínio da aplicação se encontram no componente DTO que se encontra na camada Aplicação. É com base nestas classes de domínio da aplicação, que será desenvolvido o restante da aplicação, como persistência das informações ao banco de

dados, validação e aplicação das regras de negócio, processos e apresentação das informações ao usuário.

Para a persistência dos dados, foi utilizado o padrão *Repository*, que visa concentrar todas as consultas que serão efetuadas sobre uma classe do domínio, fornecendo uma interface que possibilite lidar com as informações como se fosse uma coleção de dados. Na Figura 36, podemos ver um exemplo de como foi definido esta interface, que se encontra dentro do componente Dada da camada de Dados.

Figura 36 - Interfaces *IRepository*

```
public interface IRepository<TEntidade> where TEntidade : Dominio.Entidade
{
    void Inserir(TEntidade entidade);
    void Atualizar(TEntidade entidade);
    void Excluir(TEntidade entidade);
    TEntidade BuscarPeloId(int id);
    TEntidade BuscarPelaExp(Expression<Func<TEntidade, bool>> exp);
    IList<TEntidade> BuscarTodos();
}
public interface IMedicoRepository : IRepository<Dominio.Medico>
{
    IList<Dominio.Medico> BuscarPelaEspecialidade(Dominio.Especialidade especialidade);
}
```

Fonte: Do autor.

Como podemos ver na Figura 36, foi criada a interface genérica *IRepository*, que define os métodos padrão para a persistência de todas as classes de domínio do sistema, recebendo por parâmetro a classe sobre a qual são efetuadas as operações. O termo *<TEntidade>* é uma funcionalidade do *C#* chamado de *generics*, que permite tratar tipos de dados de forma genérica.

A fim de exemplificar melhor como se deu a implementação do protótipo, serão apresentados alguns detalhes com a exemplificação das implementações sobre a classe *Medico* do protótipo, as demais classes foram desenvolvidas nos mesmos modelos desta.

Abaixo da interface genérica, é exibida a interface *IMedicoRepository* que tem o objetivo de definir os métodos que estarão disponíveis para manipular a classe Médico. É nesta interface que devem ser definidas as operações especificas sobre a classe *Medico*, pois as demais operações padrão para a persistência dos dados são herdadas pela classe *IRepository*.

A implementação destas interfaces que foram definidas no componente Data, se encontra no componente Data.NH, o qual também se encontra na camada Dados e utiliza os componentes *NHibernate* e *Fluent NHibernate* para efetuar a persistência dos dados do sistema ao banco de dados.

A seguir é apresentado um exemplo da implementação das interfaces mencionadas anteriormente:

Figura 37 - Exemplo da implementação das interfaces *IReposytory*

```
public class Repository<TEntidade> : Data.IRepository<TEntidade> where TEntidade : Dominio.Entidade
{
    private readonly NHibernate.ISession _session;
    public Repository(object session)...
    public virtual void Inserir(TEntidade entidade)...
    public virtual void Atualizar(TEntidade entidade)...
    public virtual void Excluir(TEntidade entidade)...
    public virtual TEntidade BuscarPeloId(int id)...
    public virtual TEntidade BuscarPela(Expression<Func<TEntidade, bool>> exp)...
    public virtual IList<TEntidade> BuscarTodos()...
}
public class MedicoRepository : Repository<Dominio.Medico>, Data.IMedicoRepository
{
    private readonly NHibernate.ISession _session;
    public MedicoRepository(object session)
        : base(session)...
    public IList<Dominio.Medico> BuscarPelaEspecialidade(Dominio.Especialidade especialidade)...
    public IList<Dominio.Medico> BuscarTodosLst()...
}
```

Fonte: Do autor.

Como podemos ver na Figura 37, se encontram a implementação da interface *IRepository*, que implementa as operações básicas de todas as classes, como a de inserir, atualizar, excluir e localizar um registro no banco de dados. A classe *Repository* recebe por parâmetro no seu construtor, a sessão do *NHibernate*, que consiste basicamente no objeto de conexão ao banco de dados, pois é com está sessão que são efetuadas as operações que acessam a base da dados para persistir os objetos do sistema.

Ainda na Figura 37, se encontra a classe *MedicoRepository*, que herda da classe *Repository*, que implementa as operações padrões, e da interface *IMedicoRepository*, que estabelece os métodos específicos para lidar com as informações de Médicos, com o método *BuscarMedicosPelaespecialidade*.

Um ponto importante ressaltar, é que pelo fato de todas as classes de persistência do sistema buscam herdar da classe *Repository*, com isto é possível reutilizar uma boa parte de

código do sistema, evitando escrever um código que ficasse repetido em diversos locais da aplicação.

Na Figura 38, podemos ver um exemplo de como foi efetuado o mapeamento do modelo ER ao modelo orientado a objetos do sistema:

Figura 38 - Exemplo de mapeamento

```
public class MedicamentoMap : ClassMap<Dominio.Medicamento>
{
    public MedicamentoMap()
    {
        Table("Medicamento");
        LazyLoad();
        Id(x => x.Id).Column("Medicamento_id").GeneratedBy.Identity();
        Map(x => x.Nome).Column("MedicamentoNome").Not.Nullable().Length(50);
    }
}
```

Fonte: Do autor.

Na Figura 38, esta exemplificado como foi efetuado o mapeamento da classe Medicamento para a tabela Medicamento do banco de dados, lembrando que este mapeamento foi efetuado utilizando o componente *Fluent NHibernate*. Pois com a utilização do *Fluent NHibernate* com o *NHibernate*, podemos desenvolver uma aplicação que funcione com mais de um tipo de banco de dados, mudando apenas as informações de conexão ao mesmo, sem ser necessário reescrever o código de acesso ao banco de dados.

Já na Figura 39, é apresentado um exemplo da implementação da classe *MedicoServico*, que tem o objetivo de disponibilizar os serviços que serão consumidos pelas aplicações clientes, como as operações que irão acessar a interface *IMedicoRepository*, para persistir as informações, tratando as validações das regras de negócio do sistema.

Figura 39 - Exemplo da implementação de um serviço

```
[ServiceBehavior]
public class MedicoServico : IMedicoServico
{
    Data.IMedicoRepository repositorio;
    Data.IUnitOfWork unitOfWork;
    private Data.IMedicoRepository CriaRepository(object session)
    {
        SimpleInjector.ObjectFactory(1);
        return StructureMap.ObjectFactory.With("session")
                 .EqualTo(session).GetInstance<Data.IMedicoRepository>();
    }
    private void Valida(Dominio.Entidade entidade)...
    public void Salvar(Dominio.Medico entidade)
    {
        Valida(entidade);
        using (unitOfWork = StructureMap.ObjectFactory.GetInstance<Data.IUnitOfWork>())
        {
            repositorio = CriaRepository(unitOfWork.Session);
            if (entidade.Id == 0)
                repositorio.Inserir(entidade);
            else
                repositorio.Atualizar(entidade);
            unitOfWork.Commit();
        }
    }
    public void Excluir(int id)...
    public Dominio.Medico BuscarPeloId(int id)...
    public IList<Dominio.Medico> BuscarTodos()...
}
```

Fonte: Do autor.

Como podemos ver na Figura 39, a classe de *MedicoServico* implementa os métodos que estão definidos na interface *IMedicoServico*, isto se faz necessário para o funcionamento do WCF, pois ele utiliza esta interface, para disponibilizar os serviços aos clientes.

Outro detalhes que é possível visualizar na implementação do serviço, é que a camada Aplicação apenas tem o conhecimento das interfaces que se encontram no componente Data da camada Dados, pois é o componente *Simple Injector* que faz a injeção desta dependência, o que auxilia num maior desacoplamento dos componentes do sistema, facilitando sua manutenção, porque o componentes da camada Aplicação não tem o conhecimento da implementação dos componentes da camada Dados.

Para a apresentação das informações e o controle das interações com o usuário, dividiu-se a camada de Apresentação em quatro componentes: o *Model Facade*, o *Presenter*, o *WinForms* e o *Web*.

No componente *Model Facade*, foram criadas as classes que consomem os serviços que são disponibilizados pela camada Aplicação, na Figura 40 podemos ver um exemplo de como é consumido um serviço:

Figura 40 - Exemplo do consumo de um serviço

```
public class MedicoFacade : IServicoMedicoFacade
{
    private MedicoServicoClient CriaServico()
    {
        string url = "net.tcp://localhost:8081/MedicoServico";
                    MedicoServicoClient cliente = new MedicoServicoClient();
        cliente.Endpoint.Address = new System.ServiceModel.EndpointAddress(new Uri(url));
        cliente.Endpoint.Binding = new System.ServiceModel.NetTcpBinding();
        cliente.Endpoint.Contract.ContractType = typeof(IMedicoServico);

        return cliente;
    }

    #region IServicoGenericoFacade

    public Medico RetornaPeloId(int id)...

    public void Salvar(Medico entidade)...

    public void Excluir(Medico entidade)...

    public IList<Medico> RetornarTodos()...

    #endregion

    public IList<EspecialidadeServico.Especialidade> RetornarEspecialidades()...
}
```

Fonte: Do autor.

No componente *Presenter*, são definidas as interfaces que deve ser implementadas pelos componentes *WinForms* (que implementa as interfaces para a aplicação *Desktop*) e *Web* (que implementa as interfaces para uma aplicação WEB). Desta forma é possível reutilizar o mesmo código que irá fazer o controle das interações dos usuários, indiferentemente da interface que for disponibilizada para os usuários do sistema, permitindo garantir que uma mesma tela do sistema tenha o mesmo comportamento.

Outro ponto é que é o componente *Presenter* que irá fazer toda a intermediação entre os componentes de interação com o usuário, com o componente *Model Facade*, que é responsável por acessar a camada Aplicação, a qual concentra todas as regras de negócio, os processos e a persistência das informações do sistema.

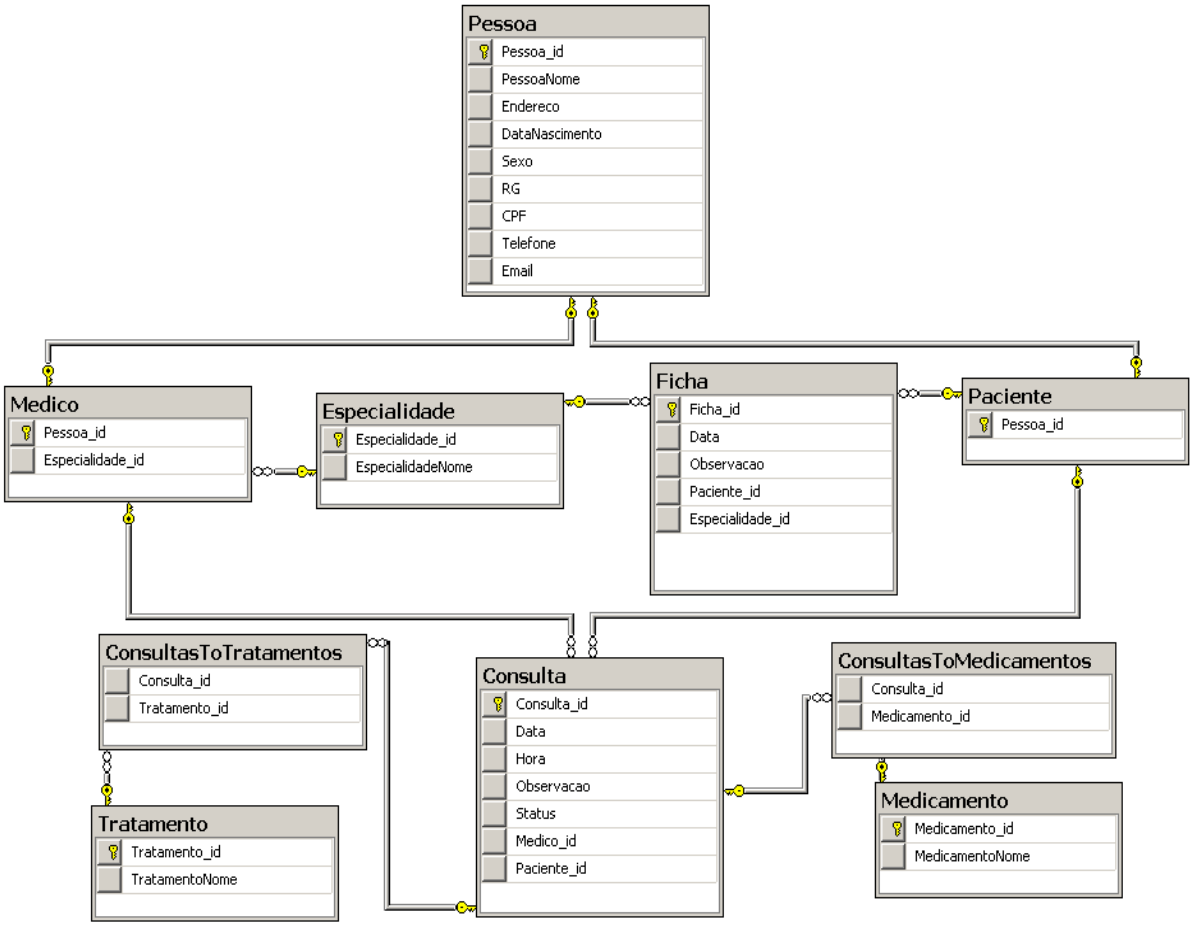
5.1.4 Modelo do Banco de Dados

Segundo Pressman (2010), “usando o diagrama entidade relacionamento como notação fundamental, a modelagem de dados concentra-se na definição de objetos de dados e na maneira pela qual eles se relacionam entre si”.

O diagrama entidade relacionamento (ER), também conhecido como modelo Relacional, foi gerado com base no diagrama de classes do protótipo, pois o *Fluent NHibernate* possibilita gerar o *script* do modelo relacionado a partir dos mapeamento efetuados, sendo assim, o desenvolvedor não necessita se preocupar com a modelagem do modelo ER, basta apenas que elabore a modelagem das classes de domínio e faça o correto mapeamento do mesmo com o *Fluent NHibernate*.

A seguir podemos observar o modelo Relacional que foi criado para armazenar os dados do protótipo elaborado:

Figura 41 - Modelo Relacional do protótipo



Fonte: Do autor.

5.2 Avaliação e resultados

Nesta seção são detalhados os resultados obtidos a partir do estudo e dos testes realizados com o protótipo, buscando apresentar informações importantes obtidas durante a realização do estudo.

5.2.1 Principais Dificuldades

Dentre as principais dificuldades encontradas na elaboração do protótipo, as que mais se destacaram, foram a configuração do WCF e alguns detalhes no uso do *NHibernate*. Pois durante a elaboração do protótipo, deparou-se com um problema na utilização do WCF, que é o componente responsável pela comunicação entre o aplicativo cliente, que consome os serviços, e o aplicativo servidor, que fornece os serviços.

Pelo fato de que o próprio WCF faz o controle da serialização dos objetos que trafegam entre o cliente e o servidor, é necessário lembrar-se de demarcar as classes que fazem parte de um serviço com o atributo *[DataContract]*, assim como demarcar as propriedades destas classes com o atributo *[DataMember]* para que as mesmas sejam serializadas, caso contrário não será possível utilizar estas classes na aplicação cliente que consome o serviço e ou poderá ocorrer algumas exceções na execução da aplicação.

Outro problema percebido com o WCF e o *NHibernate*, foram com as classes do domínio que eram serializadas pelo WCF e que possuíam algum atributo do tipo *Lazy Load* do *NHibernate*, que faz com que aquela propriedade seja recuperada do banco de dados apenas no momento em que for acessada, porém para isto, é necessário que exista alguma seção do *NHibernate* aberta.

Ocorre que algumas das classes do protótipo possuíam uma propriedade *Lazy Load*, e no momento em que o WCF iria fazer a serialização de um objeto desta classe, durante a serialização, o WCF tentava acessar a propriedade *Lazy Load*, porém naquele trecho de código a seção do *NHibernate* já estava fechada, o que causava um erro.

Então para resolver este problema, foi criada uma classe que é chamada antes de retornar um objeto para a camada Aplicação, a fim de resolver estas propriedades e permitindo o WCF serializar corretamente estes objetos.

Outro detalhe importante em relação ao WCF, é a forma como são configurados os protocolos e o formato de comunicação dos serviços, pois é possível efetuar a configuração do WCF tanto via código como via arquivo de configuração XML. A configuração via XML é mais complexa, porém com sua utilização existe a possibilidade de alterar alguma configuração sem ser necessário recompilar a aplicação, já via código, para poder ajustar algum detalhe, como por exemplo, alterar a forma de comunicação de HTTP para TCP, será necessário ajustar o código e recompilar a aplicação, já via arquivo XML, basta ajustar o arquivo e reiniciar a aplicação.

5.2.2 Benefícios encontrados

Com a elaboração do protótipo, foi possível perceber uma série de benefícios no uso da arquitetura proposta, entre as quais se destacam a facilidade de manutenção, a testabilidade e a escalabilidade.

O maior benefício encontrado, foi a facilidade de manutenção da aplicação, pois como a arquitetura proposta visa separar a aplicação em componentes e camadas, estabelecendo contratos entre estes componentes e camadas através de interfaces, os componentes estão mais desacoplados, o que facilita uma alteração no sistema sem que o mesmo tenha algum efeito colateral.

A separação em camadas e componentes, permite também que sejam utilizados testes unitários e testes de integração na aplicação, permitindo ter uma maior abrangência da cobertura destes testes que são facilmente automatizados, o que resulta numa maior qualidade da aplicação, pois é possível identificar rapidamente um erro no sistema.

Outro benefício encontrado, foi a escalabilidade da aplicação com esta arquitetura, pois com a utilização do WCF, é possível implantar o sistema de várias maneiras, de acordo com a necessidade e disponibilidade da aplicação. Pois pode-se implantar o sistema de forma mais simples, através de uma aplicação s*elf-host*, ou publicar o sistema no IIS, no qual nos

permite configurar um conjunto de servidores que irão garantir uma alta disponibilidade e um poder maior de processamento para a aplicação.

5.2.3 Resultados gerais do protótipo

Com o desenvolvimento do protótipo foi possível validar a arquitetura proposta no presente estudo, e comprovar que ela atende plenamente os objetivos estabelecidos, que são de elaborar uma arquitetura para um sistema de gestão que aplique padrões de projetos.

A seguir serão apresentados a forma pela qual foram atendidos todos os requisitos não funcionais estabelecidos:

- **O sistema deve ser desenvolvido em camadas**: a arquitetura proposta visa a criação de um sistema em três camadas, que por sua vez é dividida em vários componentes que compõem o sistema;

- **O sistema deve ter comunicação com mais de um banco de dados**: com a utilização do *NHibernate*, é possível acessar um conjunto de vários bancos de dados, sem ser necessário alterar o sistema;

- **O sistema deve possuir interfaces portáveis para a plataforma Linux**: todos os componentes utilizados na arquitetura podem ser executados no *framework* MONO, permitindo assim que a aplicação seja executada tanto em Windows, MAC ou ambiente Linux;

- **O sistema deve suportar interfaces WEB**: a adoção do padrão de projeto MVP, permite separar a implementação da interface da aplicação, permitindo fazê-la WEB ou Desktop, reutilizando todo o código do controle das interações do usuário e o fluxo das informações;

- **O sistema deve ter uma estrutura que viabilize sua manutenção**: a decomposição do sistema em pequenos componentes, que relacionam-se entre si através de suas interfaces, além de diminuir o acoplamento e aumentar a coesão do sistema, facilita a manutenção do mesmo;

- **O sistema deve ter componentes reusáveis**: o fato de o sistema ser composto por diversos componentes, permite que seus componentes sejam facilmente reutilizados em outras aplicações;

- **O sistema deve fornecer alguns serviços SOAP para integrações**: a utilização do WCF para fornecer toda a infraestrutura pela comunicação do sistema entre a camada de Apresentação e Aplicação, também permite que ou outro aplicativo consumisse estes serviços, indiferente da tecnologia deste aplicativo;

- **O sistema deve ter um bom desempenho, com operações de cadastros não superiores a 15 segundos**: nos testes realizados sobre o protótipo, a abertura de todas as telas de cadastros ficou inferior a dois segundos;

- **O sistema deve permitir automatizar testes**: a separação em camadas e pequenos componentes permite testar facilmente cada parte que compõem o sistema;

- **O sistema deve possibilitar sua personalização**: a utilização de interfaces que estabelecem os contratos entre os componentes do sistema, junto com a utilização da injeção de dependência, permite que um componente seja facilmente trocado por outro, possibilitando a personalização do sistema.

Entre os resultados obtidos, o que mais se destacou foi a facilidade de se ajustar o sistema, sem que este ajuste resulte em efeitos colaterais, pois cada componente busca isolar bem sua responsabilidade, facilitando modificar algo.

A utilização do padrão *Separated Interface*, que visa isolar a interface de uma classe em um componente diferente de sua implementação, permitiu a arquitetura substituir facilmente seus componentes, pois basta desenvolver outro componente que implemente a mesma interface e substituí-lo na aplicação, informando ao *Simple Injetor* que passe a utilizar este novo componente para resolver aquela interface.

Observou-se ainda que a arquitetura proposta no presente trabalho, pode ser considerada um *framework*, visto que a mesma possui as características de um, que consiste em uma abstração de código que é comum em vários projetos de software provendo uma

funcionalidade genérica, com o objetivo de reutilizar o código em demais projetos e a fim de prover um modelo.

Portanto com a elaboração do protótipo provou-se que a arquitetura proposta atendeu os requisitos não funcionais e os objetivos estabelecidos no presente trabalho, resultando em uma arquitetura que possa ser utilizada em sistemas de gestão, permitindo que estes sejam facilmente modificados para evoluírem.

6 CONCLUSÕES

Durante o desenvolvimento do presente trabalho, com a leitura e pesquisa realizada para efetuar o embasamento teórico e referencial bibliográfico, verificou-se que existem diversas ferramentas, métodos e processos que a Engenharia de Software propõe com o objetivo de assegurar a qualidade de um software, permitindo que o mesmo evolua.

Percebeu-se também que um dos pontos cruciais para a evolução e manutenção de um sistema, é justamente o cuidado com a elaboração da arquitetura, porque é ela que vai fornecer toda a base estrutural necessária para a evolução desse sistema, pois, fazendo uma analogia de um software com uma casa, a arquitetura de um software é como o alicerce da casa, sobre o qual é construída toda a casa.

A aplicação de padrões de projeto na arquitetura e no desenvolvimento de um software visa a auxiliar que a arquitetura de um software seja contínua, robusta e bem estruturada, visto que os padrões de projetos são vistos como uma forma de resolver um problema conhecido, utilizando uma solução que foi consagrada e que ainda busca aplicar as melhores práticas e princípios em um software orientado a objetos.

Sendo assim, a arquitetura que foi proposta no presente trabalho, conseguiu atender todos os objetivos e requisitos não funcionais que foram estabelecidos, provando ser um arquitetura que permite que seja desenvolvido um sistema de gestão que possa evoluir facilmente, suportando mais de um tipo de banco de dados, além de possibilitar ter interfaces para WEB e *Desktop*.

Outro ponto importante percebido durante o presente trabalho foi que a elaboração de diagramas que visam demonstrar o funcionamento do sistema, permite ter um maior

entendimento de como deve ser o funcionamento da aplicação, facilitando o entendimento de como o sistema se comporta, a fim de verificar se já não existe algum componente que atenda a uma nova necessidade.

Em relação aos resultados obtidos com o trabalho, acredita-se que a implantação dessa arquitetura na empresa ABC Informática, trará muitos resultados, visto que a arquitetura proposta no presente trabalho proporciona a aplicação da mesma em um conjunto variado de sistemas, sejam eles *Desktop,* WEB ou ainda um misto, já que é possível modificar facilmente a interface da aplicação, sem a necessidade de reescrever toda a lógica da aplicação, além de que, em conjunto ao desacoplamento do banco de dados utilizado, será possível que o cliente opte pela solução de banco de dados que melhor se encaixe, possibilitando reduzir os custo de licenças e até mesmo possibilitar a execução do sistema por completo em um ambiente Linux, garantido uma maior flexibilidade do sistema.

Portanto, o presente trabalho permitiu confirmar que a elaboração de uma arquitetura de software que aplique padrões de projetos, visa a não só aplicar as boas práticas e conceitos da Engenharia de Software, mas também auxilia na garantia de um conjunto de benefícios ao software no qual são usados, a fim de garantir não só a qualidade do mesmo como, também, podendo auxiliar no aumento da produtividade e, consequentemente, na redução dos custos de desenvolvimento e na diminuição do tempo de retorno aos clientes.

REFERÊNCIAS

ABRAN, Alain; MOORE, James W.; BOURQUE, Pierre; DUPUIS, Robert; TRIPP, Leonard L. **Guide to the Software Engineering Body of Knowledge**. California: IEEE, 2004.

FOWLER, Martin. **Patterns of enterprise application architecture**. Boston: Addison-Wesley, 2003.

FOWLER, Martin. **UML Essencial: Um breve Guia para a linguagem-padrão de modelagem de objetos**. 3ª ed. Porto Alegre: Bookman, 2004.

GAMMA, Erich; SALGADO, Luiz A. Meirelles. **Padrões de projeto: soluções reutilizáveis de software orientado a objetos**. Porto Alegre: Bookman, 2008.

GOLDENBERG, Mirian. **A arte de pesquisar: como fazer pesquisas qualitativas em Ciências Sociais**. 4ªed. Rio de Janeiro: Record, 2000.

GUEDES, Gilleanes T. A. **UML 2: uma abordagem prática**. São Paulo: Novatec, 2009.

LARMAN, Craig; BRAGA, Rosana T. Vaccare; MASIERO, Paulo Cesar. **Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo**. Porto Alegre: Bookman, 2008.

OLIVEIRA, Jayr Figueiredo de. **Metodologia para desenvolvimento de projetos de sistemas**. São Paulo: Erica, 2001.

PÁDUA, Elisabete Matallo Marchesini de. **Metodologia da Pesquisa: Abordagem teórico-prática**. 10. ed. Campinas, SP: Papirus, 2004.

PALME, Daniel. **IoC Container Benchmark - Performance comparison**. Disponível em: <http://www.palmmedia.de/Blog/2011/8/30/ioc-container-benchmark-performance-comparison>. Acesso em: 14 ago. 2013.

PFLEEGER, Shari Lawrence; FRANKLIN, Dino. **Engenharia de software: teoria e prática**. São Paulo: Prentice Hall, 2004.

PRESSMAN, Roger S.; PENTADO, Rosângela Ap. D. **Engenharia de software**. Porto Alegre: AMGH, 2010.

SANTOS, Antônio Raimundo dos. **Metodologia científica: a construção do conhecimento**. 2ª ed. Rio de Janeiro: DP&A editora, 1999.

SHALLOWAY, Alan; TROTT, James R. **Explicando padrões e projeto: uma nova perspectiva em projeto orientado a objeto**. Porto Alegre: Bookman, 2004.

SOMMERVILLE, Ian; MELNIKOFF, Selma Shin Shimizu; ARAKAKI, Reginaldo. **Engenharia de software**. São Paulo: Pearson/Addison-Wesley, 2007.

TONSIG, Sérgio Luiz. **Engenharia de software: análise e projeto de sistemas**. Rio de Janeiro: Ciência Moderna Ltda., 2008.

WAINER, Jacques. **Métodos de pesquisa quantitativa e qualitativa para a Ciência da Computação**. In: KOWALTOWSKI, Tomasz; BREITMAN, Karin; organizadores. Atualizações em Informática 2007. Rio de Janeiro: Ed. PUC-Rio; Porto Alegre: Sociedade Brasileira de Computação, 2007.