UNIFACS UNIVERSIDADE SALVADOR
MESTRADO EM SISTEMAS E COMPUTAÇÃO

LUIS HENRIQUE DA HORA NASCIMENTO

INTEGRAÇÃO DE FONTES DE DADOS CONVENCIONAIS E NÃO CONVENCIONAS ATRAVÉS DE UMA ABORDAGEM WOPA (*WRITE ONCE, PERSIST ANYWHERE*)

Salvador
2015

LUIS HENRIQUE DA HORA NASCIMENTO

INTEGRAÇÃO DE FONTES DE DADOS CONVENCIONAIS E NÃO CONVENCIONAS ATRAVÉS DE UMA ABORDAGEM WOPA (*WRITE ONCE, PERSIST ANYWHERE*)

Dissertação apresentada ao Mestrado em Sistemas e Computação da UNIFACS Universidade Salvador, Laureate International Universities como requisito para a obtenção do título de Mestre.
Orientador: Prof. Dr. Jorge Campos.

Salvador
2015

FICHA CATALOGRÁFICA
Elaborada pelo Sistema de Bibliotecas da UNIFACS Universidade Salvador, Laureate International Universities

Nascimento, Luis Henrique da Hora

Integração de fontes de dados convencionais e não convencionas através de uma abordagem WOPA (Write Once, Persist Anywhere) Luis Henrique da Hora Nascimento - Salvador: UNIFACS, 2015.

92 f.: il.

Dissertação Programa de Pós-Graduação em Sistemas e Computação de UNIFACS Universidade Salvador, Laureate International Universities como requisito parcial à obtenção do título de Mestre.

Orientador: Prof. Dr. Jorge Campos.

1. Programação Orientada a Objetos. I. Campos, Jorge de, orient. II. Título.

CDD: 005.133

RESUMO

A maioria dos programas de computador requer algum tipo de mecanismo para armazenamento e recuperação de dados, porém esses mecanismos não costumam possuir integração direta com as linguagens de programação. Essa integração com as fontes de dados requer um grande esforço de programação e isso motivou o desenvolvimento de diversos trabalhos para lidar com o problema. As abordagens mais comuns baseiam-se em mapeamento objeto-relacional, consultas integradas à linguagem e outras soluções que não costumam ser suficientemente flexíveis. Deste modo, o código resultante fica vinculado a um conjunto específico de recursos, requerendo que boa parte dele seja totalmente refeito a cada novo projeto. Esse trabalho apresenta uma solução para abstrair as fontes de dados, permitindo que elas sejam mapeadas através de bibliotecas e configurações fora do código do programa de modo que seja possível a interoperabilidade entre elas. Como resultado, buscou-se obter o aumento da reutilização de código, o desacoplamento em relação às fontes de dados, a redução do retrabalho e o aumento da produtividade, sem limitar a capacidade de expressão do programador.

Palavras-chave: Abstração de Fontes de Dados. Programação Orientada a Objetos. Métodos de Consulta.

ABSTRACT

Most computer programs require some mechanism for storing and retrieving data, but these mechanisms do not use to have direct integration with programming languages. This integration requires too much programming effort, which led to the development of several works dealing with this problem. The most common approaches are based in object-relational mapping, language-integrated query and other solutions that are not flexible enough. Thus, the resulting code is linked to a specific set of resources, requiring the full rebuilding of a big part of it. This paper presents a solution for abstracting data sources, allowing them to be mapped through libraries and settings outside the source code in an interoperable manner. As result, we attempted to achieve the increase of code reuse, the decoupling of data sources, the reducing of rework and the increasing of productivity, without limiting the expression capability of the programmer.

Keywords: Data Sources Abstraction. Object-Oriented Programming. Query Methods.

LISTA DE FIGURAS

Figura 1 - Modelo abstrato do fluxo de dados em um aplicativo 11
Figura 2 - Modelo de implementação do fluxo de dados em um aplicativo 13
Figura 3 - Principais contextos do ambiente de execução 21
Figura 4 - Modelo MVC 26
Figura 5 - Modelo básico de persistência do Active Records 29
Figura 6 - Modelo de persistência do Active Records com seleção de fonte de dados 29
Figura 7 - Exemplo de comunicação via Bluetooth no sistema Android 34
Figura 8 - Interação entre programa e fonte de dados no modelo DOD 38
Figura 9 - Arquitetura de um sistema baseado no FrameDOD 38
Figura 10 - Arquitetura do FrameDOD 44
Figura 11 - Interação entre programa e fonte de dados no modelo DOD 46
Figura 12 - Arquitetura do FrameDOD (reprodução da Figura 10) 46
Figura 13 - Classe Paciente 47
Figura 14 - Exemplo de mapeamento HOML detalhado para o modelo Paciente 48
Figura 15 - Exemplo de mapeamento HOML simplificado para o modelo Paciente 48
Figura 16 - Exemplo de uso de interface de dados generalizada 49
Figura 17 - Exemplo de uso de interface de dados especializada 49
Figura 18 - AST gerada pela operação SELECT relativa à entidade PACIENTE 50
Figura 19 - Exemplo genérico de documento HOML 53
Figura 20 - Declaração de bibliotecas em HOML 53
Figura 21 - Declaração de drivers em HOML 54
Figura 22 - Exemplo de mapeamento de fontes de dados em HOML 54
Figura 23 - Declaração de grupos de fontes de dados em HOML 55
Figura 24 - Declaração de entidades em HOML 56
Figura 25 - Exemplo mapeamento de entidades em HOML 57
Figura 26 - Definição de operações personalizadas em HOML 57
Figura 27 - Definição de operações personalizadas em HOML 62
Figura 28 - Estrutura básica do arquivo “app.config” 65
Figura 29 - Configuração básica do Entity Framework (gerada automaticamente) 66
Figura 30 - Configuração básica do NHibernate. 66
Figura 31 - Configuração básica do FrameDOD com arquivo HOML separado 67

Figura 32 - Configuração do FrameDOD sem arquivo HOML separado 67
Figura 33 - Mapeamento do banco de dados no Entity Framework (Gerado automaticamente)68
Figura 34 - Mapeamento do banco de dados no FrameDOD 69
Figura 35 - Mapeamento da classe Paciente no NHibernate 70
Figura 36 - Operação Insert 71
Figura 37 - Operação Select 71
Figura 38 - Operação Update 72
Figura 39 - Operação Delete 72

LISTA DE ABREVIATURAS E SIGLAS

AC	Abstração de consultas
ASCII	*American Standards Committee for Information Interchange*
AST	*Abstract Syntax Tree*
CAP	Controle abstrato da persistência
CBD	Controle das bases de dados
CRUD	*Create Read Update Delete*
HOML	*Hierarchical Object Mapping Language*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
ICU	Interface com o usuário
IDE	*Integrated Development Environment*
ISP	*Java Server Pages*
LINQ	*Language Integrated Query*
MD	Modelo de dados
MDOD	Modelo de desenvolvimento orientado a dados
MOR	Mapeamento Objeto-Relacional
MVC	*Model View controller*
ODBC	*Open Data Base Connectivity*
OO	Orientação a objetos/Orientado(s) a objetos
RN	Regra de negócio
SGBD	Sistema Gerenciador de Bancos de Dados
SO	Sistema Operacional
SQL	*Structured Query language*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
UML	*Unified Modeling Language*
VSD	Visão sobre os dados
XML	*Extensible Markup Language*

SUMÁRIO

1	**INTRODUÇÃO**	**17**
1.1	JUSTIFICATIVA	12
1.2	OBJETIVOS	14
1.3	METODOLOGIA	15
1.4	LIMITAÇÃO DO ESCOPO	18
1.5	AUDIÊNCIA	18
1.6	ESTRUTURA DO TRABALHO	18
2	**TRABALHOS RELACIONADOS**	**19**
2.1	CONTEXTUALIZAÇÃO GLOBAL DOS TRABALHOS	21
2.2	MODELOS E PADRÕES DE PROJETOS	25
2.3	*FRAMEWORKS*	27
2.4	FONTES DE DADOS NÃO CONVENCIONAIS	32
2.5	ANÁLISE DOS TRABALHOS RELACIONADOS	35
3	**EVOLUÇÃO DO DESENVOLVIMENTO ORIENTADO A DADOS: DO MODELO AO *FRAMEWORK***	**38**
3.1	TRATAMENTO DE FONTES DE DADOS NÃO CONVENCIONAIS	40
3.2	O LIMITE ENTRE A HETEROGENEIDADE E A UNIFORMIDADE	41
3.3	INTRODUÇÃO DA HOML	41
3.4	O FRAMEDOD	43
3.5	EXTENSIBILIDADE	51
3.6	ESTRUTURA DA HOML	52
3.7	ESTRUTURA SINTÁTICO-SEMÂNTICA	58
3.8	TESTES DE CAMPO	59
4	**ANÁLISE COMPARATIVA: *ENTITY FRAMEWORK* X FRAMEDOD X NHIBERNATE**	**64**
4.1	CONFIGURAÇÃO BÁSICA DOS *FRAMEWORKS*	65
4.2	EXECUÇÃO DAS OPERAÇÕES SOBRE OS DADOS	70
4.3	OUTRAS CARACTERÍSTICAS	73
5	**CONCLUSÕES**	**76**
	REFERÊNCIAS	78
	APÊNDICE A – FORMAS DE CLASSIFICAÇÃO DAS FONTES DE DADOS	83

APÊNDICE B – VANTAGENS E DESVANTAGENS DAS FONTES DE DADOS84
APÊNDICE C – INTERFACES QUE PRECISAM SER IMPLEMENTADAS POR UM DRIVER 86

1 INTRODUÇÃO

Durante o processo de desenvolvimento de *software*, diversas situações requerem a aquisição, o armazenamento persistente ou a troca de dados. A criação do código necessário para dar suporte a esses recursos consome um tempo significativo e as fontes de dados manipuladas são as mais diversas. Igualmente diversas são as formas de comunicação e as representações utilizadas para lidar com essas fontes. O programador precisa lidar com essa diversidade, adaptando o seu código às ferramentas disponíveis, conforme as necessidades do projeto e a disponibilidade de tempo. Para isso, são utilizados de padrões, técnicas, *frameworks*, *Integrated Development Environments* (IDEs) e outros recursos que facilitam o desenvolvimento, aprimoram a integração da equipe e reduzem o tempo de entrega dos artefatos do projeto.

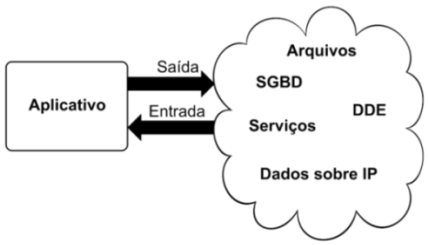
Para cada tipo de recurso, conforme a sua natureza, é necessário utilizar uma técnica, ferramenta ou artefato de *software* diferente. Isso dificulta a uniformização do código e aumenta a complexidade do projeto, reduzindo a produtividade. Uma vez que o programador precisa cumprir prazos restritos, não há tempo para que ele mesmo desenvolva uma solução abrangente. Isso força o uso de soluções prontas ou a adoção de medidas pontuais para resolver somente os problemas imediatos. Dentre as formas de aquisição e/ou persistência de dados mais comuns, podemos citar: manipulação direta de arquivos, Sistemas Gerenciadores de Bancos de Dados (SGBDs), Webservices, *Dynamic Data Exchange* (DDE), entre outros. Cada forma de persistência possui suas próprias particularidades, mas todas trabalham com a mesma matéria-prima: os dados. Neste contexto, não importa qual seja a origem, o destino ou a aplicação destes dados, mas o fato de que a forma como o programador os manipula costuma diferir da forma como são persistidos ou adquiridos.

A aquisição de dados, normalmente ocorre em sua forma bruta e não estruturada, seja como vetor de bytes (dados binários) ou como tipos de dados primitivos simples (texto, número, data, entre outros). Depois de adquiridos, esses dados são organizados conforme a necessidade do programador e o domínio de aplicação, porém precisam ser transformados para possibilitar a sua persistência de modo organizado. Isso significa que a forma como os dados são adquiridos varia com a sua origem e a forma como os dados são persistidos depende do meio de persistência.

O mapeamento objeto-relacional é uma das soluções mais utilizadas para traduzir os

dados provenientes de bancos de dados para as estruturas utilizadas em memória (Objetos), porém limita-se a um domínio de aplicação específico. Essa técnica consiste na construção de objetos (modelos criados pelo programador) a partir de dados obtidos de um banco de dados relacional. Para isso, são utilizadas algumas regras que permitem compatibilizar as duas representações, facilitando o intercâmbio de dados entre elas. Isso significa que o conhecimento prévio sobre qual fonte de dados será utilizada determina as decisões de projeto e orienta como o código deverá ser construído. Essa abordagem acaba forçando o programador a desviar o foco do seu objetivo principal, demandando uma grande quantidade de tempo desenvolvendo o código necessário para lidar com a persistência dos dados antes de poder tratar a regra de negócio.

Figura 1 - Modelo abstrato do fluxo de dados em um aplicativo



Fonte: autoria própria (2015)

Abstraindo-se a complexidade das rotinas necessárias para lidar com as diversas fontes de dados, o fluxo de entrada e saída de um aplicativo convencional pode ser representado conforme ilustrado na Figura 1. O programa geralmente obtém dados de algum elemento externo (entrada), depois esses dados são processados e o resultado é enviado de volta à origem ou para um segundo elemento externo (saída). Idealmente, com base nesse raciocínio, quando um programa interage com uma fonte de dados qualquer, nenhum conhecimento mais profundo sobre essa fonte deveria ser necessário. Esse conhecimento, entretanto, acaba sendo totalmente ou parcialmente embutido em alguma parte do código, ditando como o código do programa deve ser escrito ou determinando as decisões de projeto.

A proposta deste trabalho consiste em aproximar a forma como o programa é desenvolvido da forma como o fluxo de informação deveria acontecer, permitindo ao programador ter acesso aos dados de forma totalmente desacoplada da sua fonte. Para cumprir esse propósito, foi necessária a criação de um modelo de desenvolvimento no qual os dados determinam como o código deverá ser escrito, independentemente da sua fonte. Esse modelo foi utilizado como referência para a criação de um *framework* e uma linguagem que atuam interpostos entre o *software* e os recursos. Com isso, buscou-se criar uma camada de

abstração para simplificar o acesso aos dados e a integração entre as fontes de dados. Desse modo, todo o conhecimento relativo à natureza das fontes de dados fica fora do código.

1.1 JUSTIFICATIVA

Segundo Sommerville (2011), os principais desafios da engenharia de *software* são três. O primeiro é a heterogeneidade dos recursos, ambientes, linguagens e outros elementos com os quais o programador e o *software* precisam lidar. O segundo é grande demanda pela redução no tempo de entrega do *software*. O terceiro é a construção de *softwares* confiáveis, com baixos índices de erros ou defeitos.

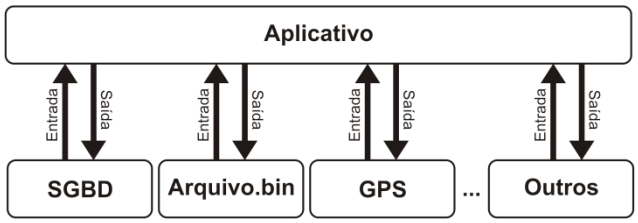
O problema conhecido pela engenharia de *software* como “desafio da heterogeneidade crescente” (SOMMERVILLE, 2011) tem consequências diretas no tempo de entrega dos artefatos produzidos e na qualidade do *software* criado. Isso ocorre, pois o tempo é um recurso limitado e cada requisito adicional consome uma parte desse tempo ou demanda mais tempo para a execução do projeto. Para mitigar o problema, torna-se necessária a adoção de medidas que podem comprometer a qualidade ou atrasar a entrega, elevando o custo.

O crescimento da heterogeneidade torna o processo de desenvolvimento mais lento e o tempo disponível mais escasso. Quanto maior o número de recursos que precisam ser tratados, maior é a quantidade de código a ser escrito para cada um deles. Por esse motivo fica mais difícil cumprir os requisitos de qualidade definidos pelo cliente dentro do prazo especificado. Em casos extremos o descumprimento destes requisitos pode levar à aplicação de multas e outras penalidades contratuais, causando sérios prejuízos ao executante e ao cliente. Dentre os recursos com maior heterogeneidade, encontram-se as fontes de dados, que podem ser Sistemas Gerenciadores de Bancos de Dados (SGBDs), arquivos binários, dispositivos, entre outros mecanismos. Esse recurso é particularmente crítico, já que todo *software* trabalha com aquisição, processamento, troca e/ou persistência de dados. Essas atividades são essenciais, pois o *software* pode lidar paralelamente com inúmeras fontes de dados distintas e com uma quantidade igualmente grande de formas para interagir com elas. Isso aumenta a complexidade do *software* e dificulta o dimensionamento do tempo necessário o seu desenvolvimento.

Quando observamos o funcionamento de um *software* de um ponto de vista abstrato, percebemos que suas interações ocorrem conforme o modelo de fluxo de dados apresentado na Figura 1. Isso significa que o *software* obtém dados de uma fonte externa, transforma esses

dados e os envia transformados para outra fonte externa, que pode ou não ser a mesma de onde os dados se originaram. Quando um programa é desenvolvido, o programador costuma lidar com esse mesmo fluxo, manipulando diretamente cada fonte de dados ou abstraindo somente aquelas que possuem mecanismos de acesso parecidos entre si. Essa situação é ilustrada na Figura 2, onde um mesmo aplicativo precisa utilizar diversos canais de comunicação independentes com códigos especializados para tratar cada um deles. Nessa ilustração, cada canal de entrada e saída representa uma interação com uma fonte de dados diferente e, para cada uma delas, um código especializado a ser escrito.

Figura 2 - Modelo de implementação do fluxo de dados em um aplicativo



Fonte: autoria própria (2015)

Para lidar com o contexto apresentado, o programador precisa utilizar de bom senso, boas práticas e criatividade para solucionar problemas periféricos, desviando o foco do objetivo principal e demandando tempo e recursos que são limitados. É comum que o programa inclua rotinas especializadas para cada fonte de dados distinta, conforme o exemplo da Figura 2, onde temos um SGBD, um arquivo binário e um GPS, cada um com suas próprias rotinas especializadas de entrada e saída de dados. Diante deste cenário, a abstração das fontes de dados permite aproximar o modelo de implementação (Figura 2) do modelo idealizado (Figura 1), reduzindo a quantidade de código a ser escrito e proporcionando o aumento da produtividade através da reutilização de código. Uma vez que as fontes de dados sejam tratadas de forma homogênea, torna-se mais simples a sua manipulação e a integração entre elas, o que viabiliza e justifica o investimento nesta abordagem.

O modelo idealizado, descrito na Figura 1, pode ser comparado com o funcionamento de um sistema operacional (SILBERSCHATZ, 2010): as fontes de dados seriam tratadas como os dispositivos e a interação com elas seria uniformizada através dos *drivers*. Seguindo esse modelo, a aquisição dos dados ocorre através de peças de *software* genéricas e intercambiáveis denominadas *drivers*, sem qualquer conhecimento especializado sobre a natureza das fontes de dados mapeadas. As configurações que permitem essa interação seriam tratadas de forma análoga ao que é feito pelo utilitário de configuração do *Basic Input Output System* (BIOS) e os *drivers* reconheceriam e utilizariam essas configurações. As semelhanças,

no entanto, encerram-se neste ponto, uma vez que o utilitário de configuração proposto consiste em um *Software* em vez de um *Firmware*. Além disso, as configurações seriam acessíveis ao programador e mantidas em arquivos estruturados facilmente compreensíveis (baseados em XML), ainda que esses arquivos não precisem ser manipulados manualmente.

1.2 OBJETIVOS

A proposta apresentada nesta dissertação consiste na abstração de fontes de dados heterogêneas convencionais (bancos de dados) e não convencionais (dispositivos, fluxos de dados e outros) através de uma abordagem *Write Once Persist Anyware* (WOPA), buscando reduzir o tempo de desenvolvimento através da reutilização intensiva de código.

O objetivo principal deste trabalho é a apresentação de um *Framework* e uma linguagem que atuem juntos no processo de abstração de fontes de dados, permitindo que o programador interaja com esses recursos de maneira uniforme, com pouco código especializado, alto grau de reutilização e interoperabilidade. A solução proposta busca facilitar a aquisição, a transferência e a persistência dos dados, mesmo trabalhando com fontes distintas e diferentes entre si. Para alcançar o nível de transparência desejado, alguns objetivos específicos devem ser atingidos:

Definição de uma linguagem de consulta independente de fonte de dados – Uma vez que cada fonte de dados possui características próprias e algumas delas nem mesmo possuem uma linguagem de consulta, faz-se necessária a criação de uma forma abstrata e flexível para definição das operações sobre os dados;

Independência de localização – O local onde os dados estão, ou para onde os dados devem ir, pode ser uma URL, um serviço, um endereço IP, uma instância de aplicativo, um arquivo, entre outras possibilidades. Quando essa informação permeia o código, cria-se um ponto de heterogeneidade que o programador precisa tratar. Por esse motivo, nenhuma informação sobre como acessar os arquivos de dados ou fontes de dados deve existir no código do programa, pois isso requer de criação de código especializado, aumentando o tempo de desenvolvimento. Todas as informações de localização, conexão e acesso devem ser parametrizadas fora do código;

Independência de ambiente operacional – Muitas ferramentas de programação modernas são capazes de criar programas que operam em múltiplas plataformas, sejam hardwares diferentes, sistemas operacionais diferentes ou até mesmo máquinas virtuais

diferentes. Nenhuma peculiaridade do sistema operacional ou do ambiente de execução deve requerer modificações no código do programa, pois isso comprometeria a homogeneidade que o *Framework* proposto busca prover;

Independência de fonte de dados – Um mesmo programa pode lidar com múltiplas fontes de dados cujos métodos de acesso e de comunicação podem ser radicalmente diferentes. A existência dessas peculiaridades não deve implicar na criação de código especializado, de modo que qualquer recurso não suportado por uma fonte de dados específica deve ser provido de forma transparente;

Independência sintático-semântica – Diferentes fontes de dados podem possuir suas próprias linguagens de consulta, variações de uma mesma linguagem ou, no pior caso, nem mesmo possuem algo equivalente a uma linguagem de consulta. Isso significa que as mesmas instruções podem possuir semânticas diferentes para uma fonte de dados diferentes ou podem não ter semântica para algumas dessas fontes. Mesmo quando a semântica coincide, a sintaxe pode ser diferente. Para lidar com esse problema, é necessária a criação de uma padronização que permita a interação com as fontes de dados sem que suas especificidades permeiem o código do programa principal;

Extensibilidade – Quando um *framework* é utilizado como interface entre o programa principal e um recurso externo, é comum que o criador desse *framework* defina um conjunto extenso de funcionalidades para lidar com um grande número de situações. Apesar disso, não é possível prever todos os casos e o programador pode ter necessidades específicas que não foram contempladas na criação do *framework*. Para garantir a flexibilidade e a capacidade de expressão, o *framework* deverá dispor de recursos que permitam ao programador a criação de suas próprias extensões, agregando novos recursos ao conjunto existente.

Os objetivos apresentados foram motivados por necessidades específicas identificadas ao longo da pesquisa bibliográfica e dos experimentos executados durante a criação do *framework* proposto. Esses objetivos e os problemas que os inspiraram são abordados em detalhes no capítulo 3, onde também são apresentadas algumas situações que ilustram como esses problemas influenciaram o desenvolvimento do *framework*.

1.3 METODOLOGIA

Para atingir os objetivos propostos, foi utilizada uma abordagem baseada na observação dos problemas, coleta e análise de dados, definição de hipóteses, experimentação

de prováveis soluções, verificação e validação dos resultados obtidos. Esse processo foi executado de forma iterativa e sistemática visando-se encontrar a solução mais abrangente e flexível possível dentro das limitações de tempo e de recursos.

O primeiro passo consistiu na observação de como os *software*s costumam ser desenvolvidos. Isso permitiu a identificação da recorrência de problemas relacionados à aquisição, troca e persistência de dados. Essas rotinas constituem um ponto de retrabalho, pois lidam com aspectos específicos de cada caso e precisam ser reescritas para cada novo projeto. Os resultados obtidos nesta etapa levaram a uma pesquisa bibliográfica para identificar quais soluções estavam disponíveis para tratar esses problemas e quais abordagens eram utilizadas.

Através da revisão da literatura e da execução de experimentos, diversas soluções diferentes foram encontradas, tanto acadêmicas quanto comerciais. Cada solução encontrada abordava diversos aspectos do problema, desde a interface com o usuário até a persistência dos dados, porém de forma isolada ou especializada. Muitos trabalhos tratam do mapeamento objeto-relacional e outras formas de interação com bancos de dados e tecnologias correlatas, porém deixam de lado as fontes de dados mais simples como arquivos binários, fluxos de dados e comunicação entre processos.

Os trabalhos identificados através da revisão da literatura foram analisados e comparados para a identificação e classificação dos mesmos, permitindo a compilação dos apêndices desta dissertação. Os resultados obtidos serviram de referência para comparação dos trabalhos e para a escolha da abordagem mais apropriada para os objetivos propostos.

Ao fim deste estudo foi identificada uma carência em relação ao tratamento de fontes de dados heterogêneas, em especial àquelas não baseadas no modelo relacional. Igualmente deficiente é a disponibilidade de meios para facilitar a interoperabilidade entre as diversas fontes de dados existentes.

A partir da análise das informações obtidas e com base na revisão da literatura, foi identificado que o tempo dedicado às rotinas de aquisição, armazenamento e recuperação de dados compreende uma parte significativa do tempo total de desenvolvimento. Muito desse trabalho precisa ser refeito a cada novo projeto, pois cada fonte de dados diferente tem suas próprias peculiaridades e cada projeto requer um subconjunto específico dos recursos de cada uma dessas fontes.

Nesse contexto, identificou-se que a abstração das fontes de dados pode ser utilizada como ferramenta para redução da complexidade, permitindo a diminuição retrabalho e o aumento da produtividade. Essa constatação permitiu definir um rumo para a pesquisa, direcionando o foco para os trabalhos especificamente voltados para as diversas formas de

abstração existentes, porém ainda seria necessário identificar qual abordagem teria a melhor aceitação e a menor curva de aprendizado.

Uma vez escolhida a abordagem a ser seguida, surgiu a necessidade de criar uma forma para permitir ao programador expressar as operações sobre os dados e garantir a abstração, minimizando a criação rotinas especializadas de programação.

Através de um processo iterativo, diversas abordagens foram analisadas e testadas visando encontrar aquelas que melhor se adequassem à proposta dessa dissertação. Nesse estudo, foram consideradas a eficácia, a eficiência e a complexidade do uso de cada uma das soluções. Para isso foram estabelecidos critérios qualitativos e quantitativos que permitiram a classificação das soluções e a escolha da abordagem mais apropriada.

Após a escolha da abordagem a ser seguida, vários problemas precisaram ser solucionados, o que levou à necessidade de separação entre as expressões de consulta e as estruturas responsáveis pelo seu processamento. Isso demandou a utilização de uma linguagem específica para expressar as operações, cumprindo dois requisitos básicos: minimizar a possível rejeição pelos programadores e minimizar a curva de aprendizado necessária à sua utilização. A escolha natural recaiu sobre a Structured Query Language (SQL), por estar presente no dia-a-dia da maioria dos programadores e ser amplamente conhecida e familiar, porém essa decisão mostrou-se inapropriada. A SQL, no entanto, possui diversas variações, de modo que o dialeto com o qual o programador está familiarizado acabaria diferindo daquele adotado, gerando confusão. A estrutura da SQL também torna complexo o processamento das expressões de consulta, requerendo muito estorço do programador para estender a linguagem ou criar processadores de linguagem personalizados.

Uma alternativa à criação de um novo dialeto de SQL foi parcialmente inspirada por alguns dos trabalhos estudados. A solução criada teve como propósito ir além do simples mapeamento, fundindo as linguagens SQL e XML em uma nova linguagem, aproveitando o conhecimento prévio do programador e visando expressar tanto as operações sobre os dados quanto as definições do mapeamento objeto-relacional. Com a criação dessa estrutura, buscou-se simplificar a criação de processadores de linguagem e até mesmo a criação de ferramentas para construir e manter as expressões, propiciando a substituição de boa parte do trabalho manual pela automação parcial ou total.

A etapa final consistiu na aplicação da solução criada em sistemas já existentes, para que fosse possível verificar sua adequação aos objetivos almejados, permitindo a identificação de problemas e a lapidação da solução.

1.4 LIMITAÇÃO DO ESCOPO

O método de abstração proposto neste trabalho tem como base o modelo relacional e o paradigma de programação orientado a objetos. Para isso, é utilizada uma representação pseudo-relacional das fontes de dados a serem manipuladas. Isso significa que a solução proposta adequa-se somente aos casos nos quais a representação dos dados na forma relacional seja possível. Essa limitação não restringe a solução às fontes de dados nativamente relacionais, porém exclui aquelas cuja representação relacional seja impossível.

1.5 AUDIÊNCIA

Esta dissertação tem como principais públicos alvos os estudantes, pesquisadores, programadores, engenheiros de *software* e outros profissionais de computação e acadêmicos interessados em engenharia de *software*, mecanismos para abstração de fontes de dados e desenvolvimento de *software*. Neste trabalho foram aplicadas diversas técnicas de pesquisa, análise, projeto e desenvolvimento de *software* e são abordadas diversas soluções correlatas nessas áreas e em áreas afins.

1.6 ESTRUTURA DO TRABALHO

O restante desta dissertação está dividido em 4 partes. O capítulo 2 trata dos trabalhos relacionados e apresenta uma breve discussão sobre as abordagens utilizadas, assim como vantagens e desvantagens de cada uma delas. O capítulo 3 descreve a evolução do trabalho até o seu estado atual, apresenta a arquitetura do *framework* proposto e descreve o seu funcionamento, abordando os detalhes da linguagem de mapeamento proposta e apresentando exemplos relevantes de utilização. Ao final do capítulo 3 também são apresentados alguns testes de campo que foram executados ao longo do desenvolvimento do trabalho e influenciaram fortemente na sua evolução. O capítulo 4 apresenta alguns exemplos práticos, comparando o FrameDOD, o Entity Framework e o NHibernate. O capítulo 5 apresenta a análise dos resultados obtidos, as conclusões e propostas para trabalhos futuros.

2 TRABALHOS RELACIONADOS

As atividades de aquisição, persistência e troca de dados são recorrentes e possuem grande importância em quase todos os *software*s. A diversidade das fontes de dados existentes é enorme, porém as diferenças entre elas demandam a criação de rotinas especializadas que consomem tempo e elevam o custo do projeto. Os problemas encontrados incluem a falta de padronização, o longo tempo de aprendizado, a documentação deficiente, as peculiaridades de cada projeto e as limitações de custo e prazo (SOMMERVILLE, 2011).

Durante a pesquisa bibliográfica, as fontes de dados mais comuns foram identificadas com base em diversos trabalhos estudados. Isso resultou na compilação do Quadro 1 que serviu de orientação para o restante da pesquisa. Este quadro reúne as principais fontes de dados de forma categorizada, apresentando exemplos relevantes de cada categoria e permitindo uma visualização macro do problema a ser tratado.

Quadro 1: Classificação das principais fontes de dados quanto à categoria

Categoria	**Fontes de dados**
Arquivos Estruturados	• CSV • HTML • XML • XSL
Arquivos simples	• Arquivos binários • Arquivos de acesso aleatório • Arquivos sequenciais
Bancos de dados	• Access • DBase • Paradox
Comunicação entre processos	• Broadcast • Memória compartilhada • Multicast • NFC • Unicast
Dispositivos de armazenamento	• Disco rígido • Flash Drives • Unidades SSD
Dispositivos de entrada e saída	• Centrais telefônicas • GPS • Sensores • Servomecanismos
SGBDs	• MySQL • Oracle • PostgreSQL • SQL Server
Serviços	• E-Mail • SMS • WebServices
Frameworks	• Hibernate • Active Records • LINQ
Outros	• Sistema de arquivos

Fonte: autoria própria (2015)

No APÊNDICE pode ser encontrada uma versão estendida da classificação das

fontes de dados e no APÊNDICE é feita uma análise das vantagens e desvantagens de cada uma. O progresso desse estudo permitiu a analise dos trabalhos que lidam com as fontes de dados encontradas e levou à compilação do Quadro 2.

Quadro 2: Classificação dos principais trabalhos quanto às características

Característica / Trabalho	Plataformas de Desenvolvimento	Modelos e padrões de Desenvolvimento	Mapeamento Objeto/Relacional	Serviços	Integração entre bases de dados	Ferramentas de Desenvolvimento	Outras iniciativas
Abordagem de Gateway (MOLZ, 1999)	X		X				
Active Records (TORRES et al, 2014)		X					
ADO.Net (CHAUDHURI et al, 2014)	X		X				
Entity Framework (GRIFFIN, 2013)			X				
Agora (MANOLESCU 2000)			X		X		
Bancos de dados federados (SHETH, LARSON 1990)	X			X	X		
DB2 Universal Data Joiner (VENKATARAMAN et al., 1998)	X			X	X		
Denodo (PAN et al, 2002)	X				X		
DevExpress (KIMMEL, 2010)			X				
GeraConstrutorSQL (VIDAL, 2005)			X				
GignoMDA (RICHLY, 2006)							X
Hibernate (BAUER, 2005)	X		X				
iBATIS/MyBatis (BEGIN, 2006)	X		X				
JDBC (CHAUDHURI et al, 2014)	X						
Le Select (BOUGANIM et al., 2001)2	X				X		
LINQ (MARGUERIE, 2008)	X		X				
LLBLGen Pro[1]			X				
Model1 (HUSTED, 2003)		X					
Model2 (HUSTED, 2003)		X					
Modelo MVC (HUSTED, 2003) (PAZ, 2013)		X					
ODB[2]	X		X				
ODBC (MICROSOFT, 2010)	X		X				
OdeFS (CONSTANTIN et al., 2012)							X
Oracle (LONEY, 2008)	X			X	X		
RemusDB (MINHAS et al., 2011)	X			X	X		
SQLAlchemy (COPELAND, 2008)	X		X			X	
SQLObject (STEVENSON, 2008)	X		X				
Storm (STEVENSON, 2008)	X		X				
Struts (HUSTED, 2003)	X						
Telerik OpenAccess ORM[3]			X			X	
Terralib (CÂMARA et al., 2008)	X		X				
TopLink (BHATTI, 2013)			X				
Web Services[4][5]				X	X		
WebORB Integration Serve[6]r	X		X	X	X	X	

Fonte: autoria própria (2015)

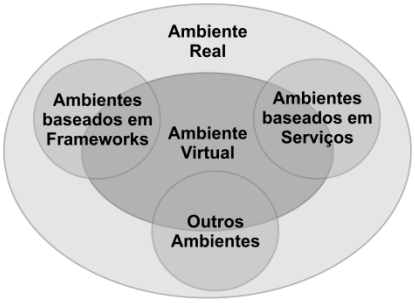
[1] http://www.llblgen.com/
[2] http://operondb.jp/
[3] http://www.telerik.com/docs/default-source/documents/data-access/getting-started-with-telerik-openaccess-orm-guide.pdf?sfvrsn=2
[4] http://www.w3.org/TR/ws-arch/
[5] http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice
[6] http://www.themidnightcoders.com/products.html

No Quadro 2, os principais trabalhos encontrados foram distribuídos com base em sete tipos de classificação: Plataformas de Desenvolvimento; Modelos e padrões de Desenvolvimento; Mapeamento Objeto/Relacional; Serviços; Integração entre bases de dados; Ferramentas de Desenvolvimento e Outras iniciativas. A escolha dessas classificações se deu como resultado da análise das abordagens encontradas e das inferências feitas durante esse estudo. Cada categoria está relacionada a alguma característica apresentada pelos trabalhos estudados e cada trabalho pode se enquadrar em uma ou mais classificações. A compilação deste quadro permitiu uma comparação mais eficiente entre os trabalhos, possibilitando a enumeração das abordagens mais utilizadas e a extração das suas características, vantagens, desvantagens e outros aspectos. Durante essa pesquisa, foi identificada uma forte tendência ao tratamento de bancos de dados relacionais e tecnologias correlatas. Foram encontrados trabalhos voltados a outros tipos de fontes de dados, porém quase sempre em contextos altamente especializados, que vão de encontro à abstração que o trabalho proposto nesta dissertação busca alcançar.

2.1 CONTEXTUALIZAÇÃO GLOBAL DOS TRABALHOS

É importante notar que não existe uma correlação direta entre os as fontes de dados apresentadas no Quadro 1 e os trabalhos apresentados no Quadro 2. Isso ocorre, pois as abordagens adotadas em alguns dos trabalhos admitem a utilização de múltiplos tipos de fontes de dados enquanto outras são restritas a um determinado tipo ou segmento. Essa observação serviu como critério inicial para redução da lista de trabalhos a um subconjunto mais relevante, uma vez que os trabalhos que apresentam limitações excessivas ou requerem condições muito específicas contrariam os objetivos almejados.

Figura 3 - Principais contextos do ambiente de execução



Fonte: autoria própria (2015)

Uma das principais condições restritivas é o próprio ambiente de execução, que pode abranger diversos contextos (Figura 3) e consistem em combinações de hardware e *software* que compõem o meio necessário à execução de um programa. Esse meio pode ser o ambiente real (o sistema operacional diretamente instalado sobre o hardware), um ambiente virtual (uma máquina virtual como a do Java), um *framework* (como o .Net *Framework*) ou mesmo um conjunto de serviços (Como o Apache ou o JBoss).

O ambiente real consiste no conjunto de hardware e *software* (normalmente o sistema operacional) dentro do qual os programas costumam ser executados. Nesse ambiente podem existir máquinas virtuais, emuladores, simuladores e outras camadas assistidas por hardware e/ou *software*, compondo ambientes virtuais que escondem o ambiente de execução real e criam abstrações sobre as quais outros programas podem ser executados, sem a comunicação direta com o sistema hospedeiro. Em ambos os casos, virtual ou real, podemos utilizar *Frameworks*, Serviços e outros mecanismos para abstrair complexidades ou agregar funcionalidades ao ambiente.

Os ambientes baseados *frameworks* são estruturas de programação que servem como plataformas para a construção de programas mais complexos, abstraindo certas peculiaridades do hardware e do sistema operacional sem obscurecê-los. Eles são compostos por bibliotecas que abrangem funcionalidades, entidades e outros elementos, que podem ser utilizados conforme a necessidade. Um ambiente de execução baseado em *frameworks* é capaz suprir todas as necessidades do programa, eliminando sua dependência do ambiente de execução hospedeiro de forma análoga a uma máquina virtual, permitindo que o próprio *framework* funcione como um ambiente aparte.

Em contraste com os *frameworks*, os ambientes criados através de serviços obscurecem completamente o seu ambiente hospedeiro, expondo somente um conjunto predefinido de funcionalidades. O acesso a essas funcionalidades ocorre através de uma interface de comunicação padronizada, pelo uso de protocolos que precisam ser seguidos pelo cliente que consome o serviço, pelo servidor e pelos programas hospedados.

Os serviços atuam de forma completamente independente e separada do programa cliente, pois são *softwares* completos e autônomos. Eles utilizam diversas formas de comunicação entre processos para disponibilizar as suas funcionalidades e abstraem todo o ambiente de execução e toda a infraestrutura sobre a qual operam, fornecendo acesso somente às funcionalidades que se deseja servir.

Tanto os *frameworks* quanto os serviços costumam depender de alguma plataforma hospedeira, responsável por prover um ambiente mínimo necessário à sua execução. Na

Figura 3, os ambientes de execução foram organizados de modo a expor essas interdependências naturais e a imagem apresenta os *frameworks* e os serviços dispostos em polos opostos por não possuem dependência mutua. No entanto, isso não significa que *framework*s não podem utilizar serviços ou que serviços não possam ser baseados em *frameworks*.

Apesar da grande capacidade de abstração que o uso de serviços oferece para o encapsulamento das fontes de dados, essa abordagem não é muito adaptável. Isso ocorre, pois é utilizado um mecanismo previamente concebido com alto grau de dependência de uma infraestrutura específica. Quaisquer modificações nessa infraestrutura e nos protocolos seguidos podem afetar a compatibilidade com os programas clientes. A modularidade dos serviços, também é problemática, pois não é possível destacar do serviço principal o subconjunto das partes desejadas para leva-las com o programa cliente. Essa condição de dependência está relacionada tanto ao seu ambiente de execução quanto à infraestrutura necessária para que ele funcione corretamente. Isso impede que o serviço seja levado para o ambiente de execução do *software* cliente ou que suas partes possam ser isoladas e destacadas em função da necessidade do usuário. Essa abordagem também requer a instalação de um programa separado (o serviço), que tem necessidades de ambiente de execução independentes do programa cliente.

No contexto da abstração de fontes de dados, o uso de serviços permite encapsular as fontes de dados, expondo para o programador somente uma interface de comunicação padronizada. Através dessa interface, as operações sobre os dados são executadas de forma transparente, sem que o programa cliente precise conhecer quais fontes de dados estão sendo realmente manipuladas. Desse modo, uma ou mais fontes de dados podem ser mantidas por um mecanismo intermediário que reduz a complexidade do acesso e da comunicação, utilizando uma interface unificada. Uma técnica semelhante também é utilizada na abordagem de Gateway (MOLZ, 1990), nos Bancos de Dados Federados e em outros trabalhos, sendo definidos três níveis de interoperabilidade: federação, *harvesting* e *gathering* (SAYÃO et al, 2012).

A federação é o nível mais alto de interoperabilidade (SAYÃO et al, 2012) e requer que todos os elementos que participam do processo possuam um compromisso em seguir padrões e protocolos rigidamente definidos. Isso é feito com o propósito específico de garantir a integração do sistema e requer grande esforço das organizações e indivíduos responsáveis pela manutenção da compatibilidade entre os membros da federação. Ao mesmo tempo, também é necessário manter esses elementos atualizados sem que a compatibilidade

entre eles seja afetada (SAYÃO et al, 2012).

Por causa do comprometimento requerido para se estabelecer e manter uma federação, um nível mais baixo de interoperabilidade foi criado: o *Harvesting*. Neste nível, o comprometimento das entidades para manutenção da interoperabilidade limita-se à definição de metadados que são coletados e analisados automaticamente, demandando pouco esforço na definição e na manutenção de padrões (SAYÃO et al, 2012).

Mesmo com um nível de compromisso reduzido, existem situações nas quais o *Harvesting* também se mostra inadequado (SAYÃO et al, 2012). Para isso foi criado um terceiro nível de interoperabilidade denominada *gathering*, que consiste na agregação automática de dados sem protocolos ou padrões estabelecidos (SAYÃO et al, 2012). Nessa técnica, um elemento concentrador (o *gateway*) é responsável por adquirir e processar as informações, repassando aos clientes somente os resultados relevantes. O *gathering* é uma técnica particularmente adequada em sistemas verdadeiramente heterogêneos, no qual existe pouco ou nenhum compromisso prévio com a interoperabilidade (SAYÃO et al, 2012).

Todas as soluções apresentadas atuam como parte do programa final ou integradas com ele, porém outras abordagens também são utilizadas. Muitos ambientes integrados de desenvolvimento (IDEs pelo inglês *Integrated Development Environment*) possuem funcionalidades que permitem a integração de ferramentas externas que agregam novas funcionalidades e atuam tanto no IDE quanto no código do programa gerado. Nos casos extremos, o IDE fornece de forma nativa certas facilidades, dispensando ferramentas externas. Isso permite intervir diretamente no processo de criação dos programas e não no ambiente de execução, frequentemente requerendo pouco esforço do programador. Por esse motivo, não é necessária a instalação dessas ferramentas quando o programa final é distribuído. Contudo, essa abordagem limita o programador aos recursos da ferramenta utilizada, pois as extensões compatíveis com um IDE frequentemente não são compatíveis com os demais. Isso significa que essa ferramenta específica não pode ser empregada se um IDE diferente for utilizado. Como não existe uma ferramenta de desenvolvimento verdadeiramente completa, universal e adequada a todas as situações, a troca dessa ferramenta eventualmente ocorrerá e o programador não poderá contar sempre com os seus recursos.

Muitas outras abordagens foram identificadas e incluem, de forma não restrita, padrões de projetos, interfaces de dados, modelos de desenvolvimento, técnicas de programação, sistemas gerenciadores de bancos de dados (SGBDs), linguagens de consulta, *frameworks* e também combinações simultâneas de várias dessas soluções. Dentre elas, duas mostraram grande adequação aos objetivos desta dissertação em decorrência da versatilidade

e abrangência que elas oferecem: De um lado, os modelos e padrões de projeto definem regras e conceitos abstratos que fornecem orientações para a criação de *softwares* mais coesos, sem limitações intrínsecas de ambiente. Do outro lado, os *frameworks* fornecem estruturas concretas que servem de base para a construção de *softwares*, levando à mesma coesão oferecida pelos modelos e padrões, frequentemente derivando deles.

Para facilitar o estudo, os trabalhos mais relevantes foram distribuídos ao longo de três seções. A seção 2.2 aborda os modelos e padrões de projeto. A seção 2.3 aborda os *frameworks*. A seção 2.4 aborda brevemente as fontes de dados não convencionais e algumas iniciativas para trata-las. No fim do capítulo, a seção 2.5 estabelece um comparativo geral entre os trabalhos, analisando o estado da arte.

2.2 MODELOS E PADRÕES DE PROJETOS

Segundo Gamma et al (2000), “Projetar *software* orientado a objetos é difícil, mas projetar *software* reutilizável orientado a objetos é ainda mais complicado. Você deve identificar os objetos pertinentes, fatorá-los em classes no nível correto de granularidade, definir as interfaces das classes, as hierarquias de herança e estabelecer as relações-chave entre eles. O seu projeto deve ser específico para o problema a resolver, mas também genérico o suficiente para atender problemas e requisitos futuros(...)”.

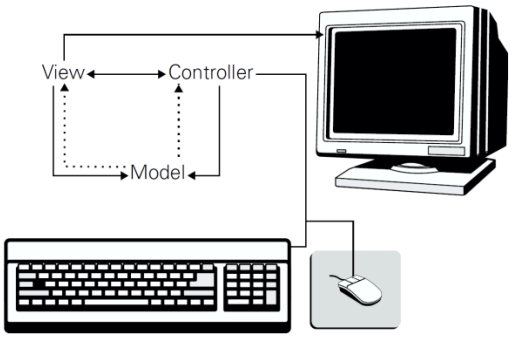
Para lidar com a complexidade apresentada pelos sistemas orientados a objetos, os modelos e padrões de projeto fornecem diretrizes que permitem tratar essa complexidade com soluções que já foram testadas e validadas. Essas soluções, entretanto, precisam ser implementadas pelo programador que as utiliza. Os modelos e os padrões compilam conhecimentos derivados de experiências passadas e permitem trilhar passos conhecidos para a solução de problemas recorrentes. Isso torna o trajeto entre a concepção e o desenvolvimento mais suave e previsível, levando à criação de *softwares* mais coesos em menos tempo do que seria necessário caso o projeto fosse executado sem nenhum conhecimento prévio. Dentre os modelos mais disseminados, podemos citar o modelo relacional (CODD, 1990), que tornou-se referência quase unanime para criação de sistemas gerenciadores de bancos de dados e influenciou até sistemas que não se baseiam diretamente nele.

Partindo de um nível mais elevado de abstração e focando especificamente nas linguagens orientadas a objetos, Gamma et al. (2000) definiu um conjunto de 23 padrões de

projeto. Eles são divididos em padrões de criação, padrões estruturais e padrões comportamentais, abrangendo as mais diversas situações encontradas pelo programador em sua rotina diária. Não faz parte do escopo dessa dissertação abordar cada um desses padrões, porém eles serviram de referência para diversos trabalhos que vieram depois e ainda possuem grande importância até hoje.

Outro trabalho relevante é o modelo MVC (HUSTED; FRANCISCUS, 2003), que apresenta uma solução para o desenvolvimento de *softwares* baseado em três camadas que dão nome ao modelo: Model, View e Controller. Ele trata desde a interface com o usuário até os modelos de dados definidos pelo programador, estabelecendo relações entre essas entidades e padronizando a construção do programa em torno dessas interações, conforme apresentado na Figura 4.

Figura 4 - Modelo MVC



Fonte: Husted e Franciscus (2003, p31)

Conforme definido pelo modelo MVC, cada interface com a qual o usuário interage (View) representa uma visão sobre os dados (Model). Toda a interação com uma dessas visões dispara um evento que é interceptado pelo Controller e recebe o tratamento apropriado. Seguindo as regras definidas no Controller, essa ação pode gerar modificações no Model e depois o fluxo é redirecionado para uma View, que pode ou não ser a mesma por onde a interação começou. Esse modelo é extremamente simples, mas permite a criação de sistemas relativamente complexos e robustos, com intensa reutilização de código e baixo grau de acoplamento. Apesar disso, o MVC não trata a persistência desses dados nem estabelece regras de como isso deve ser feito, mas define uma estrutura consistente que permite a integração com as fontes de dados de forma mais coesa e previsível.

O MVC é um modelo puramente conceitual que não trata os aspectos de implementação da solução nem lida com a persistência dos dados, porém dois outros modelos dão um passo adiante: o Model 1 e o Model 2 (HUSTED, 2003). Segundo Husted, (2003), o Model 1 combina a lógica de apresentação com a lógica de negócio para tratar as interações

com o usuário. Desse modo, quando o usuário interage com o *software*, essa interação é processada pela regra de negócio específica que fica mesclada com a interface, o que gera alto grau de acoplamento. O Model 2 resolve esse problema aprimorando a solução proposta pelo Model 1 com o uso do MVC, incorporando a figura do Controller e separando as camadas. O Model 2 não define qualquer tipo de vínculo com uma forma específica de persistência. Em vez disso ele define a criação de uma quarta camada onde a persistência dos dados deve ser tratada, interagindo diretamente com os modelos e respondendo às interações do usuário através da orquestração do Controller, ampliando a funcionalidade do MVC e mantendo o desacoplamento entre as camadas.

Uma das vantagens no uso de modelos é o fato deles não exigirem o uso de ferramentas específicas nem serem dependentes do ambiente de execução, definindo apenas diretrizes para ajudar o programador a resolver o problema a ser tratado. O uso de modelos, entretanto, requer a adoção de certo compromisso com os padrões definidos, sendo imprescindível o uso de boas práticas para que se obtenha um resultado satisfatório. A implementação desses modelos também requer um esforço significativo até que uma solução utilizável seja construída.

2.3 FRAMEWORKS

Apesar da abordagem baseada em modelos ter grande potencial como solução para a abstração de fontes de dados, a aceitação de novos modelos pelo público alvo mostrou-se insatisfatória. Isso motivou uma mudança na abordagem, direcionando a pesquisa para a produção de um resultado mais concreto.

Seguindo o estudo já realizado e mantendo a linha de pesquisa do projeto original, duas abordagens mostraram grande potencial em relação aos objetivos almejados. A primeira utiliza um recurso chamado Bancos de Dados Federados (BARBOSA et al., 2004) e a segunda consiste no uso de *Frameworks*. Os trabalhos que não se enquadram nessas abordagens foram descartados, por não cumprirem a maior parte dos objetivos almejados, mostrando-se insatisfatórios em relação à proposta dessa pesquisa.

Os Bancos de Dados Federados (BDFs) (BARBOSA et al., 2004) consistem em mecanismos de abstração através do qual um Sistema Gerenciador de Bancos de Dados (SGBD) é utilizado como interface entre a fonte de dados real e o aplicativo que requisita os dados. As requisições são feitas através de instruções SQL convencionais, que são traduzidas

em ações específicas na fonte de dados de destino. Isso ocorre sem que o aplicativo possua qualquer conhecimento sobre o que está por trás do SGBD principal e permite o acesso a vários mecanismos diferentes através de uma interface unificada.

O uso de um SGBD como interface traz uma grande desvantagem aos BDFs, pois isso limita as funcionalidades do sistema àquelas definidas pelo fabricante desse SGBD, havendo pouca ou nenhuma flexibilidade. Alguns SGBDs dispõem de código fonte aberto, porém não possuem suporte nativo ao uso de fontes de dados não convencionais e requerem intervenções complexas para serem adaptados. Isso acaba restringindo seu uso aos bancos de dados relacionais e tecnologias semelhantes. Existe também a restrição de ambiente, já que o SGBD precisa ser instalado em um ambiente específico para que suas funcionalidades tornem-se disponíveis. Se o ambiente de execução for um sistema operacional ou dispositivo para o qual o SGBD não possui suporte, seu uso torna-se restrito ou mesmo inviável.

Diferente dos bancos de dados federados, os *frameworks* não são restritos a um domínio de aplicação específico. Existem *frameworks* para as mais diversas finalidades, inclusive para a manipulação de fontes de dados. Os *Frameworks* utilizam uma abordagem quase oposta aos BDFs, pois consistem em bibliotecas de funcionalidades integradas e extensíveis que servem de base para a construção de *softwares* mais complexos. Isso se opõe aos BDFs por se tratar de uma plataforma criada com o propósito de ser estendida ao invés de uma solução pronta, permitindo ao programador adaptá-lo à sua necessidade específica. O *software* resultante dispensa o uso de elementos adicionais e o *framework* é levado ao ambiente de execução como parte integrante do programa compilado, superando a restrição inerente aos BDFs.

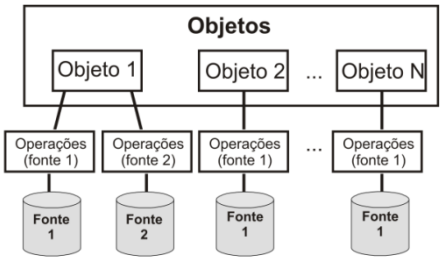
Após a análise dos resultados encontrados, optou-se por restringir a pesquisa ao o estudo dos *frameworks* que tornaram-se o ponto focal do trabalho. Nesse contexto, duas abordagens principais puderam ser identificadas, permitindo a classificação de todos os trabalhos restantes: os padrões Expert e Database Broker (MOLZ, 1990).

O padrão Expert (MOLZ, 1990) define que cada entidade deve tratar a própria persistência e devem ser dotadas de todas as funcionalidades necessárias para esse propósito. Isso cria alto grau de especialização e requer intervenções no código a cada mudança, acoplando fortemente as entidades com as fontes de dados. Essa característica é particularmente indesejável para o propósito desse trabalho, uma vez que a independência das fontes de dados é um dos objetivos almejados. Essa independência não é possível quando a entidade a ser mapeada está fortemente acoplada com a fonte de dados.

Dentre os trabalhos com alguma relevância no mercado, o único baseado no padrão

Expert foi o Active Records (FOWLER, 2002). Ele é um padrão de projeto e também um *framework*, com foco direcionado especificamente para linguagens orientadas. A sua estrutura é representada na Figura 5.

Figura 5 - Modelo básico de persistência do Active Records

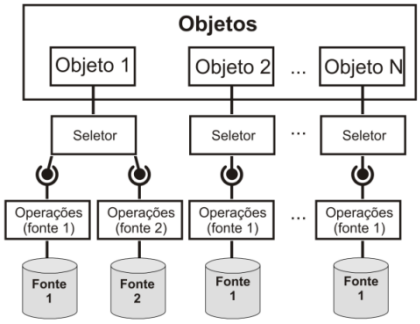


Fonte: autoria própria (2015)

De acordo com as diretrizes do padrão Active Records, as operações sobre os dados são encapsuladas dentro do próprio objeto a ser persistido. Isso significa que para cada operação sobre esses dados existe um método especializado, conforme a representação da Figura 5.

É natural imaginar que a mesma entidade possa interagir com mais de uma fonte de dados e duas abordagens podem ser adotadas nesse caso. A primeira já foi apresentada na Figura 5 e consiste em replicar os métodos para cada uma das fontes de dados, conforme a conveniência. Essa opção gera muito retrabalho e prejudica seriamente a transparência e a legibilidade do código uma vez que o vínculo com cada fonte de dados acaba ficando explícito na interface entre o objeto e os demais módulos e camadas do programa.

Figura 6 - Modelo de persistência do Active Records com seleção de fonte de dados



Fonte: autoria própria (2015)

Uma solução mais elegante é representada na Figura 6 e consiste em incluir no projeto da classe uma estrutura auxiliar para permitir a seleção dinâmica das fontes de dados que os métodos deverão afetar. Tanto o seletor quanto as operações ainda são encapsulados em cada

objeto, porém a manipulação do objeto torna-se mais simples e mais transparente.

O Active Records apresenta uma solução eficaz para o problema da abstração das fontes de dados, pois as operações e o acesso aos dados ficam encapsulados no próprio objeto e não precisam ser recriadas ao longo do programa principal. Por outro lado, o conhecimento sobre cada fonte de dados fica intrínseco na construção das entidades, restringindo suas funcionalidades ao subconjunto que o objeto encapsula. Suas rotinas rigidamente definidas criam forte acoplamento com as fontes de dados, requerendo intervenções em cada uma das entidades sempre que alguma dessas fontes sofre alterações, resultando em retrabalho.

Outro problema inerente ao Active Records diz respeito aos possíveis efeitos colaterais de alterações na estrutura de persistência. Esses efeitos podem se propagar por todo o programa, pois tais alterações podem interferir de maneira indesejável no comportamento das entidades que foram modificadas. Isso exige a execução de testes em todos os pontos do programa que referenciam essa entidade, mesmo que a interação não ocorra diretamente com os métodos que sofreram intervenção.

Seguindo uma estratégia oposta ao padrão Expert e ao Active Records, o padrão Database Broker (MOLZ, 1990) define que as entidades não devem conter quaisquer informações sobre sua própria persistência. Em vez disso, devem existir entidades auxiliares capazes de tratar a persistência, mantendo as entidades principais desvinculadas das fontes de dados. Essa abordagem mostra grande coerência com a proposta de abstração das fontes de dados, pois desvincula completamente as estruturas de dados e a persistência dos dados. Por causa dessa característica, ela é seguida pela maioria dos *frameworks* estudados, o que permitiu direcionar o rumo da pesquisa e diminuir consideravelmente o conjunto de trabalhos a serem estudados.

Dentre os *frameworks* analisados, quatro destacaram-se por apresentarem grande similaridade com a solução proposta deste trabalho. Todos eles seguem o padrão Database Broker, porém utilizam abordagens significativamente distintas:

a) ADO.Net (Active Data Object do .Net *Framework*);
b) LINQ (Language INtegrated Query);
c) Entity Framework
d) Hibernate (BAUER 2005).

Antes de falar sobre o ADO.Net, é necessário fazer uma distinção entre ADO e ADO.Net. O ADO é um conjunto de entidades para acesso a dados adotado no Visual Studio (plataforma de desenvolvimento da Microsoft) até meados de 2001. O ADO utilizava um modelo de acesso no qual uma conexão com o banco de dados era mantida permanentemente

aberta enquanto os dados estivessem sendo manipulados.

No ADO, todas as ações sobre a interface (Recordset), ocorriam sempre no registro ativo que era controlado por um identificador posicional ordinal (Cursor). Quaisquer alterações eram imediatamente refletidas no banco de dados e precisavam ser confirmadas ou canceladas antes que o Cursor pudesse ser movido ou que o Recordset pudesse ser fechado.

O ADO.Net abandonou o modelo de conexão permanente do seu predecessor, substituindo os Recordsets e Cursores por entidades que representam uma estrutura similar a um banco de dados relacional diretamente na memória (*DataTables*, *DataRelations*, *DataSets* e outras). Esse modelo permite ao programador interagir com os dados de maneira desconectada, lendo e gravando no banco de dados em lotes, somente quando julgar conveniente ou necessário. Essa abordagem cria certo grau de transparência em relação às operações sobre os dados, uma vez que muitas operações não precisam ser definidas detalhadamente, porém não abstrai completamente as particularidades do SGBD. O programador ainda precisa tratar separadamente o mapeamento objeto-relacional e precisa definir as expressões em SQL que serão utilizadas para interagir com o banco de dados. Apesar disso, diversos aspectos dessas interações são abstraídos pelo *framework*, sem a necessidade de intervenções adicionais no código.

Um exemplo da abstração oferecida pelo ADO.Net consiste no uso de transações para criação de operações atômicas, que só podem ser concluídas integralmente ou revertidas em caso de falha. Para suprir essa necessidade, o ADO.Net possui uma entidade especializada denominada "*Transaction*" e que pode ser instanciada e vinculada à operação sem a necessidade de escrevê-la diretamente em código SQL. Toda a operação com o SGBD relativa à criação e manutenção da transação é tratada pelo próprio mecanismo de forma completamente transparente.

A LINQ (Language Integrated Query), complementa a funcionalidade oferecida pelo ADO.Net, simplifica o mapeamento objeto-relacional através do uso de extensões que agregam funcionalidades à linguagem de programação. Seu funcionamento consiste em incorporar representações de alto nível das expressões de consulta diretamente na linguagem de programação hospedeira. Essas expressões interagem com as entidades do programa de forma totalmente transparente e o resultado é compilado e incorporado ao aplicativo. Essa abordagem obtém grande desempenho, porém requer que os mapeamentos sejam rigidamente definidos no código.

Através da análise do seu funcionamento e das suas características, dentre os trabalhos pesquisados, o Hibernate e o Entity *Framework* foram identificados como as soluções mais

bem sucedidas na abstração das fontes de dados. Ambos são *frameworks* que visam prover o mapeamento objeto-relacional e ambos usam abordagens parecidas, abstraindo as fontes de dados através de mapeamentos transparentes. Ambos seguem modelos de desenvolvimento semelhantes, encapsulando as operações sobre as fontes de dados em entidades que são selecionadas e instanciadas dinamicamente com base em declarações armazenadas em arquivos XML. Cada *framework* utiliza sua própria linguagem, porém as duas soluções guardam muitas similaridades e também compartilham alguns problemas.

O Hibernate possui como vantagem a capacidade de enriquecer as expressões de consulta através de representações de alto nível que são traduzidas pelo *framework* conforme o banco de dados que estiver mapeado. Isso é feito pela combinação de expressões textuais com classes predefinidas que referenciam, respectivamente, os membros da classe mapeada e as operações do modelo relacional. Esse aprimoramento, no entanto, é feito no código, levando à mesma deficiência da LINQ e impondo limitações no que pode ser feito com o uso de mapeamentos definidos em arquivos externos ao código fonte do programa.

O Entity *Framework* apresenta como vantagem a extrema simplicidade na sua utilização. Além de incluir ferramentas que automatizam e abstraem todo o processo de mapeamento, possui suporte a um conjunto maior de funcionalidades sem a necessidade de código. Isso inclui *Stored Procedures*, relacionamentos complexos e outros tipos de estruturas suportados pelos bancos de dados modernos.

Tanto o Hibernate quanto o Entity *Framework* apresentam baixo suporte à extensibilidade, o que restringe as operações possíveis a um conjunto limitado de possibilidades. Apesar do conjunto de funcionalidades nativas ser relativamente grande, qualquer funcionalidade nova ou que fuja aos limites dos bancos de dados relacionais é difícil ou até mesmo inviável. Outra limitação importante apresentada por ambos os *frameworks* é a incapacidade de manipular nativamente mais de uma fonte de dados por vez. O programador pode intervir através de código para contornar parcialmente essa limitação, alternando programaticamente entre as fontes quando necessário. Isso precisa ser feito diretamente no código e todas as fontes de dados envolvidas precisam compartilhar a mesma estrutura para serem compatíveis com o mapeamento, que é fixo e único.

2.4 FONTES DE DADOS NÃO CONVENCIONAIS

Uma característica comum entre todas as soluções apresentadas é o forte

direcionamento para fontes de dados convencionais, ou seja: Bancos de dados e tecnologias correlatas. Essas fontes de dados, entretanto, não são as únicas que podem ser utilizadas em um programa. Existe grande variedade de fontes de dados e o uso de cada uma delas depende das necessidades específicas de cada projeto, conforme as demandas e recursos do cliente. Tais fontes de dados são aqui chamadas de “não convencionais”, pois fogem ao conceito de bancos de dados. É comum que elas não apresentem a estrutura de armazenamento e a organização de um banco de dados relacional. Dentre as fontes de dados não convencionais mais comuns, podemos citar arquivos binários e arquivos de texto não estruturado (sequenciais ou não). Esses arquivos precisam ser analisados para que sua composição seja identificada antes que os dados possam ser extraídos e utilizados.

Quando é necessário o uso de fontes de dados não convencionais, o programador precisa recorrer à manipulação direta dos dados em sua forma bruta. Em alguns casos existem bibliotecas que podem ser utilizadas para a interação com essas fontes de dados, porém cada caso precisa ser tratado de forma individual e particular, sendo necessária a criação de peças especializadas de *software* para que seja obtido algum grau de padronização.

Entre as fontes de dados não convencionais em ascensão podemos citar os dispositivos de acesso ao Sistema de posicionamento Global (GPS, pelo inglês *Global Positioning System*), mensagens enviadas através do Serviço de Mensagens Curtas (SMS, pelo inglês *Short Message System*) e os fluxos de dados enviados através do *Bluetooth* e da Comunicação de Campo Próximo (NFC pelo inglês *Near Field Communication*), entre outras. Essas fontes de dados são cada vez mais comuns em dispositivos móveis e muitas delas também estão sendo adotadas em outras plataformas.

A aquisição de dados a partir de um dispositivo de GPS requer a utilização de comandos específicos, que variam de um dispositivo para o outro, conforme o padrão adotado pelo fabricante, apesar de existirem convenções sobre a representação dos dados obtidos. No caso de mensagens SMS, existe uma padronização vigente em relação aos comandos utilizados pelos modems, chamados de comandos AT, porém ocorrem pequenas variações de um fabricante para outro e até entre dispositivos diferentes do mesmo fabricante. No caso de dispositivos móveis, o envio e o recebimento de SMS ocorrem através das funcionalidades do sistema operacional, sem a necessidade do uso de comandos AT, porém cada sistema operacional diferente possui seu próprio conjunto de funções com métodos de acesso bem específicos.

Figura 7 - Exemplo de comunicação via Bluetooth no sistema Android

Listagem 2. Verificando o suporte à API.

```
1. BluetoothAdapter adaptador =
     BluetoothAdapter.getDefaultAdapter();
2. if (adaptador == null) {
3.   // Aparelho não suporta Bluetooth
4. }
```

Listagem 3. Deixando o aparelho visível.

```
1. if (!adaptador.isEnabled()) {
2.   Intent enableBtIntent =
       new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
3.   startActivityForResult(enableBtIntent, BT_ATIVAR);
4. }
```

Fonte: Devmedia, Mobile Magazine 35[7]

Em relação às tecnologias como NFC e Bluetooth, apesar de não serem as fontes de dados propriamente ditas, ambos requerem tratamento especial para que permitam a transferência dose dados. Eles diferem de outras formas de conectividade, cuja utilização ocorre de forma transparente. Em ambos os casos, o programador precisa requisitar explicitamente o estabelecimento da conexão, verificando a conectividade e iniciando ou interrompendo explicitamente a conexão. Podemos observar esse tipo de situação na Figura 7, onde é ilustrado o acesso ao adaptador *Bluetooth* em um dispositivo com sistema operacional Android. Fazendo uma analogia com uma conexão de rede convencional, isso seria equivalente a solicitar ao sistema operacional uma referência para a placa de rede, que precisaria ser configurada e conectada à rede de forma manual, antes mesmo de se estabelecer uma conexão através de sockets. Na prática, caso se tratasse de uma conexão de rede convencional, simplesmente solicitaríamos a conexão com um determinado endereço e o sistema operacional cuidaria desse processo de forma totalmente transparente, sem a necessidade de uma intervenção explícita através de código. Para o NFC e o Bluetooth, isso não é tão transparente e precisamos de um passo extra, apenas para estabelecer a rede entre os dispositivos. Muitas das tecnologias emergentes já se tornaram extremamente populares em

[7] Disponível em: http://www.devmedia.com.br/comunicacao-via-bluetooth-no-android-artigo-webmobile-35/20464

dispositivos móveis tais como *tablets*, *smartphones*, telefones celulares, entre outros. Essas plataformas apresentam grande potencial comercial e, segundo nota[8] divulgada em novembro de 2010 pela Agência Nacional de Telecomunicações (Anatel), o número de telefones celulares no Brasil ultrapassa o número de habitantes.

Apesar da presença quase universal dessas tecnologias, muitas delas ainda são pouco exploradas em sistemas de automação comercial, porem seu uso é intensivo em *softwares* destinados ao mercado de entretenimento. A maioria dos dispositivos apresenta pelo menos um tipo de tecnologia de conectividade e uma parte significativa deles possui suporte à instalação de aplicativos. Esses casos costumam ser tratadas de modo particular, não havendo padronização entre eles.

Partindo para tecnologias mais tradicionais, como os bancos de dados, também existem situações nas quais os métodos convencionais deixam de ser aplicados. Apesar de fugir ao escopo dessa dissertação, um caso particular, que mostrou-se digno de menção e vem ganhando força nos últimos anos, é o No-SQL. Seus defensores alegam que o modelo relacional e a SQL prejudicam o desempenho e a eficiência das operações sobre os dados (DE SOUZA, 2015).

O NoSQL, do inglês *Not Only SQL*, tem como foco principal sistemas com grandes volumes de dados e acesso concorrente intensivo, priorizando o desempenho em detrimento à facilidade de uso (DE SOUZA, 2015). As operações em NoSQL incluem elementos que definem detalhadamente cada aspecto das operações, permitindo que o programador otimize manualmente as consultas de forma refinada. Isso requer profundo conhecimento da estrutura e do funcionamento do banco de dado que está sendo manipulado, indo de encontro à abstração das fontes de dados.

2.5 ANÁLISE DOS TRABALHOS RELACIONADOS

Cada uma das soluções apresentadas busca resolver problemas específicos, algumas deles agindo de forma complementar em relação a outras ou agregando funcionalidades não contempladas pelas demais. As abordagens seguidas diferem umas das outras e muitas delas não abstraem completamente as fontes de dados. As que atingem nível satisfatório de abstração apresentam limitações indesejáveis, restringindo o conjunto de operações e/ou sacrificando a interoperabilidade.

[8] Disponível em: http://www.anatel.gov.br/Portal/exibirPortalNoticias.do?acao=carregaNoticia&codigo=21613

Conforme apresentado no Quadro 2, quase todos os trabalhos encontrados tratam satisfatoriamente o mapeamento entre as fontes de dados e as estruturas utilizadas nas linguagens orientadas a objetos, porém quase todos são restritos ao escopo dos bancos de dados relacionais. Desses trabalhos, apenas dois possuem suporte nativo à integração entre fontes de dados e somente um desses apresenta um grau satisfatório de abstração, mas utiliza uma abordagem baseada em serviços. Conforme visto anteriormente, os serviços são programas independentes que podem estar em um ambiente de execução diferente do programa principal. Isso impõe restrições à utilização dos serviços, pois exige a disponibilidade de um canal de comunicação e impede a sua utilização em um contexto desconectado. A distribuição do software também fica prejudicada, já que o ambiente de execução do programa e o ambiente de execução do serviço do qual ele depende não é necessariamente o mesmo.

Nenhum dos trabalhos encontrados trata nativamente a interoperabilidade entre fontes de dados convencionais e não convencionais. Apesar disso, muitos deles utilizam abordagens que possuem potencial para oferecer esse tipo de funcionalidade, embora esse potencial não seja aproveitado.

A abordagem baseada em modelos apresenta grande potencial para lidar com todos os problemas apresentados nessa dissertação, porém a aceitação de novos modelos pelo público alvo mostrou-se insatisfatória, reduzindo significativamente a probabilidade de sucesso desse tipo de solução. Essa abordagem também requer muito tempo de aprendizado e exige do programador dedicação e compromisso com a manutenção de padrões, tomando muito tempo até que se atingir um nível adequado de familiaridade.

A abordagem baseada em *frameworks* apresenta as mesmas vantagens da abordagem baseada em modelos, porém permite a apresentação de uma solução pronta capaz de contornar as limitações de aprendizado. Diversos dos trabalhos encontrados utilizam essa abordagem, porém nenhum deles trata a interoperabilidade, limitando-se ao uso de uma fonte de dados por vez. Essa decisão de projeto eventualmente força o programador a desviar o foco do problema principal para contornar essa restrição, gerando retrabalho e aumentando o tempo e o custo do projeto.

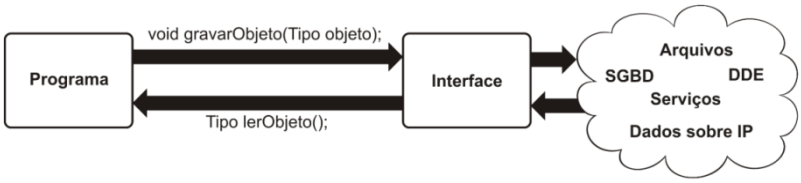
Mesmo desprezando a ausência da interoperabilidade nos *frameworks* encontrados e o foco restrito aos bancos de dados relacionais e tecnologias correlatas, as abordagens escolhidas baseiam-se em regras rigidamente definidas. Isso limita a capacidade de expressão do programador a um subconjunto que muitas vezes não é suficiente, forçando a criação de código adicional apenas para contornar as restrições.

Esse trabalho visa contornar os problemas apresentados, propondo uma solução com alto grau de abstração, sem comprometer o poder de expressão, focado na extensibilidade e na integração entre fontes de dados convencionais e não convencionais.

3 EVOLUÇÃO DO DESENVOLVIMENTO ORIENTADO A DADOS: DO MODELO AO *FRAMEWORK*

O Modelo de Desenvolvimento Orientado a Dados (Modelo DOD ou MDOD) foi concebido no início do desenvolvimento dessa dissertação. Ele consiste em um modelo de abstração no qual o programa interage com os dados sem a necessidade de conhecer previamente a sua fonte, conforme ilustrado na Figura 8. O termo "Orientação a Dados" foi cunhado justamente para enfatizar que, nesse contexto, os dados são importantes e não a sua fonte. Nesse modelo, o programa conhece a existência de um repositório de dados, representado como uma nuvem, mas desconhece as fontes de dados, interagindo somente com uma interface padronizada através da qual todas as operações são realizadas.

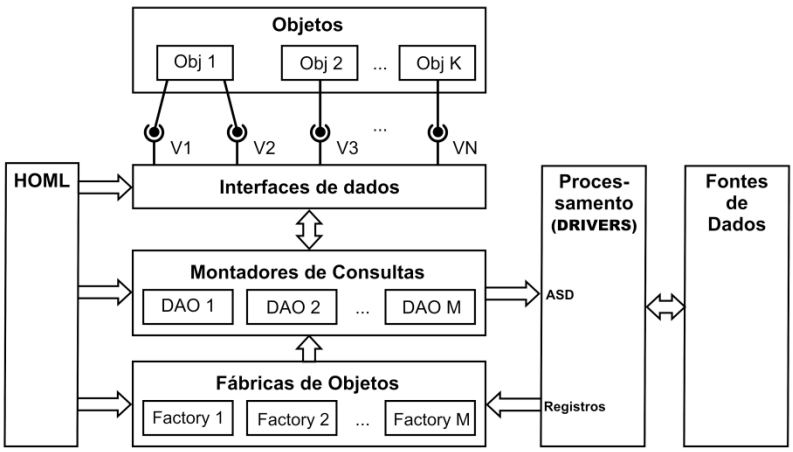
Figura 8 - Interação entre programa e fonte de dados no modelo DOD



Fonte: autoria própria (2015)

Para permitir a implementação do modelo DOD, foi desenvolvida a arquitetura do apresentada na Figura 9, que deu origem ao *Framework* para Desenvolvimento Orientado a Dados (FrameDOD), proposto nessa dissertação.

Figura 9 - Arquitetura de um sistema baseado no FrameDOD



Fonte: autoria própria (2015)

O MDOD possui forte inspiração no modelo MVC (Model View Controller), tendo como ponto de partida a camada onde o MVC para: o tratamento das fontes de dados. Na arquitetura apresentada na Figura 9, os objetos são as entidades do modelo de negócio, que o MVC chama de *Models*. Para cada tipo diferente de objeto, existem diversas visões (V1, V2, V3,..., VN), que não devem ser confundidas com as *Views* definidas pelo MVC. Essas visões são interfaces de comunicação que abstraem as fontes de dados e determinam de onde os dados vêm, para onde eles vão e como serão manipulados nesse processo. Para isso, todas as entidades do sistema e suas operações precisam ser declaradas em arquivos de configuração que utilizam o que denominamos Linguagem Hierárquica para Mapeamento de Objetos (HOML pelo acrônimo em inglês). Essa linguagem foi criada especificamente para dar suporte ao FrameDOD e surgiu como uma necessidade no decorrer do seu desenvolvimento. Esse processo de declaração das fontes de dados, das entidades e das suas operações é denominado mapeamento e permite que todas as partes da arquitetura tomem conhecimento de quais fontes de dados estão disponíveis e de como se comunicar com elas. Isso tem como propósito permitir que os modelos de dados (tabelas, arquivos, etc.) sejam transformados em modelos de memória (objetos) e vice-versa, sem a necessidade de intervenção no código.

Quando um objeto é enviado através de uma Visão, o FrameDOD consulta os mapeamentos em HOML para determinar como esse objeto deverá ser representado, conforme a operação que foi solicitada. Essa representação é enviada ao montador de consultas que utiliza o par Objeto/Visão para consultar os mapeamentos e determinar em que fonte de dados essa requisição deverá ser processada. Para cada fonte de dados declarada existe um adaptador especializado associado (*Driver*), que reconhece tanto as representações utilizadas pelo FrameDOD quanto a forma de comunicação específica da fonte de dados, permitindo a tradução bidirecional entre os dois meios. Para isso, a interface de dados transfere os objetos para o Montador de consultas apropriado, onde eles são transformados Árvores de Sintáxe Abstratas (AST) e encaminhadas o *Driver* específico. Essa requisição é traduzida pelo Driver e processada na fonte de dados, obtendo-se como resposta um conjunto de dados brutos. Esses dados são encaminhados à fábrica de objetos específica, onde são transformados em objetos e encaminha de volta à interface que originou a requisição.

Para entender melhor a motivação para a criação do FrameDOD, é necessária uma breve explicação sobre o MVC. No modelo MVC, todas as interações com o usuário ocorrem através das *Views*, que correspondem às telas que o usuário vê. Essas interações são processadas pelo *Controller*, que interliga as *Views* com os *Models*. Isso significa que para cada *View* existe um *Model* correspondente. O MVC, entretanto, não define o que deve ser

feito com os dados daí em diante, cabendo ao programador criar uma quarta camada para trata-los. O MDOD e o FrameDOD se encaixam justamente nessa lacuna.

3.1 TRATAMENTO DE FONTES DE DADOS NÃO CONVENCIONAIS

O tratamento de fontes de dados não convencionais no FrameDOD ocorre através de uma representação pseudo-relacional da fonte de dados. Para o programa que solicita os dados, essa representação e transparente, pois tudo o que esse programa tem acesso é à interface de comunicação. Tomemos como exemplo o acesso a um dispositivo de GPS, conforme segue abaixo.

O GPS é uma fonte de dados não convencional extremamente atípica. Diferente de um banco de dados, o GPS não espera por requisições, em vez disso, os dados são enviados em intervalos regulares, mesmo que o programa não esteja aguardando por eles. Para que esse tipo de fonte de dados seja utilizado pelo FrameDOD, podemos adotar duas abordagens distintas. A primeira consiste na buferização dos dados e a segunda consiste em utilizar requisições bloqueantes.

A buferização dos dados seria feita pelo driver, sem a necessidade de intervenções por parte da do programa principal. As últimas localizações enviadas pelo GPS seriam mantidos na memória ou em um dispositivo de armazenamento com informações espaço-temporais. O programa principal faria apenas solicitações eventuais e receberia uma lista de resultados que poderiam ser filtrados ou ordenados de forma análoga a uma operação Select básica.

A utilização de requisições bloqueantes seria uma forma alternativa de abordagem, nesse caso cada requisição traria somente um registro como resultado, porém o programa permaneceria bloqueado por um tempo a ser especificado na requisição. Transformando esse procedimento em uma operação Select, ela seria equivalente à seguinte expressão: “SELECT * FROM GPS WHERE Timeout < 5000”. Nesse caso o driver esperaria por uma resposta do GPS por 5000 milissegundos. Se o dado esperado chegar antes do fim desse tempo, o controle é imediatamente devolvido ao programa principal, retornando um item como resultado. Se o dado não chegar antes do fim dos 5000 milissegundos, o controle é devolvido compulsoriamente ao programa, trazendo uma lista vazia ou um item nulo, dependendo do tipo de requisição que foi feita.

Esse exemplo demonstra bem como as fontes de dados não convencionais podem ser abstraídas através de operações relacionais simples, porém a forma ideal de tratar cada caso

dependerá sempre da necessidade do usuário e o driver poderá ser adaptado para se ajustar a necessidades específicas.

3.2 O LIMITE ENTRE A HETEROGENEIDADE E A UNIFORMIDADE

Conforme apresentado, o tratamento de fontes de dados convencionais e não convencionais é feito através de uma representação pseudo-relacional. Desse modo, o Driver é responsável por resolver os problemas relacionados à fonte de dados que está sendo manipulada, respondendo às requisições de forma padronizada. Isso significa que temos dois contextos diferentes. O primeiro contexto é o canal de comunicação Driver/Programa, através do qual as requisições fluem de forma padronizada e uniforme, com requisições baseadas em operações relacionais e respostas baseadas em objetos. O segundo contexto é completamente heterogêneo, consistindo nas fontes de dados que são acessadas pela interface Driver/Fonte de dados.

A interface entre o contexto homogêneo e o contexto heterogêneo é feita através dos drivers e da HOML. Os drivers são responsáveis pela comunicação detalhada, dentro do padrão utilizado por cada fonte de dados em particular. Essas informações são então traduzidas para uma representação pseudo-relacional, permitindo que o programa faça requisições simples e obtenha dados uniformizados. Nesse modelo de abstração, as diferentes formas de representação dos dados precisam receber nomes que seriam vistos pelo programa principal como se fossem Tabelas. Do mesmo modo, os diferentes dados contidos em cada representação seriam vistos de forma análoga a campos nessas tabelas. Através de uma interface padronizada, esses dados podem ser enumerados e manipulados através de requisições relacionais, sem que o programa principal precise compreender como cada fonte de dados funciona.

É importante observar que a heterogeneidade não deixa de existir, porém isso é abstraído para que o programa principal não precise lidar com esse problema.

3.3 INTRODUÇÃO DA HOML

A ideia original dessa dissertação consistia em propor MDOD como solução, porém a dificuldade na aceitação de novos modelos pelo público alvo direcionou a pesquisa para a apresentação de resultados mais concretos. Isso levou à criação de um *framework* denominado

FrameDOD que implementa as funcionalidades previstas no MDOD.

No início do processo de desenvolvimento, a estrutura criada para o FrameDOD teve como inspiração a Language Integrated Query (LINQ), baseando-se em estruturas de programação para tentar representar as operações sobre os dados. Essa abordagem funcionou satisfatoriamente, porém criava restrições severas à extensibilidade por definir rigidamente no código as regras de manipulação das fontes de dados. Isso prejudicava a transparência do processo e exigia a criação manual de diversas estruturas do mecanismo.

Para atingir o nível de abstração e extensibilidade desejados, foi necessária a definição de algumas metas que exigiram o aprimoramento da arquitetura e levaram à criação de uma linguagem denominada *Hierarchical Object Mapping Language* (HOML). A criação da HOML teve como base o estudo de trabalhos como o Hibernate e o Entity *Framework*, que utilizam arquivos XML para definição das regras de mapeamento entre os modelos de persistência e os modelos de memória. Essa abordagem oferece grande flexibilidade, porém as decisões de projeto podem facilmente levar a uma situação na qual o usuário cria aversão ao se sentir oprimido pela complexidade dos arquivos de configuração. Esse tipo de reação ocorre principalmente pela falta de familiaridade do usuário com a estrutura e a linguagem adotadas para a configuração. Para minimizar esse problema, alguns cuidados foram tomados na concepção da HOML.

Os principais elementos da HOML foram baseados no modelo relacional, utilizando marcas como “SELECT”, “INSERT”, “DELETE” E “UPDATE” para se referir às operações do modelo relacional que esses termos invocam. Todas as operações em HOML são, portanto, derivadas das expressões SQL equivalentes, utilizando formas encadeadas dessas expressões. Isso significa que a marca “SELECT” possui como membros elementos como “FROM”, “JOIN”, “WHERE”, “ORDER” e outros, trazendo familiaridade para o usuário que já está acostumado com a utilização desses elementos. Isso significa que a formação das expressões segue uma linha intuitiva que pode ser inferida a partir dos conhecimentos prévios de SQL.

Obviamente, muitos elementos da HOML não são tão familiares para o usuário, porém o conjunto de elementos não derivados da SQL é extremamente pequeno, restringindo a quantidade de coisas novas que o usuário precisa aprender a um pequeno conjunto de termos padronizados. Outra preocupação na elaboração da HOML foi a implementação de recursos para diminuir a quantidade de código que o usuário precisa escrever, através da adoção de algumas convenções. Por exemplo: se um modelo de persistência possui estrutura equivalente à sua contraparte no modelo de memória, não é necessário que o usuário escreva detalhadamente cada uma das operações, pois isso pode ser extrapolado a partir da própria

estrutura da entidade. Nesse caso, as operações básicas possuem uma semântica padronizada, dispensando a sua declaração explícita. Por isso, o usuário não é obrigado a definir explicitamente essas operações, mas terá a opção de declará-las manualmente, se desejar. Alguns exemplos desse tipo de declaração serão apresentados adiante.

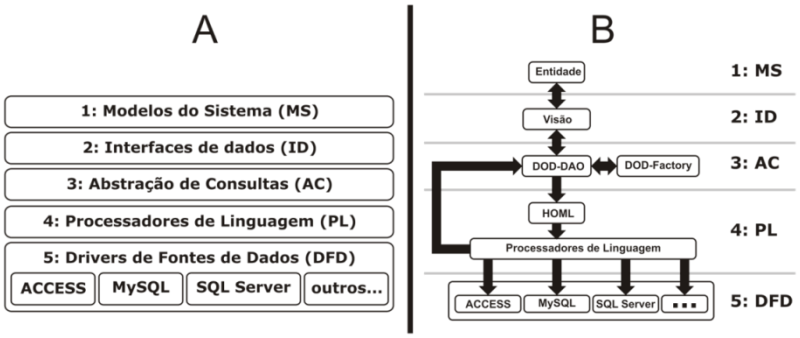
3.4 O FRAMEDOD

O FrameDOD foi criado como uma implementação do MDOD, de modo que ambos compartilham a mesma estrutura. Para a composição de uma arquitetura adequada aos objetivos propostos, é importante considerar que a interação com a fonte de dados, qualquer que seja essa fonte, sempre tem como objeto primário os dados e não a fonte em si. Apesar disso, as particularidades de cada fonte de dados acabam permeando o código, interferindo no projeto e no funcionamento do programa. Isso significa que o mecanismo utilizado para contornar esse problema deve ser capaz de permitir a interação com os dados de maneira uniforme e genérica sem a necessidade de incluir explicitamente no código as informações sobre onde ou como esses dados serão obtidos. Ao mesmo tempo, esse mecanismo não deve ser restritivo, já que as necessidades do programa não são limitadas pelas decisões de projeto do *framework*. Para evitar que isso ocorra, a extensibilidade é essencial, permitindo ao programador criar e integrar novos recursos ao arcabouço existente, com o menor esforço possível, sempre que julgar necessário.

Para lidar com os problemas apresentados, o FrameDOD se baseia no modelo de interação onde as operações sobre os dados não requerem o conhecimento prévio da sua fonte. Para isso, foi adotada a arquitetura apresentado na Figura 10. Nesse modelo conceitual, pode ser observado que a as operações de entrada e saída de dados ocorrem através uma interface abstrata denominada “VISÃO”, que encapsula toda a comunicação com a fonte de dados e disponibiliza um conjunto de métodos através dos quais as interações ocorrem. A abordagem proposta é baseada em alguns padrões de projeto (GAMMA et al. 2000) e busca atingir o nível de interoperabilidade *gathering* (SAYÃO et al, 2012). Para isso, o FrameDOD oferece uma estrutura capaz de analisar as entidades a serem persistidas, extraindo automaticamente as informações necessárias para promover a aquisição e a persistência dos dados de forma transparente. O programador pode interferir no funcionamento desse mecanismo criando definições personalizadas que são armazenadas fora do código do programa, em arquivos de configuração. Desse modo, o conhecimento prévio sobre a fonte de dados se torna

desnecessário para a sua utilização e as definições que determinam como as operações serão processadas podem ser alteradas a qualquer momento, antes ou depois da criação do código, sem depender qualquer tipo de intervenção posterior no programa.

Figura 10 - Arquitetura do FrameDOD



Fonte: autoria própria (2015)

O modelo de interação do FrameDOD assemelha-se ao funcionamento do Hibernate e do Entity *Framework*, porém não possui algumas das limitações dessas abordagens. O suporte a múltiplas fontes de dados simultâneas, por exemplo, é nativo e vários mapeamentos podem ser declarados para uma mesma entidade, sejam esses mapeamentos definidos para a mesma fonte de dados ou para fontes de dados distintas. Para isso, cada operação sobre os dados e cada fonte de dados recebem nomes exclusivos que podem ser utilizados dentro do programa, dispensando o conhecimento do seu funcionamento no momento da utilização. Essa estratégia também permite que novas fontes de dados e novas funcionalidades sejam adicionadas ou removidas livremente, sem qualquer necessidade de alterações no código do programa principal.

Para endereçar os objetivos propostos, o primeiro problema a ser solucionado foi a abstração da localização da fonte de dados. Essa localização pode ser um endereço de rede, um caminho de arquivo, um endereço de memória, uma URL, entre outras possibilidades. Para cada um desses casos é necessário um tratamento específico que deve levar em consideração não apenas a forma de localização do recurso, mas também o método de acesso correspondente a cada fonte de dados. A solução encontrada para contornar esse problema foi a utilização do conceito de *Driver*, que é amplamente utilizado em sistemas operacionais (SILBERSCHATZ 2010).

Um *driver* consiste em um componente de *software* que pode ser acoplado ou desacoplado conforme a conveniência. Para isso ele deve implementar um conjunto de interfaces padronizadas (APÊNDICE) através das quais as operações de entrada e saída podem ser executadas de maneira uniforme. Cada *driver* permite a interação com mecanismos

especializados sem que seja necessário o conhecimento do seu funcionamento interno.

No FrameDOD, os *drivers* são responsáveis pela padronização das fontes de dados, encapsulando as operações de baixo nível e disponibilizando os mecanismos necessários ao tratamento das instruções de entrada e saída. Os *drivers*, no entanto, não são suficientes para tratar todo o problema, pois as informações sobre onde está a fonte de dados e quais os parâmetros necessários para acessá-la precisam ser definidas em algum lugar. Para isso foi criada a HOML, que fornece informações a todas as camadas da arquitetura. Essa linguagem permite declarar os parâmetros específicos de cada *driver*, os mapeamentos que definem como cada entidade interage com as fontes de dados e as referências a recursos externos que devem ser agregados ao arcabouço.

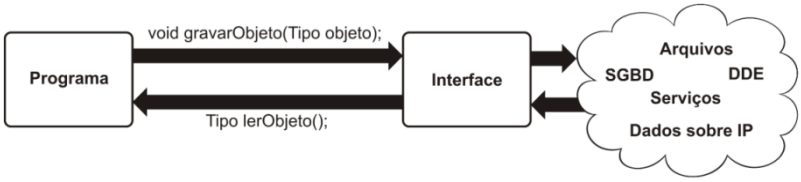
Todas as operações definidas em HOML representam árvores de sintaxe abstratas (ASTs) que são analisadas e processadas pelo *driver* correspondente à fonte de dados que está sendo utilizada. Cada *driver* deve possuir como característica básica a capacidade de analisar um conjunto mínimo de instruções, porém não precisa se limitar a esse conjunto. Isso significa que instruções adicionais podem ser incorporadas, permitindo que novos recursos sejam agregados à linguagem. As ASTs funcionam, portanto, como instruções que o *driver* deverá executar, produzindo os resultados definidos na documentação da linguagem. Esses resultados podem ser as operações propriamente ditas (executadas diretamente na fonte de dados) ou uma linguagem de consulta secundária que poderá ser processada pelo mecanismo de persistência utilizado.

Na rdtrutura básica do FrameDOD, todos os dados do sistema são representados por objetos definidos pelo programador. Esses objetos podem ser lidos ou gravados em fontes de dados específicas através de interfaces de dados genéricas e a interação com essas interfaces ocorre através de métodos (verbos) que identificam cada operação a ser executada. Cada verbo possui uma definição em HOML e sua semântica é depende do *driver* responsável pelo seu processamento, respeitando-se os lexemas e semânticas básicos já definidos na linguagem. Para cada verbo existe um conjunto específico de mapeamentos que instruem o *driver* sobre como os dados deverão ser processados na fonte de dados específica. Esses mapeamentos podem ser inferidos dinamicamente (a partir dos objetos) definidos individualmente (para cada método) ou definidos de modo genérico (para múltiplos *drivers*), conforme as especificidades de cada projeto.

Para ilustrar o funcionamento da estrutura proposta, podemos retomar o cenário já apresentado na Figura 8 e reproduzido na Figura 11. Nesse exemplo temos dois métodos: **gravarObjeto(objeto)** e **lerObjeto()**. Quando o método **lerObjeto** é chamado, uma interface

de dados específica é utilizada para carregar uma AST (do inglês *Abstract Syntax Tree*) a partir do arquivo HOML.

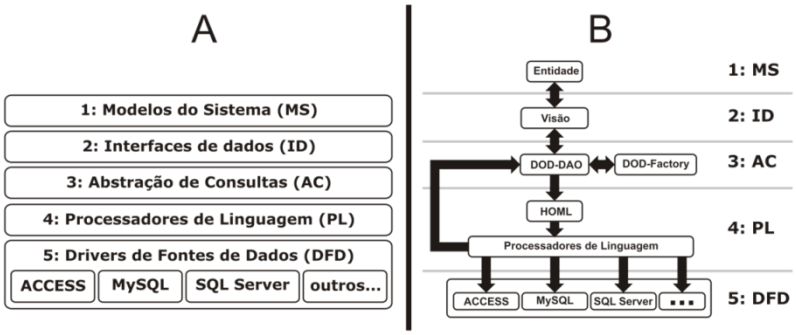
Figura 11 - Interação entre programa e fonte de dados no modelo DOD



Fonte: autoria própria (2015)

Cada interface de dados está associada com uma fonte de dados específica e a AST representa a operação a ser executada nesta fonte de dados. Para isso, a AST é passada ao montador de consulta para que sejam adicionados quaisquer parâmetros necessários antes de enviá-la ao estágio seguinte.

Figura 12 - Arquitetura do FrameDOD (reprodução da Figura 10)



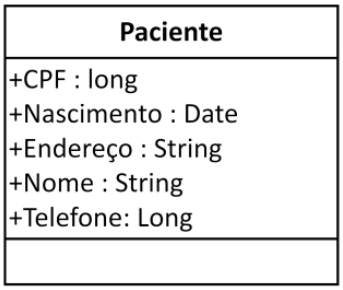
Fonte: autoria própria (2015)

Depois de montada, a AST é encaminhada para o processador de linguagem (item 4 da Figura 12), que se encarrega da sua interpretação e pela execução das ações apropriadas sobre a fonte de dados. Após o processamento, um conjunto de dados brutos (Registros) é obtido da fonte de dados e enviado à fábrica de objetos. Essa fábrica recorre novamente aos mapeamentos definidos em HOML para construir e popular instâncias das entidades apropriadas. Os objetos resultantes são então devolvidos à interface que originou a requisição e finalmente entregues na origem da chamada, concluindo o processamento da operação.

Na arquitetura apresentada (Figura 12), as classes de dados definidas pelo programador são denominadas **Modelos do sistema**, servindo para representar as entidades que fazem parte do processo que o *Software* busca reproduzir. Para entender melhor o que os modelos do sistema significam, podemos tomar como exemplo uma versão simplificada de um sistema de controle de consultas médicas, onde a classe Paciente (Figura 13) é um

exemplo de entidade da camada de Modelos do Sistema e não precisa possuir nenhuma característica especial para garantir a sua persistência (padrão *Database Broker*). Por exemplo: se uma instância da classe Paciente precisa ser gravada, uma segunda entidade cuidará dessa operação, analisando a estrutura da instância de Paciente e replicando seus dados no repositório. A classe Paciente seria totalmente passiva nesse processo. Isso significa que nenhum método seria criado na classe para tratar qualquer aspecto específico da sua persistência.

Figura 13 - Classe Paciente



Fonte: autoria própria (2015)

Para que as entidades dos modelos do sistema possam ser lidas ou gravadas em uma fonte de dados, é necessária a existência de entidades capazes de reconhecer e manipular cada modelo, além de permitir que o programador especifique a operação que deseja executar. As **Interfaces de dados** são estruturas responsáveis por essa tarefa e utilizam as definições em HOML para cumprir o seu papel. É importante não confundir o conceito de interface de dados com o conceito de interface de programação. Para entender o seu funcionamento, utilizaremos novamente a classe Paciente. Assumiremos que essa entidade será mapeada para um banco de dados e deve suportar a realização de quatro operações básicas: Inclusão, Exclusão, Alteração e Consulta.

Em um arquivo HOML, as fontes de dados e os modelos recebem nomes não ambíguos. As fontes de dados e os modelos são utilizados em contextos diferentes, de modo que nunca haverá ambiguidade entre eles e uma fonte de dados pode ter o mesmo nome de um modelo. Para cada fonte de dados é possível especificar um conjunto de operações associadas com cada um dos modelos declarados. Essas operações recebem nomes não ambíguos denominados **Verbos**, que são utilizados para solicitar a sua execução. Isso significa que verbos definidos para o mesmo modelo dentro da mesma fonte de dados não podem ter o mesmo nome, porém o mesmo verbo pode ter definições diferentes desde que seja em fontes de dados ou modelos distintos.

Figura 14 - Exemplo de mapeamento HOML detalhado para o modelo Paciente

```
<mappings>
 < Fonte1>
  <Paciente>
   <attributes>
    <attribute name="CPF" field="CPF" keytype="primary"/>
    <attribute name="Nascimento" field="nascimento"/>
    <attribute name="Endereco" field="endereco"/>
    <attribute name="Nome" field="nome"/>
    <attribute name="Telefone" field="telefone"/>
   </attributes>
   <source entity="tabela_pacientes"/>
   <operations>
    <select/>
    <insert/>
    <delete/>
    <update/>
   </operations>
  </Paciente>
 </Fonte1>
</mappings>
```

Fonte: autoria própria (2015)

No exemplo proposto, temos uma fonte de dados e um modelo, chamados respectivamente Fonte1 e Paciente. Para o modelo Paciente, na fonte de dados Fonte1, foram definidos os verbos Insert, Update, Delete e Select, utilizando o mapeamento apresentado na Figura 14. Este exemplo demonstra a utilização da forma completa do mapeamento, porém pode ser observado que os atributos da classe mapeada correspondem exatamente aos nomes dos campos na respectiva tabela e que as operações definidas correspondem às operações básicas.

Figura 15 - Exemplo de mapeamento HOML simplificado para o modelo Paciente

```
<mappings>
 < Fonte1>
  <Paciente>
   <attributes loadfrommodel="true">
    <attribute name="CPF" keytype="primary"/>
   </attributes>
   <source entity="tabela_pacientes"/>
   <operations template="crud"/>
  </Paciente>
 </Fonte1>
</mappings>
```

Fonte: autoria própria (2015)

O caso ilustrado é chamado de trivial, pois os atributos podem ser inferidos automaticamente a partir da estrutura da classe e as operações podem ser definidas a partir de um modelo padrão, dispensando a definição detalhada. Dessa forma, podemos utilizar uma versão simplificada do mapeamento apresentado (Figura 15), podendo ser resumido ao mínimo necessário para que sejam especificadas características especiais, tais como chaves, índices e outras. Em relação às operações, o modelo CRUD (*Create, Read, Update, Delete*) (SILVA, 2011) especifica que as operações básicas de inclusão, exclusão, consulta e alteração estarão disponíveis, sem a necessidade de declarações individuais.

É importante observar que, no exemplo apresentado, não foi necessário especificar

como os verbos Insert, Update, Delete e Select serão processados. Isso se deve ao fato desses verbos possuírem semântica predefinida, de modo que a sua simples declaração já é suficiente para que o mecanismo determine como processá-las. O programador poderá especificar comportamentos personalizados para esses verbos, mas não é obrigado a fazê-lo.

Uma vez que os modelos, as fontes de dados e os verbos tenham sido declarados, a interação com a interface de dados pode ocorrer de duas formas distintas. A primeira forma consiste em utilizar uma interface de dados genérica que recebe objetos como argumentos e devolve objetos como resposta (classe Object). Este método requer que sejam especificados, no momento da utilização, que tipo de objeto deve ser instanciado e o nome da fonte de dados a ser consultada.

Figura 16 - Exemplo de uso de interface de dados generalizada

```
List<object> pacientes = HomlDao.Read(typeof(Paciente), "Fonte1", "Select");
foreach (object paciente in pacientes)
{
    Console.Write(((Paciente)paciente).CPF);
}
```

Fonte: autoria própria (2015)

A Figura 16 ilustra a leitura de uma lista de objetos do tipo Paciente, oriundos da fonte de dados “Fonte1”, através de uma interface de dados genérica. Esse método é simples, pois não requer a criação de entidades especializadas, porém exige o uso de coerção (*casting*) para garantir a compatibilidade de tipos, dependendo da linguagem de programação utilizada.

Figura 17 - Exemplo de uso de interface de dados especializada

A	`List<Paciente> pacientes = PacienteDao1.Select();`
	`foreach (Paciente paciente in pacientes)`
	`{`
	`    Console.Write(paciente.CPF);`
	`}`

B	`public class PacienteDao1 : HomlDao<Paciente>`
	`{`
	`    public PacienteDao1() : base("Fonte1") { }`
	`}`

Fonte: autoria própria (2015)

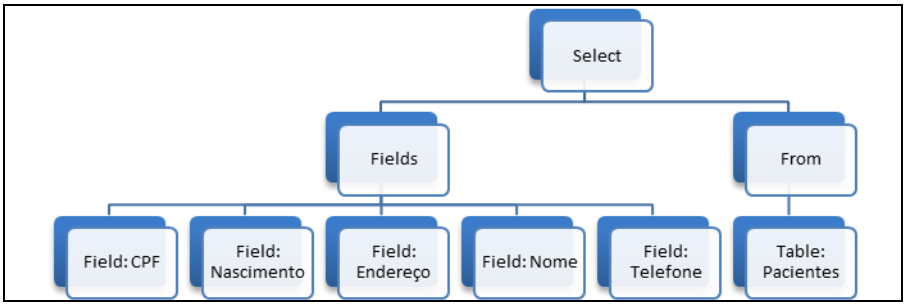
A segunda forma de leitura de dados consiste em criar classes especializadas a partir de protótipos (*Generics*), permitindo que os mapeamentos entre o modelo e a fonte de dados sejam feitos de maneira estática (Figura 17 A). Nesse caso, o nome da fonte de dados a ser utilizada pode ser definido na própria classe que especializa a interface de dados (Figura 17 B). Esses exemplos não têm como propósito apresentar todo o potencial do FrameDOD, mas ilustram quanto código é necessário para a sua utilização. Caso fosse persistida uma entidade mais complexa, esse código não seria alterado, pois todo o mapeamento é feito nos arquivos HOML e não no código.

Para cada objeto da camada de modelos haverá uma ou mais entidades na camada de

interfaces, cada uma delas contento um ou mais verbos que representarão operações específicas. Cada uma dessas operações pode interagir com uma ou mais fontes de dados, conforme a sua definição em HOML. Isso significa que é possível utilizar uma mesma operação para produzir efeitos em várias fontes de dados simultaneamente, mesmo que essas fontes de dados sejam tratadas por *drivers* distintos.

Conforme pode ser observado na arquitetura apresentada, as interfaces de dados fazem uso dos serviços da camada de **Abstração de Consultas**. Essa camada permite que o mecanismo traduza as operações solicitadas através das interfaces de dados em ações na fonte de dados e também permite a tradução dos dados resultantes para os tipos de dados esperados por cada interface.

Figura 18 - AST gerada pela operação SELECT relativa à entidade PACIENTE



Fonte: autoria própria (2015)

As operação são tratadas por duas entidades conceituais: Dod-DAO e DOD-Factory. A entidade DOD-DAO atua incorporando os dados na AST HOML (Figura 18) para serem utilizados como parâmetros para a operação a ser executada A árvore resultante é encaminhada ao processador de linguagem que analisará a operação representada e executará a ação correspondente. Caso a ação executada devolva dados como resposta, eles serão tratados pela entidade DOD-Factory, responsável pela transformação dos dados brutos em entidades específicas, sejam elas simples ou compostas. As informações necessárias para essa transformação podem ser definidas no arquivo HOML ou inferidas automaticamente a partir da estrutura de dados para a qual a interface de dados foi construída. Para ilustrar essa situação, tomaremos novamente o exemplo da Figura 17. A execução dessa operação resultaria na criação de uma AST conforme apresentado na Figura 18.

O processador de linguagem analisa a AST recebida da camada de abstração de consultas e seleciona os *drivers* que serão responsáveis pelo efetivo processamento das requisições, chamando os métodos apropriados para o processamento requisitado. Esse processamento poderá ou não trazer resultados que, se existirem, serão encaminhados de volta à entidade requisitante (DOD-DAO). O processo de seleção da fonte de dados ocorre no

momento em que o programador escolhe a Interface de dados que será chamada. Isso significa que para utilizar uma fonte de dados diferente ou um conjunto de mapeamentos diferente basta utilizar outra interface de dados.

3.5 EXTENSIBILIDADE

Conforme visto na seção anterior, o FrameDOD interage com as fontes de dados através de *drivers* que definem a semântica das expressões e executam as operações correspondentes. Alguns desses *drivers* são fornecidos como parte integrante do *framework*, porém o programador pode criar e agregar ao arcabouço existente os seus próprios *drivers* com quaisquer funcionalidades adicionais desejadas. Esse tipo de flexibilidade exigiu a criação de uma solução para permitir que as funcionalidades novas e as antigas pudessem coexistir sem conflitos ou incompatibilidades.

Para entender melhor o problema, podemos supor que o programa precise obter dados de um dispositivo GPS e armazená-los em um banco de dados relacional. Os dois mecanismos possuem funcionamentos radicalmente distintos. O GPS produz dados de forma contínua e utiliza comunicação byte a byte através de uma porta serial com envio e recebimento de dados brutos que possuem semântica própria e estrutura que depende do aparelho GPS utilizado. Os parâmetros de configuração incluem o número da porta serial, quantidade de bits de dados, bits de parada, algoritmo de controle de fluxo e taxa de transmissão. O banco de dados relacional possui interação através de expressões SQL e os parâmetros de conexão incluem o nome e a senha do usuário, o endereço do servidor e outros elementos. Os dados oriundos desta fonte possuem estrutura definida pelo programador ou pelo DBA. O acesso ao banco de dados relacional ocorre por meio de requisições que partem do cliente para o servidor, enquanto o acesso ao GPS é dinâmico, ocorrendo por meio de notificações que ocorrem sempre que novos dados estão disponíveis.

Fica claro com o exemplo apresentado que cada fonte de dados requer um tratamento especializado, porém o objetivo nos dois casos é exatamente o mesmo: os dados. No caso do GPS, o *driver* intercepta todas as notificações, mantendo um buffer com os dados em uma estrutura padronizada. Quando o programa requisitar esses dados, o *driver* simplesmente os devolverá conforme a demanda. No caso do banco de dados relacional, não existe a necessidade de um buffer e os dados requisitados serão simplesmente lidos sob demanda. Todo esse tratamento ocorre dentro de cada *driver* e as únicas informações que precisam ser

definidas no HOML são os parâmetros de conexão.

Cada *driver* é responsável pelo processamento dos seus parâmetros específicos, de modo que o programa não precise entender ou mesmo saber que esses parâmetros existem na hora de utilizar a fonte de dados dentro do código. Uma vez que um verbo seja enviado, caberá ao *driver* processar e devolver a resposta correspondente, analisando os mapeamentos definidos e transformando os dados de maneira apropriada. Os mesmos objetos obtidos a partir das requisições feitas ao GPS podem ser diretamente enviados ao SGBD onde eles terão outra semântica e outro conjunto de mapeamentos que permitirão a sua persistência de maneira transparente.

Cada *driver* possui seu próprio conjunto de lexemas, porém todos eles seguem um padrão estrutural que permite a sua integração. Isso significa que todos os *drivers* devem ser capazes de reconhecer um conjunto básico de lexemas cuja semântica é padronizada e não deve ser sobrescrita, garantindo a compatibilidade entre os *drivers*. Essas características serão discutidas na próxima seção deste capítulo.

3.6 ESTRUTURA DA HOML

A HOML tem sua estrutura baseada em XML e seus lexemas básicos são inspirados no modelo relacional e nos elementos da SQL. As expressões em HOML são compostas por elementos declarativos, que informam quais fontes de dados estão sendo utilizadas, quais *drivers* fornecem o suporte para a manipulação de cada fonte de dados, quais entidades são mapeadas e como essas entidades são lidas e gravas em cada fonte de dados (operações).

Para desempenhar as funções propostas, quatro seções básicas são definidas, conforme apresentado na Figura 19:

a) cabeçalho;
b) referências;
c) declarações;
d) mapeamentos.

O cabeçalho tem propósito informativo e serve para documentação do arquivo. Ele permite que o usuário defina suas próprias marcas de forma livre, respeitando o padrão XML, sem nenhum tipo de semântica predefinida. As demais seções do documento permitem declarar as bibliotecas, *drivers* e fontes de dados que serão utilizadas, definir agrupamentos para simplificação dos mapeamentos, declarar as entidades que serão mapeadas, definir quais

operações estarão disponíveis e determinar as regras de processamento dessas operações.

Figura 19 - Exemplo genérico de documento HOML

```
<?xml version="1.0" encoding="utf-8" ?>
<homl  version="1.0">
  <header>
    <description>Documento de exemplo</description>
  </header>
  <references>
    <drivers><!--Lista de drivers que estão sendo declarados--></drivers>
  </references>
  <declarations>
    <datasources><!--Lista de fontes de dados a serem mapeadas com os drivers declarados--></datasources>
    <sets><!--Grupos personalizados para referenciar múltiplas fontes de dados em conjunto--></sets>
    <entities><!--Lista de entidades que estão sendo mapeadas--></entities>
  </declarations>
  <mappings>
    <all><!—Definições de mapeamentos para todas as fontes de dados--></all>
    <setName><!-- Definições de mapeamentos para um grupo de fontes de dados personalizado --></setName >
    <DataSourceName><!-- Definições exclusivas para uma fonte de dados específica--></DataSourceName >
  </mappings>
</homl>
```

Fonte: autoria própria (2015)

A primeira seção funcional do documento contém a área de referências e permite declarar as bibliotecas externas e os *drivers* que serão utilizados. Bibliotecas são arquivos que podem conter *drivers*, modelos e outros elementos necessários para que as declarações em HOML funcionem corretamente. Para isso, elas precisam ser declaradas através da marca “library”, onde o atributo “path” especifica o caminho absoluto ou relativo da biblioteca no sistema de arquivos. O caminho absoluto deve conter o endereço completo do arquivo desde a raíz do diretório, enquanto o caminho relativo deve conter o endereço parcial, partindo da localização do arquivo HOML. Os dois casos são ilustrados na Figura 20.

Figura 20 - Declaração de bibliotecas em HOML

```
<references>
  <libraries>
    <library path="c:\drivers\MySQL.dll"/>
    <library path="subpasta\GPS.dll"/>
  </libraries >
</references>
```

Fonte: autoria própria (2015)

Uma vez que as bibliotecas tenham sido declaradas, é necessária a utilização da marca “driver” para especificar cada um dos *drivers* que serão utilizados, conforme ilustrado na Figura 21.

A declaração de um *driver* requer a especificação dos atributos “name” e “class”. O atributo “name” permite especificar um nome não ambíguo para cada *driver* declarado, não sendo possível utilizar o mesmo nome para dois *drivers* distintos. O atributo “class” permite especificar o nome plenamente qualificado (incluindo o *namespace* completo) da classe que implementa o *driver*, conforme definido na biblioteca que o contém. É possível a declaração

do mesmo *driver* mais de uma vez, porém esse tipo de procedimento seria desnecessário.

Figura 21 - Declaração de drivers em HOML

```
<references>
  <drivers>
    <driver name="MySQL" class="meudriver.MySQL.Driver"/>
    <driver name="GPS" class="meudriver.GPS.Driver"/>
  </drivers>
</references>
```

Fonte: autoria própria (2015)

Cada um dos *drivers* declarados na seção de referências pode ser utilizado para declarar tantas fontes de dados quanto se julgar necessário, conforme ilustrado na Figura 22. Para isso é utilizada a marca “datasource”, dotada dos atributos “name” e “driver”. O atributo “name” permite a identificação de uma fonte de dados de forma exclusiva. Isso significa que o mesmo nome não pode ser usado para mais de uma fonte de dados, porém nada impede que a mesma fonte de dados seja declarada mais de uma vez. Esse tipo de procedimento permite que o mesmo modelo possua mais de uma definição dentro da mesma fonte de dados, dando ao programador a opção de reaproveitá-lo em mais de um contexto sem a necessidade de declarar mais de uma versão da mesma classe. O atributo “driver” permite escolher um dos *drivers* já referenciados e utilizá-lo para processar a fonte de dados que está sendo declarada.

Figura 22 - Exemplo de mapeamento de fontes de dados em HOML

```
<datasources>
  <datasource name="MySQL1" driver="MySQL">
    <attribute name="server" value="localhost"/>
    <attribute name="userid" value="usuario"/>
    <attribute name="password" value="senha"/>
    <attribute name="database" value="banco_de_dados"/>
  </datasource>
  <datasource name="MySQL2" driver="MySQL"/>
    <attribute name="server" value="localhost"/>
    <attribute name="userid" value="usuario"/>
    <attribute name="password" value="senha"/>
    <attribute name="database" value="banco_de_dados"/>
  </datasource>
  <datasource name="GPS1" driver="GPS"/>
    <attribute name="port" value="COM1"/>
    <attribute name="protocol" value="NMEA 0183"/>
    <attribute name="databits" value="8"/>
    <attribute name="stopbits" value="0"/>
    <attribute name="parity" value="none"/>
    <attribute name="baudrate" value="115200"/>
  </datasource>
</datasources>
```

Fonte: autoria própria (2015)

A marca “datasource” possui como filhas um número indeterminado de marcas do tipo “attribute”, que permitem especificar os parâmetros de configuração necessários para que se estabeleça a conexão com a fonte de dados declarada. Cada *driver* distinto pode ter seu próprio conjunto de atributos, definidos pelo criador do *driver*. Esses atributos são

documentados no próprio *driver*, que precisa incluir a capacidade de enumerá-los de forma automatizada. Essa enumeração deve incluir a descrição de cada atributo, indicando um a um quais deles são obrigatórios e quais são opcionais. Esse recurso permite que o programador possa utilizar uma ferramenta para auxiliá-lo na criação dos mapeamentos, sem a necessidade de conhecer previamente o *driver* que está utilizando.

Figura 23 - Declaração de grupos de fontes de dados em HOML

```
<sets>
 < set name="Grupo1">
    <item name="MySQL1"/>
    <item name="MySQL2"/>
 </set>
</sets>
```

Fonte: autoria própria (2015)

Para evitar o uso de declarações redundantes, a HOML permite que múltiplas fontes de dados sejam organizadas em grupos, viabilizando a criação de mapeamentos compartilhados (Figura 23). É possível atribuir qualquer nome a um grupo, exceto pela palavra reservada “All”, que possui semântica própria e faz referência a um grupo predefinido que representa todas as fontes de dados declaradas. Não é permitida também a utilização de nomes que já tenham sido atribuídos a outros grupos e/ou fontes de dados, pois os grupos e as fontes de dados são utilizados nos mesmos contextos. Isso possibilita a criação um grupo cujos itens são combinações de outros grupos com fontes de dados individuais. A única restrição imposta nesse agrupamento é que somente itens já declarados podem ser agrupados. Isso significa que um grupo só pode ser um elemento de outro grupo se ele for declarado primeiro e que um grupo não pode ser incluído como item dele mesmo, o que também proíbe a declaração do grupo especial “All” como membro de qualquer outro grupo. Essa restrição não se aplica às fontes de dados individuais, pois elas são analisadas antes dos grupos, independentemente da ordem de declaração.

O último elemento na sessão de declarações consiste em uma lista as entidades que serão mapeadas. Essas entidades são declaradas, uma a uma, através da marca “entity”, cujos atributos principais são “name” e “class”. O atributo “name” deve conter uma identificação não ambígua para cada entidade, permitindo a simplificação do seu uso ao longo do documento HOML. O atributo “class” deve ser o nome plenamente qualificado para a classe correspondente à entidade que está sendo declarada. Eventualmente pode ser necessária a declaração de uma interface (interface de programação) ou uma classe abstrata, entidades que não podem ser instanciadas. Nesse caso, o atributo “instanceclass” pode ser utilizado para especificar uma classe derivada cuja instância possa ser criada. Um exemplo de declaração é ilustrado na Figura 24, onde são declaradas as entidades “meuprojeto.entidades.Consulta” e

“meuprojeto.entidades.IPaciente”, sendo a última uma interface cuja classe concreta a ser instanciada é “meuprojeto.entidades.IPaciente”. É importante observar que não é necessário especificar de que biblioteca as referências se originam, pois todas as bibliotecas que serão utilizadas já foram declaradas na seção “libraries”.

Figura 24 - Declaração de entidades em HOML

```
    <entities>
      <entity name="Usuario" class="meuprojeto.entidades.IPaciente" instanceclass="meuprojeto.entidades.Paciente"/>
      <entity name="Location" class="meuprojeto.entidades.Consulta"/>
    </entities>
</declarations>
```

Fonte: autoria própria (2015)

Conforme demonstrado, cada entidade deve receber um nome exclusivo, porém esse nome não precisa ser o mesmo da classe declarada. O nome tem como propósito referenciar a classe dentro da estrutura do HOML, porém não afeta o funcionamento do mecanismo. Não é permitido atribuir dois nomes diferentes para a mesma classe, porém é possível definir diversos mapeamentos para ela, bastando que esses mapeamentos sejam relacionados a fontes de dados declaradas com nomes diferentes, ainda que façam referência ao mesmo destino.

Uma vez que todas as referências e declarações tenham sido feitas os mapeamentos podem ser criados, porém existe a possibilidade de mapeamentos distintos feitos para a mesma entidade entrem em conflito entre si. Por exemplo: se um mapeamento X1 for feito para a fonte de dados F1 e outro mapeamento diferente for feito para o grupo G1, que contem F1, a fonte de dados F1 herdará de G1 esse novo mapeamento, porém eles estariam em conflito. Nessa situação o critério de solução leva em consideração a especificidade do mapeamento e a sua precedência. Isso significa que uma definição de mapeamentos feita para uma fonte de dados tem precedência sobre outra que tenha sido feita para o grupo no qual essa fonte de dados foi inserida. Caso o critério da especificidade não possa ser utilizado, prevalece aquele que for definido primeiro. Tomemos como exemplo uma fonte de dados F2 que esteja contida nos grupos G2 e G3. Se os mapeamentos para uma mesma entidade E1 forem definidos em G2 e em G3, eles estarão em conflito, porém nenhum dos dois será mais específico que o outro. Nesse caso, prevalece aquele que for declarado primeiro.

A Figura 25 apresenta um exemplo de mapeamento da entidade “Paciente” para o grupo de fontes de dados “Grupo1”. Neste exemplo foram declaradas as operações “select”, “insert”, “update” e “delete”, porém não foram especificadas como essas operações serão processadas. Tais operações já possuem semântica padronizada, não sendo necessária a definição detalhada de como deverão funcionar, exceto quando o comportamento desejado for diferente do padrão. Quando uma operação não possuir semântica padronizada, ela deverá ser

processada pelo driver, que responderá de forma apropriada ou disparará uma exceção indicando que a operação não foi definida.

Figura 25 - Exemplo mapeamento de entidades em HOML

```
<mappings>
 < Grupo1>
  <Paciente>
   <attributes loadfrommodel="true">
    <attribute name="CPF" keytype="primary"/>
   </attributes>
   <source entity="tabela_pacientes"/>
   <operations>
    <select/>
    <insert/>
    <delete/>
    <update/>
   </operations>
  </Paciente>
 </ Grupo1>
</mappings>
```

Fonte: autoria própria (2015)

No exemplo ilustrado na Figura 25, a marca “source” é opcional e permite especificar uma entidade com a qual o mapeamento está associado, neste caso uma tabela chamada “tabela_paciente”.

Figura 26 - Definição de operações personalizadas em HOML

```
<mappings>
  <Grupo1>
    <Paciente>
      <attributes loadfrommodel="true">
        <attribute name="CPF"
keytype="primary"/>
      </attributes>
      <operations>
        <select>
          <from entity="tabela_pacientes1"/>
        </select>
        <insert>
          <into entity="tabela_pacientes2"/>
        </insert>
        <delete>
          <from entity="tabela_pacientes3"/>
        </delete>
        <update>
          <set entity="tabela_pacientes4"/>
        </update>
      </operations>
    </Paciente>
  </Grupo1>
</mappings>
```

Fonte: autoria própria (2015)

Uma alternativa de declaração seria a definição individual cada operação, o que permitiria que cada operação fosse associada com uma entidade diferente. Um exemplo dessa situação é ilustrada na Figura 26. Nesse exemplo alguns comportamentos automáticos ainda são utilizados, pois o mecanismo será capaz de inferir como os campos deverão ser associados e/ou usados como critérios na operação. Isso se deve à utilização do atributo “loadfrommodel” e à ausência de declarações específicas para esses comportamentos.

A semântica padrão assume que o nome da entidade é o mesmo na memória e na fonte de dados e a mesma regra vale para os nomes dos seus atributos. Isso é feito para minimizar o trabalho durante a definição das declarações, permitindo que algumas sejam automatizadas. Seguindo a mesma lógica, as operações Insert, Delete, Update e Select possuem semânticas compatíveis com respectivas operações no modelo relacional. Essas definições básicas permitem simplificar o mapeamento das entidades para casos triviais, poupando o usuário da criação de expressões detalhadas para cada operação. Mesmo em mapeamentos mais detalhados, muitos comportamentos padronizados ainda são utilizados, permitindo a redução do número de detalhes que precisam ser especificados.

Além de definir um conjunto padronizado de semânticas, a HOML define também algumas regras para lidar de forma simples com os mais diversos casos de mapeamentos, permitindo definir todas as operações do modelo relacional utilizando um conjunto mínimo de declarações. Tais facilidades criam um ambiente que simplifica o uso ferramentas capazes de automatizar o processo de construção dos mapeamentos, permitindo que a estrutura da fonte de dados seja inferida a partir de um documento HOML ou que um documento HOML seja inferido a partir de uma fonte de dados. Isso permite reduzir sensivelmente o esforço de programação e pode facilitar a distribuição do *software* produzido.

3.7 ESTRUTURA SINTÁTICO-SEMÂNTICA

A sintaxe dos elementos declarados em HOML segue uma hierarquia determinada pela forma como cada elemento é declarado, de acordo com a estrutura de um documento XML. A maioria das operações padronizadas possui inspiração direta no modelo relacional, porém segue uma sintaxe encadeada típica de um documento XML. Os lexemas utilizados tentam refletir, portanto, as mesmas instruções que seriam utilizadas em uma expressão SQL. Por exemplo, a marca “SELECT” possui como subitens as marcas “FROM” e “WHERE”, do mesmo modo a marca “INSERT” possui como subitens as marcas “INTO” e “WHERE”. Essa característica busca trazer familiaridade para a linguagem, reduzindo a curva de aprendizado do usuário, já que apenas os elementos novos precisarão ser aprendidos.

Cada elemento utilizado nos mapeamentos possui uma semântica padronizada, de modo que o seu funcionamento é regido por algumas regras que estabelecem como o mecanismo deverá se comportar na ausência de determinadas declarações. Por padrão, o mecanismo assume que todos os atributos das entidades mapeadas precisam ser especificados,

porém o programador pode optar por permitir que o mecanismo descubra esses atributos de forma automática (atributo “loadfrommodel” igual a “true”). No caso das operações, o mecanismo é preparado de forma que todas as operações do modelo relacional sejam tratadas conforme o comportamento esperado. Por exemplo, a marca “INSERT” assume que o usuário fornecerá dados que deverão ser usados para criar um novo item na fonte de dados, a marca “SELECT” assume que os dados serão lidos da fonte de dados e devolvidos para entidade solicitante. Essas e outras operações possuem conjuntos predefinidos de atributos e subitens que não podem ser sobrescritos pelos *drivers* responsáveis pelo seu processamento.

A HOML tem como premissa básica a regra de que a capacidade de expressão do programador não pode ser prejudicada. Isso significa que a linguagem deve possuir um conjunto básico de funcionalidades que visam facilitar a utilização dos seus recursos, mas que também deve ser possível agregar conjuntos adicionais de funcionalidades à linguagem, ampliando aquilo que pode ser feito. Para isso, é definido um processador de linguagem padrão capaz de lidar com todos os elementos básicos da HOML, porém cada *driver* pode ser dotado de um processador de linguagem complementar que será chamado para tratar qualquer elemento desconhecido. Desse modo, além das marcas definidas na forma padrão da HOML, cada *driver* pode adicionar novos elementos à gramática. Durante o processo de análise, toda a estrutura do documento é carregada para a memória e todos os elementos padronizados são analisados e validados. Os elementos restantes são encaminhados para cada driver, até que tudo tenha sido validado. Se algum elemento não for aceito durante o processo, todo o documento será invalidado e os elementos desconhecidos serão enumerados.

3.8 TESTES DE CAMPO

Para validar e aprimorar o trabalho desenvolvido, cada etapa do processo de criação do FrameDOD foi testada em projetos reais das empresas Grupo Terra Firme, Conectta *Softwares* e DAC Sistemas. Isso permitiu identificar e corrigir diversas falhas, além de aprimorar a experiência de uso do *software* resultante.

A primeira etapa do estudo foi desenvolvida no Grupo Terra Firme utilizando a versão preliminar do *Framework*, baseada na utilização de 5 entidades básicas que precisavam ser criadas pelo programador com base em protótipos previamente definidos: Modelo, Dao, Factory, Filter e Adapter.

Apesar de ser baseada em protótipos, a classe de modelo precisava incorporar algumas

funcionalidades que exigiam a sua adaptação para se adequar ao padrão utilizado, utilizando uma forma híbrida entre os padrões *Expert* e *Database Broker*. Depois que as entidades eram criadas, a sua utilização era simples, porém o processo requeria algum trabalho até que as entidades básicas estivessem prontas para uso.

O cenário de utilização consistia na migração de um sistema que havia sido originalmente construído em Access. Esse programa era utilizado em ambiente de rede, compartilhado por 8 a 10 usuários simultâneos. Por causa dessa estrutura, era frequente a corrupção do banco de dados, com eventuais perdas que não podiam ser compensadas pela restauração das cópias de segurança. A empresa tinha como pretensão a migração de todo o sistema para uma interface Web, desenvolvida em VB.Net com banco de dados SQL Server, porém algumas características do sistema e a frequência de uso impediam a migração em uma etapa. Para lidar com o problema, foi adotada uma solução híbrida, na qual o sistema foi migrado em etapas graduais.

O primeiro passo da migração consistiu na construção das funcionalidades essenciais do sistema, de modo o banco de dados pudesse ser transferido para o SQL Server, permitindo a execução das etapas seguintes. Nesse cenário, o mecanismo de acesso ao banco de dados precisaria interagir com o Access e o SQL Server ao mesmo tempo, constituindo um ambiente ideal para a aplicação do FrameDOD. Após a conclusão da primeira etapa, todas as funcionalidades do sistema foram gradualmente migradas até o banco de dados Access deixar de ser usado.

Apesar das facilidades oferecidas pelo FrameDOD, sua utilização no projeto mostrou algumas deficiências da concepção inicial. A abordagem utilizada era baseada inteiramente em código, o que dificultava a manutenção e requeria uma preparação inicial lenta, ainda que não requeresse tanto código quanto seria necessário com a abordagem convencional. Essa abordagem também despertava certa resistência nos programadores que não estavam habituados com o uso do FrameDOD, pois isso exigia a dedicação de certo tempo com o aprendizado das estruturas e a compreensão da sua utilização.

Com base na experiência adquirida com o primeiro teste, o *framework* foi aprimorado e passou por modificações estruturais e funcionais profundas, culminando na criação da HOML. A primeira dessas melhorias consistiu em eliminar a necessidade de modificar a classe de modelo para adequá-la ao *framework*. Para isso, o conjunto de entidades básicas precisou ser alterado e aprimorado para permitir que as informações necessárias para garantir a persistência pudessem ser inferidas automaticamente. Ao final do processo, o número de entidades que precisavam ser definidas pelo usuário foi reduzido a duas: O modelo, que já não

precisava ser modificado, e o DAO, que passou a abstrair e construir dinamicamente as demais estruturas, sem necessidade de intervenção do programador.

A nova estrutura foi utilizada como base para o segundo teste, desenvolvido na empresa Conectta *Softwares*. O novo cenário de uso consistiu na criação de um *software* novo que precisava interagir com um banco de dados SQL Server remoto, dois bancos de dados SQL Server locais (parcialmente redundantes), um WebService e arquivos CSV. Os três bancos de dados possuíam alguns dados comuns que precisavam ser sincronizados conforme fossem processados. Os arquivos CSV eram fornecidos pelos clientes para integração com o banco de dados principal. O WebService tinha como propósito a comunicação com um sistema externo, hospedado em um servidor remoto.

Os dados obtidos a partir dos arquivos CSV precisavam ser importados para o banco de dados central (remoto) onde os resultados do processamento deviam ser armazenados. O sistema possuía dois bancos de locais dados redundantes para controle das operações. O primeiro era utilizado localmente para o controle individual de cada servidor. O segundo era compartilhado pelos servidores, permitindo a troca de informações entre eles. Isso permitia a recuperação do sistema em caso de falhas, evitando que dados já processados causassem o envio de mensagens duplicadas. Após o processamento com sucesso, o servidor principal era atualizado para que os dados não fossem reenviados para processamento, porém algumas falhas de comunicação poderiam ocorrer durante o processo. Nesse caso, os bancos de dados redundantes teriam cópias das informações já processadas que poderiam ser reenviadas ao servidor central.

Nesse cenário, o FrameDOD permitia que as mesmas entidades pudessem ser persistidas nas diversas fontes de dados com uma quantidade mínima de código. Com a nova estrutura, os modelos da regra de negócio não precisavam ser modificados e poucas entidades precisariam ser criadas: Modelo, Dao, DatabaseAdapter. Nessa altura do projeto, a HOML ainda consistia em um projeto futuro, porém as bases para sua implementação já faziam parte da estrutura do *framework*.

O DatabaseAdapter consiste em uma estrutura padronizada para comunicação com uma fonte de dados através de uma interface genérica. O DAO consiste em uma entidade capaz de montar as instruções que serão utilizadas pelo DatabaseAdapter para interagir com a fonte de dados. Para isso, cada DAO é especializado em um tipo específico de entidade (Modelo) que precisa ser passado como argumento nas operações de escrita ou como filtro em qualquer operação. A Figura 27 mostra a construção das duas entidades. À primeira vista, pode parecer que as duas classes estão incompletas, porém esse é todo o código necessário

para que o mecanismo funcione.

Figura 27 - Definição de operações personalizadas em HOML

```
public class MainDatabaseAdapter : SqlDatabaseAdapter
{
    public MainDatabaseAdapter() : base("MainDatabase") { }
}

public class ExampleDAO : SqlDao<ExampleModel, MainDatabaseAdapter>
{
}
```

Fonte: autoria própria (2015)

Toda a regra de processamento necessária está implementada nas classes bases que faziam parte do *framework* e serviram de alicerce na posterior definição da estrutura dos *drivers* utilizados pela HOML.

O cenário do segundo teste foi extremamente desafiador e exigiu a criação de inúmeras soluções para problemas que ainda não tinham sido experimentados. Entre os novos problemas, podemos citar o uso de Stored Procedures, Stored Functions, manipulação de entidades compostas complexas, relacionamentos complexos, entre outras situações. Essa experiência enriqueceu enormemente o trabalho e permitiu o aprimoramento das funcionalidades oferecidas.

Diferente da situação encontrada na versão anterior do FrameDOD, a nova versão foi bem aceita por usuários que ainda não haviam experimentado o *framework*. A simplicidade para sua utilização e a quantidade mínima de intervenções no código funcionaram como atrativos que estimularam o interesse dos programadores, não sendo notada qualquer estranheza ou dificuldade no uso. É importante mencionar que os programadores que tiveram acesso ao *framework* precisaram somente das funcionalidades mais básicas, não tendo contato com os recursos mais complexos. Por esse motivo, não foi possível avaliar o grau de dificuldade para utilização e compreensão desses recursos para programadores não familiarizados com a estrutura.

O terceiro teste teve como ambiente a DAC Sistemas, onde foi utilizada a versão mais recente do FrameDOD. Neste caso, o *framework* foi fornecido ao programador acompanhado apenas de pequenos exemplos de uso, sem que existisse qualquer contato anterior com a ferramenta. Esse experimento teve como propósito analisar as dificuldades que seriam apresentadas por um programador com pouca experiência e sem qualquer familiaridade com o FrameDOD. O programador em questão apresentou poucas dúvidas, e adaptou-se rapidamente à utilização do *framework*.

O cenário de utilização consistiu em um sistema de emissão nota de Conhecimento de

Transporte Eletrônico[9] (CT-e). A utilização do sistema tinha como base um banco de dados SQL Server instalado localmente. A utilização do *framework* nesse cenário resumiu-se ao sistema CRUD (SILVA, 2011) básico, acrescido de alguns *Stored Procedures*.

[9] http://www.cte.fazenda.gov.br/

4 ANÁLISE COMPARATIVA: *ENTITY FRAMEWORK* X FRAMEDOD X NHIBERNATE

Para analisar os resultados obtidos com o uso do FrameDOD, é necessário o estabelecimento de parâmetros de comparação. Para isso, os dois *frameworks* mais utilizados do mercado foram tomados como referência: o NHibernate (Implementação do Hibernate para o .Net *Framework*) e o *Entity Framework*. Para isso, foram consideradas as operações básicas do modelo CRUD (*Create, Read, Update, Delete*) (SILVA, 2011) e foram estabelecidos os critérios simplicidade, eficácia e representatividade, considerando os resultados equivalentes obtidos através dos *frameworks* comparados.

O critério da simplicidade considera quanto esforço é necessário para que se execute a operação desejada, considerando a quantidade mínima de código a ser escrito. O critério da eficácia considera se o objetivo pode ou não ser alcançado, mesmo que a quantidade de esforço necessário não seja ideal ou que seja necessário o uso de algum recurso externo ou ferramenta extra. A representatividade considera a capacidade de atingir os objetivos almejados sem a necessidade de recursos ou ferramentas adicionais. O padrão CRUD (SILVA, 2011) foi utilizado como base para a análise, por abranger o conjunto de operações mais comum em sistemas que lidam com armazenamento de dados, apesar de não abordar o conjunto completo de funcionalidades dos *frameworks*. Seria possível esgotar todas as possibilidades de cada *framework* para estabelecer uma comparação extensiva e detalhada, porém isso fugiria ao escopo do trabalho e consumiria um tempo excessivo.

Para facilitar a análise, a comparação foi dividida em três partes. A primeira abrange a configuração básica necessária ao uso de cada *framework*, incluindo o mapeamento das fontes de dados. A segunda abrange a execução das operações CRUD (SILVA, 2011). A terceira abrange outras características relevantes como o tratamento da interoperabilidade entre fontes de dados distintas (iguais ou diferentes entre si). Para as demonstrações, será utilizada como referência a mesma classe Paciente (Figura 13) utilizada em todos os exemplos dessa dissertação. Antes disso, no entanto, são necessárias algumas considerações preliminares. Todos os exemplos apresentados foram desenvolvidos no Visual Studio 2013 e algumas das informações podem não corresponder exatamente ao que pode ser encontrado em outros IDEs. O Visual Studio utiliza um arquivo de configuração padronizado conforme o tipo de projeto (Desktop ou Web). Uma vez que os exemplos apresentados foram desenvolvidos em

um aplicativo do tipo Desktop, utilizaremos o arquivo de configuração “App.config”.

4.1 CONFIGURAÇÃO BÁSICA DOS *FRAMEWORKS*

Antes de utilizar qualquer um dos *frameworks*, algumas configurações preliminares são requeridas. O conjunto específico de configurações depende de cada um deles e os elementos principais de cada abordagem são explicados e demonstrados abaixo, porém algumas seções do arquivo “app.config” são padronizadas pela plataforma, onde todos os exemplos apresentados são construídos (Figura 28). O bloco “configSections” permite informar ao ambiente que uma nova seção de configuração será definida. Não existe um limite de quantas seções novas podem ser criadas, ficando isso a critério do usuário. A declaração de uma nova seção é feita através do elemento “section”, cujo atributo “name” especifica qual será o nome da nova seção e o atributo “type” permite especificar qual será a biblioteca responsável pelo processamento dessa nova seção. No exemplo apresentado, foi criada a seção “NomeDaNovaSecao”, apresentada imediatamente abaixo. O conteúdo da nova seção criada é processado pela biblioteca que foi declarada em sua criação e a sua semântica vai depender dessa biblioteca.

Figura 28 - Estrutura básica do arquivo “app.config”

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="NomeDaNovaSecao" type="Nome da biblioteca responsável por processor a nova seção"/>
    </configSections>
    <NomeDaNovaSecao>
        <SecaoPersonalizada/>
    </NomeDaNovaSecao>
    <connectionStrings>
        <add name="NomeDaStringDeConexão" connectionString="String de conexão " />
    </connectionStrings>
</configuration>
```

Fonte: autoria própria (2015)

A seção “connectionStrings” é uma seção opcional que também é padronizada pelo ambiente. Seu propósito básico é a declaração de *strings* de conexção ODBC, que consistem em expressões textuais que informa ao sistema operacional qual *driver* ODBC será utilizado e quais os parâmetros de configuração utilizados por esse *driver*. O processamento dessas *strings*, no entanto, é feito pela biblioteca que a estiver utilizado, o que permite que sua sintaxe possa diferir daquilo que seria aceito e padronizado pelo ODBC. Conhecendo a estrutura básica do arquivo “app.config”, poremos analisar a configuração inicial dos três *frameworks*.

Para o *EntityFramework* (Figura 29), a configuração deve iniciar pela criação de uma

nova seção chamada “entityFramework”. Nesta seção deve ser declarado um elemento denominado “defaultConnectionFactory”, que especifica qual será a biblioteca responsável pelo processamento da fonte de dados que está sendo mapeada. Esses elementos são processados de forma automática pelo *Entity Framework*, por isso os nomes precisam ser específicos. Além dos elementos personalizados, o *Entity Framework* também requer a declaração de uma *string* de conexão, que será utilizada posteriormente. No nosso exemplo está sendo criada a *string* de conexão “BancoExemploEntities” através da seção padronizada, porém a sintaxe apresentada é específica do *Entity Framework*.

Figura 29 - Configuração básica do Entity Framework (gerada automaticamente)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <!--      For      more      information      on      Entity      Framework      configuration,      visit
http://go.microsoft.com/fwlink/?LinkID=237468 -->
        <section     name="entityFramework"     type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=4.4.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
    </configSections>
    <connectionStrings>
        <add                                                                                    name="BancoExemploEntities"
connectionString="metadata=res://*/Modelo.csdl|res://*/Modelo.ssdl|res://*/Modelo.msl;provider=System.Data.SqlClie
nt;provider    connection    string=&amp;quot;data    source=(local)\SQLEXPRESS;initial    catalog=BancoExemplo;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework&amp;quot;" providerName="System.Data.EntityClient" />
    </connectionStrings>
    <entityFramework>
        <defaultConnectionFactory   type="System.Data.Entity.Infrastructure.SqlConnectionFactory,   EntityFramework"
/>
    </entityFramework>
</configuration>
```

Fonte: autoria própria (2015)

A configuração do NHibernate é semelhante à do *Entity framework*, porém requer alguns detalhes adicionais, de acordo com a fonte de dados utilizada (Figura 30). Nos dois caos, *Entity Framework* e NHibernate,, o mapeamento das entidades ocorre em outro arquivo.

Figura 30 - Configuração básica do NHibernate.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="hibernate-configuration" type="NHibernate.Cfg.ConfigurationSectionHandler, NHibernate" />
    </configSections>
    <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
        <session-factory>
            <property name="connection.provider">NHibernate.Connection.DriverConnectionProvider</property>
            <property name="connection.dialect">NHibernate.Dialect.MsSql2005Dialect</property>
            <property name="connection.connection_string_name">BancoExemploEntities</property>
            <property                name="proxyfactory.factory_class">NHibernate.ByteCode.LinFu.ProxyFactoryFactory,
NHibernate.ByteCode.LinFu</property>
            <mapping assembly="assemblyname" />
        </session-factory>
    </hibernate-configuration>
    <connectionStrings>
        <add      name="BancoExemploEntities"      connectionString="Data      Source=(local)\SQLEXPRESS      Initial
Catalog=BancoExemplo; Integrated Security=True"/>
    </connectionStrings>
</configuration>
```

Fonte: autoria própria (2015)

O NHibernate também utiliza uma *string* de conexão para identificar a fonte de dados que será manipulada e também está restrito a uma fonte de dados por vez, porém essa *string* é mais simples que aquela utilizada pelo *EntityFramework* e pode ser configurada como uma

string de conexão padrão do .Net *Framework* ou como uma propriedade dentro do bloco de configuração “session-factory” (Figura 30).

No FrameDOD, todas as configurações são definidas no arquivo HOML, porém é necessário especificar o arquivo de configuração raiz, conforme apresentado na Figura 31. No exemplo é utilizado o arquivo “config.homl” localizado na mesma pasta do executável principal do programa, mas o nome e a localização desse arquivo não são mandatórios.

Figura 31 - Configuração básica do FrameDOD com arquivo HOML separado

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="HomlSettings" type="DataLibrary.HOML.HomlSettings, DataLibrary" />
    </configSections>
    <HomlSettings root="config.homl"/>
</configuration>
```

Fonte: autoria própria (2015)

Alternativamente, o usuário pode optar por não criar um arquivo HOML separado. Nesse caso, o arquivo de configuração deverá ser modificado conforme apresentado na Figura 32.

Figura 32 - Configuração do FrameDOD sem arquivo HOML separado

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="HomlSettings" type="DataLibrary.HOML.HomlSettings, DataLibrary" />
    </configSections>
    <HomlSettings>
        <!-- As configurações HOML devem ser introduzidas dentro desta seção -->
    </HomlSettings>
</configuration>
```

Fonte: autoria própria (2015)

As duas formas de configuração apresentadas são mutuamente excludentes, porém configurações incorretas podem levar a ambiguidades. Nesse caso, se um arquivo de configuração for especificado o conteúdo da seção de configuração será automaticamente desprezado e o conteúdo do arquivo indicado será utilizado.

Uma vez que as configurações básicas estejam prontas, os mapeamentos podem ser definidos. Para *Entity Framework*, esses mapeamentos são declarados no arquivo “EntityFramework.xml” (Figura 33), localizado na pasta de execução do aplicativo. O NHibernate utiliza uma estratégia diferente, pois para cada entidade mapeada será definido um arquivo XML separado com o mesmo nome da classe e a extensão “.hbm.xml”. No nosso exemplo (classe Paciente), esse arquivo será “Paciente.hbm.xml”. Em relação ao FrameDod, todos os mapeamentos deverão ser definidos no arquivo de configuração raíz, declarado nas configurações básicas. No nosso exemplo, esse arquivo seria o “config.homl” (Figura 31).

Conforme apresentado na Figura 33, o *Entity Framework* utiliza um arquivo de mapeamento completo, contendo todas as definições para todas as entidades mapeadas e a

maior parte das configurações relativas ao banco de dados, exceto pela *string* de conexão, definida no “App.config”.

Figura 33 - Mapeamento do banco de dados no Entity Framework (Gerado automaticamente)

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="2.0" xmlns:edmx="http://schemas.microsoft.com/ado/2008/10/edmx">
  <edmx:Runtime>
    <edmx:StorageModels>
      <Schema    Namespace="BancoExemploModel.Store"    Alias="Self"    Provider="System.Data.SqlClient"    ProviderManifestToken="2008"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
xmlns="http://schemas.microsoft.com/ado/2009/02/edm/ssdl">
        <EntityContainer Name="BancoExemploModelStoreContainer">
          <EntitySet Name="tabela_paciente" EntityType="BancoExemploModel.Store.tabela_paciente" store:Type="Tables" Schema="dbo" />
        </EntityContainer>
        <EntityType Name="tabela_paciente">
          <Key>
            <PropertyRef Name="cpf" />
          </Key>
          <Property Name="cpf" Type="numeric" Nullable="false" Precision="11" />
          <Property Name="nascimento" Type="date" Nullable="false" />
          <Property Name="endereco" Type="varchar" Nullable="false" MaxLength="300" />
          <Property Name="nome" Type="varchar" Nullable="false" MaxLength="100" />
          <Property Name="telefone" Type="numeric" Nullable="false" Precision="13" />
        </EntityType>
      </Schema>
    </edmx:StorageModels>
    <edmx:ConceptualModels>
      <Schema   Namespace="BancoExemploModel"   Alias="Self"   xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation"
xmlns="http://schemas.microsoft.com/ado/2008/09/edm">
        <EntityContainer Name="BancoExemploEntities" annotation:LazyLoadingEnabled="true">
          <EntitySet Name="tabela_paciente" EntityType="BancoExemploModel.tabela_paciente" />
        </EntityContainer>
        <EntityType Name="tabela_paciente">
          <Key>
            <PropertyRef Name="cpf" />
          </Key>
          <Property Name="cpf" Type="Decimal" Nullable="false" Precision="11" Scale="0" />
          <Property Name="nascimento" Type="DateTime" Nullable="false" Precision="0" />
          <Property Name="endereco" Type="String" Nullable="false" MaxLength="300" Unicode="false" FixedLength="false" />
          <Property Name="nome" Type="String" Nullable="false" MaxLength="100" Unicode="false" FixedLength="false" />
          <Property Name="telefone" Type="Decimal" Nullable="false" Precision="13" Scale="0" />
        </EntityType>
      </Schema>
    </edmx:ConceptualModels>
    <edmx:Mappings>
      <Mapping Space="C-S" xmlns="http://schemas.microsoft.com/ado/2008/09/mapping/cs">
        <EntityContainerMapping StorageEntityContainer="BancoExemploModelStoreContainer" CdmEntityContainer="BancoExemploEntities">
          <EntitySetMapping Name="tabela_paciente">
            <EntityTypeMapping TypeName="BancoExemploModel.tabela_paciente">
              <MappingFragment StoreEntitySet="tabela_paciente">
                <ScalarProperty Name="cpf" ColumnName="cpf" />
                <ScalarProperty Name="nascimento" ColumnName="nascimento" />
                <ScalarProperty Name="endereco" ColumnName="endereco" />
                <ScalarProperty Name="nome" ColumnName="nome" />
                <ScalarProperty Name="telefone" ColumnName="telefone" />
              </MappingFragment>
            </EntityTypeMapping>
          </EntitySetMapping>
        </EntityContainerMapping>
      </Mapping>
    </edmx:Mappings>
  </edmx:Runtime>
  <Designer xmlns="http://schemas.microsoft.com/ado/2008/10/edmx">
    <Connection>
      <DesignerInfoPropertySet>
        <DesignerProperty Name="MetadataArtifactProcessing" Value="EmbedInOutputAssembly" />
      </DesignerInfoPropertySet>
    </Connection>
    <Options>
      <DesignerInfoPropertySet>
        <DesignerProperty Name="ValidateOnBuild" Value="true" />
        <DesignerProperty Name="EnablePluralization" Value="False" />
        <DesignerProperty Name="IncludeForeignKeysInModel" Value="True" />
        <DesignerProperty Name="CodeGenerationStrategy" Value="None" />
      </DesignerInfoPropertySet>
    </Options>
    <Diagrams></Diagrams>
  </Designer>
</edmx:Edmx>
```

Fonte: autoria própria (2015)

A vantagem da abordagem utilizada pelo *Entity Framework* está no fato de que todos os mapeamentos podem ser encontrados em um só lugar, o que facilita a visualização global do sistema. Por outro lado, pode ser difícil localizar um mapeamento específico caso o arquivo de configuração esteja mal formatado, pois o arquivo utilizado é extremamente complexo. Essa desvantagem, no entanto, pode ser facilmente superada pelo uso da

ferramenta de visualização que acompanha o *framework*.

Figura 34 - Mapeamento do banco de dados no FrameDOD

```
<?xml version="1.0" encoding="utf-8" ?>
<homl  version="1.0">
 <header>
  <description>Exemplo de mapeamento</description>
 </header>
 <references>
  <drivers>
   <driver name="MsSQL" class="DataLibrary.SqlServer.Driver"/>
  </drivers>
 </references>
 <declarations>
  <datasources>
   <datasource name="MySQL1" driver=" MsSQL ">
    <attribute name=" integratedsecurity " value=" true "/>
    <attribute name=" server " value=" (local)\SQLEXPRESS "/>
    <attribute name="database" value=" BancoExemplo "/>
   </datasource>
  </datasources>
  <entities>
   <entity name="Paciente" class="Dissertacao. Paciente"/>
  </entities>
 </declarations>
 <mappings>
  <all>
   <Paciente>
    <attributes loadfrommodel="true">
     <attribute name="CPF" keytype="primary"/>
    </attributes>
    <source entity="tabela_pacientes"/>
    <operations template="crud"/>
   </Paciente>
  </all>
 </mappings>
</homl>
```

Fonte: autoria própria (2015)

O FrameDOD utiliza uma estratégia semelhante à do *Entity Framework*, porém os arquivos de configuração são mais sucintos, conforme pode ser observado na Figura 34, o que facilita a leitura. Apesar disso, uma formatação ruim ainda pode prejudicar a visualização.

O NHibernate utiliza uma estratégia de mapeamento diferente do *Entity Framework* e do FrameDOD. Em vez de utilizar um arquivo de mapeamento central, ele utiliza arquivos de configuração individuais para cada entidade a ser mapeada, conforme apresentado na Figura 35. Por um lado isso torna o arquivo de configuração mais fácil de ler, porém facilita o surgimento de redundâncias e dificulta a localização de um mapeamento específico, já que os mapeamentos ficam dispersos ao longo do projeto. Para contornar o problema, é necessária a doção de algumas regras de boas práticas específicas.

Obviamente, cada uma das estratégias apresentadas possui vantagens e desvantagens, porém a configuração do *Entity Framework* é visivelmente mais complexa que as configurações equivalentes no FrameDOD e no NHibernate, sendo viabilizada pelo uso da ferramenta integrada. Apesar de se tratar de uma tecnologia proprietária da Microsoft, ele também está disponível para outros ambientes através do projeto Mono.

Figura 35 - Mapeamento da classe Paciente no NHibernate

```
<?xml version="1.0" encoding="utf-8"?>
<hibernate-mapping namespace="Exemplo" xmlns="urn:nhibernate-mapping-2.2"
assembly="Exemplo">
  <class name="Paciente" table="tabela_paciente" schema="dbo">
    <id name="Cpf" type="Long">
      <column name="cpf" not-null="true" precision="11" scale="0" sql-type="numeric" />
      <generator class="assigned" />
    </id>
    <property name="Nascimento" type="DateTime">
      <column name="nascimento" not-null="true" sql-type="date" />
    </property>
    <property name="Endereco" type="String">
      <column name="endereco" not-null="true" length="300" sql-type="varchar" />
    </property>
    <property name="Nome" type="String">
      <column name="nome" not-null="true" length="100" sql-type="varchar" />
    </property>
    <property name="Telefone" type="Long">
      <column name="telefone" not-null="true" precision="13" scale="0" sql-type="numeric" />
    </property>
  </class>
</hibernate-mapping>
```

Fonte: autoria própria (2015)

O NHibernate e o FrameDOD utilizam estratégias de mapeamento mais objetivas, requerendo muito menos código para que se estabeleça o mapeamento, apesar de divergirem em relação ao uso de arquivos únicos ou individualizados por classe. Para o exemplo apresentado, o FrameDOD apresentou quantidade de código ligeiramente menor que o NHibernate contando a configuração básica e o mapeamento, praticamente beirando a equivalência. O *Entity Framework*, por outro lado, gerou uma quantidade enorme de código, superando o FrameDOD e o NHibernate combinados.

4.2 EXECUÇÃO DAS OPERAÇÕES SOBRE OS DADOS

Uma vez que as fontes de dados estejam mapeadas, a manipulação dos dados pode ser executada. Essas operações são apresentadas abaixo na seguinte ordem:

a) Criação de um registro novo (Create);
b) Leitura de um registro existente (Read);
c) Alteração nos dados de um registro existente (Update);
d) Exclusão de um registro existente (Delete).

Para todos os *frameworks* apresentados, foi utilizada a mesma entidade e o mesmo processo de preenchimento dos dados dessa entidade. Por esse motivo, somente as operações propriamente ditas foram transcritas.

No *Entity Framework* (Figura 36 A) a execução de qualquer operação requer o uso de

uma entidade específica derivada da classe “DbContext”. Essa entidade é criada de forma automatizada, porém algumas das suas funcionalidades precisam ser implementadas manualmente. Inicialmente, uma instância dessa classe é criada e, a partir dela, a operação Inserte (método “Add()”) pode ser executada, porém os dados são gravados no banco de dados somente após a execução do método “SaveChanges()”. No FrameDOD (Figura 36 B), as operações CRUD podem ser executadas a partir de métodos estáticos definidos na classe genérica “HomlDAO”. A especificação do tipo de entidade e da fonte de dados são feitas nos argumentos do protótipo e a operação “Insert” tem efeito imediato após o método “Insert()” ser chamado.

Figura 36 - Operação Insert

A	`BancoExemploEntities bancoExemploEntities = new BancoExemploEntities();`
	`bancoExemploEntities.tabela_paciente.Add(paciente);`
	`bancoExemploEntities.SaveChanges();`

B	`HomlDao<Paciente, Datasourcexemplo>.Insert(paciente);`

C	`ISession session = NHibertnateSession.OpenSession();`
	`ITransaction transaction = session.BeginTransaction();`
	
	`session.Save(paciente);`
	`transaction.Commit();`

Fonte: autoria própria (2015)

O NHibernate (Figura 36 C) utiliza uma estrutura parecida com a do *Entity Framework*, porém as classes básicas que estabelecem comunicação com o banco de dados são nativas e não precisam ser criadas. No exemplo, uma sessão precisa ser aberta (método “OpenSession()”) antes de qualquer manipulação de dados. Dentro da sessão é iniciada uma transação, que permite a execução de operações no modelo “Tudo ou nada”, de modo que qualquer falha causará a reversão de todas as modificações. A operação de inserção é executada pela chamada do método “Save()”, porém os dados só são persistidos depois da execução do método “Commit()”.

No caso da operação “Select” (Figura 37), existe pouca diferença em relação à operação “Insert” (Figura 36) e o resultado da requisição é imediato em todos os *frameworks*. Para o *Entity Framework* (Figura 37 A), a operação “Select” é iniciada pela criação de uma instância da classe que representa o banco de dados.

Figura 37 - Operação Select

A	`BancoExemploEntities bancoExemploEntities = new BancoExemploEntities();` `IEnumerable<Paciente> pacientes = bancoExemploEntities.tabela_paciente.Select(paciente => paciente);`
B	`IList<Paciente> pacientes = HomlDao<Paciente, Datasourcexemplo>.Select();`
C	`ISession session = NHibertnateSession.OpenSession();` `IList<Paciente> pacientes = session.Query<Paciente>().ToList();`

Fonte: autoria própria (2015)

O segundo e último passo é a chamada do método “Select()”, cujo efeito é imediato, retornando uma enumeração contendo os itens obtidos do banco. No FrameDOD (Figura 37 B), a operação “Select” ocorre do mesmo modo que a operação “Insert”, porém nesse caso é devolvida uma lista de objetos do tipo especificado. Para o NHibernate (Figura 37 C), a principal mudança em relação à operação Insert é a inexistência da transação, que se torna desnecessária para operações que não causam modificações no banco de dados.

Figura 38 - Operação Update

A	`BancoExemploEntities bancoExemploEntities = new BancoExemploEntities();` `bancoExemploEntities.SaveChanges();`
B	`HomlDao<Paciente, Datasourcexemplo>.Update(paciente);`

C	`ISession session = NHibertnateSession.OpenSession();`
	`ITransaction transaction = session.BeginTransaction();`
	`session.Save(paciente);`
	`transaction.Commit();`

Fonte: autoria própria (2015)

A operação Update no *Entity Framework* (Figura 38 A) possui algumas características ocultas, pois o *Framework* mantem uma referência interna para cada objeto que ele instancia. Desse modo, não existe um método “Update()” para ser chamado. As modificações são identificadas automaticamente e são persistidas após a execução do método “SaveChanges()”, afetando todos os objetos que sofreram modificações. No FrameDOD (Figura 38 B), a operação “Update” não difere muito da operação “Delete”. As modificações são identificadas no ato da execução do método e afetam somente o objeto que foi passado como referência. No caso do NHibernate (Figura 38 C), o método utilizado para atualizar o registro existente é o mesmo utilizado para criar um item novo. Para diferenciar as operações, o NHibernate utiliza um cache interno, semelhante ao *Entity Framework*. Se o objeto a ser gravado foi instanciado pelo NHibernate, ele terá uma referência guardada e assumira que a operação a ser executada será “Update”.

Figura 39 - Operação Delete

A	`BancoExemploEntities bancoExemploEntities = new BancoExemploEntities();`
	`bancoExemploEntities.tabela_paciente.Remove(paciente);`
	`bancoExemploEntities.SaveChanges();`

B	`HomlDao<Paciente, Datasourcexemplo>.Delete(paciente);`

C	`ISession session = NHibertnateSession.OpenSession();`
	`ITransaction transaction = session.BeginTransaction();`
	`session.Delete(paciente);`
	`transaction.Commit();`

Fonte: autoria própria (2015)

A operação “Delete” (Figura 39) não apresenta nenhuma diferença significativa em relação à operação “Insert” em nenhum dos *frameworks*, exceto pela chamada do método

“Remove” para o *Entity Framework*. Tanto no *Entity Framework* quanto no NHibernate, o objeto a ser excluído precisa ter sido previamente instanciado pelo Frameworrk. Se uma nova entidade for passada como referência, isso causará uma exceção, pois o *framework* não sabe o que fazer com os dados. No caso do FrameDod, não existe diferença significativa.

Conforme visto, o suporte às operações CRUD é nativo nos três *Frameworks*, porém cada um deles adota uma abordagem diferente, apesar de apresentarem eventuais semelhanças. Através dos exemplos apresentados foi possível observar que o *Entity Framework* possui a estratégia de configuração mais complexa, porém o fato de possuir uma ferramenta integrada para a sua configuração compensa em boa parte a dificuldade para sua configuração. O NHibernate e o FrameDOD mostraram-se muito próximos em relação à facilidade de configuração, porém cada um deles utiliza uma estratégia de configuração diferente, o que pode determinar a preferência de alguns usuários por um ou pelo outro. Por causa dessas características, foi possível determinar que apenas o FrameDOD e o NHibernate foram satisfatórios em relação à simplicidade.

Todos os *frameworks* foram satisfatórios em relação à eficácia, cumprindo todas as etapas desde o mapeamento até execução das operações sobre os dados. Apesar do *Entity Framework* requerer a criação de classes especializadas e ser extremamente complexo para ser configurado, sua ferramenta de configuração supre satisfatoriamente essas necessidades.

Quanto à representatividade, o *Entity Framework* foi o único reprovado, pois a dependência da ferramenta de configuração é muito grande e alguns aspectos da sua utilização são obscuros, requerendo uma leitura exaustiva da documentação até que se adquira conhecimento suficiente para utilizá-lo.

4.3 OUTRAS CARACTERÍSTICAS

Além das configurações básicas e da execução das operações CRUD, existem outras características importantes que precisam ser analisadas na comparação dos três *frameworks*, tais como a execução de operações com entidades complexas, a interoperabilidade e a utilização com outros tipos de fontes de dados.

Os três *frameworks* possuem capacidade para o mapeamento de entidades complexas, diferindo um do outro, somente pela forma como esse mapeamento é feito. No NHibernate e no FrameDOD, isso pode ser feito tanto através dos arquivos de configuração quanto diretamente no código, enquanto o *Entity Framework* utiliza apenas os arquivos de

configuração. Isso não é por si só uma vantagem ou uma desvantagem, mas pode direcionar a escolha de alguns usuários. Nos três *Frameworks*, essas configurações são feitas com base no modelo relacional, diferindo mais quanto à sintaxe, o que não é relevante nessa análise.

Um ponto importante que precisa ser considerado no mapeamento de entidades complexas é a capacidade de cada permitir que o usuário decida quando carregar toda a estrutura da entidade ou somente uma parte dela, pois nem sempre o carregamento completo é desejável. No caso do NHibernate e do *Entity Framework*, isso é feito pela utilização de uma estratégia denominada *Lazy Load* (carga preguiçosa). Isso permite que a entidade seja carregada parcialmente quando o usuário não precisa da estrutura completa, porém o *framework* decide que parte será carregada. O FrameDOD foi criado tendo como objetivo apresentar uma alternativa viável para esse problema, adoytando uma estratégia completamente diferente. O *Lazy Load* não foi adotado, pois o FrameDOD permite que o usuário defina vários mapeamentos diferentes para as mesmas entidades. Desse modo, o usuário pode decidir detalhadamente que parte da entidade ele deseja carregar em cada momento, dependendo do contexto de utilização.

A seleção de uma forma de mapeamento em particular no FrameDOD ocorre através do par Entidade/Fonte de dados. Isso é possível, pois a mesma fonte de dados pode ser mapeada quantas vezes o usuário desejar, e cada mapeamento recebe um nome diferente. Desse modo que a seleção de um mapeamento ou de outro ocorre através da seleção desse nome, que é definido pelo usuário.

Outro aspecto importante a ser considerado é a capacidade de interoperabilidade, porém não é possível uma comparação direta entre o FrameDOD e os demais *Frameworks*, pois somente o FramedDOD possui essa funcionalidade. Nos demais *frameworks* a solução precisa ser feita através de artifícios de programação, pois eles não possuem suporte nativo nem à interoperabilidade nem à utilização de múltiplos mapeamentos.

O *Entity Framework* permite a utilização de mais de uma fonte de dados, desde que seja com mapeamentos completamente diferentes, porém sua ferramenta de configuração não permite a reutilização das entidades mapeadas e não permite o mapeamento de entidades com estruturas que sejam ligeiramente diferentes da sua representação no banco de dados. O NHibernate não possui nenhum tipo de suporte a esse tipo de funcionalidade. Foge ao escopo dessa dissertação o detalhamento de como tratar esses aspectos no *Entity Framework* e no NHibernate, pois isso levaria a uma discussão extensiva e desviaria o foco do trabalho. No FrameDOD, no entanto, o suporte à interoperabilidade é nativo, assim como o suporte a múltiplas fontes de dados, conforme já demonstrado nos capítulos anteriores.

A execução das operações com entidades complexas não apresenta nenhum tipo de característica especial no *Entity Framework* e no FrameDOD, não diferindo em nada das operações executadas com entidades simples. No NHibernate, entretanto, existe uma particularidade que precisa ser mencionada. Quando partes distintas de uma entidade complexa são mapeadas a partir de tabelas diferentes, é necessária a execução de uma operação de Junção entre as tabelas, o que é um procedimento comum a todos os *frameworks*. A diferença é que no NHibernate a operação *Inner Join* força o carregamento da entidade completa, enquanto o *Left Join* implica na utilização do *Lazy Load*. Isso pode confundir os usuários que não estejam familiarizados com a ferramente, pois mesmo as entidades que estiverem completas (com todos os dados presentes nas duas tabelas) serão carregadas de forma parcial.

Outra característica presente somente no FrameDOD é o suporte nativo à utilização de fontes de dados não convencionais. Apesar de todos os *frameworks* terem a capacidade de extensibilidade, o FrameDOD foi projetado para atender a essa necessidade específica. Isso significa que a comunicação entre as camadas do FrameDOD fornecem elementos estruturados de alto nível que permitem ao criador driver decidir como implementar as operações. No NHibernate e no *Entity Framework*, a premissa de construção é baseada na utilização de bancos de dados relacionais, de modo que a utilização de outros tipos de fontes de dados é uma tarefa muito mais complexa. Isso requer que cada driver implemente um processador de linguagem muito mais poderoso para tratar todos os aspectos das operações e da linguagem de consulta utilizada.

5 CONCLUSÕES

Partindo dos primeiros experimentos até a versão atual, o FrameDOD passou por muitas transformações que foram necessárias para tratar os problemas encontrados ao longo do seu desenvolvimento. As primeiras versões apresentavam algumas deficiências de usabilidade que criavam certa resistência ao seu uso e foram gradualmente sanadas a cada novo experimento. Os experimentos tiveram papel extremamente importante no aprimoramento da ferramenta, uma vez que trouxeram à tona diversos problemas e situações que não haviam sido experimentadas antes e que não foram identificadas nos testes de mesa.

O terceiro experimento foi relativamente importante, pois teve como propósito avaliar a facilidade de adaptação de um programador com pouca experiência e sem qualquer contato prévio com o FrameDOD. O resultado apresentado foi satisfatório, uma vez que poucas dúvidas foram apresentadas e poucos minutos foram necessários para que a familiarização ocorresse. O FrameDOD ainda continua em uso nas duas últimas empresas onde os experimentos foram executados, mas não foram coletadas informações atualizadas relativas à primeira empresa testada, uma vez que não foi mantido contato após o fim do experimento.

Dentre os resultados obtidos ao longo do trabalho, foi desenvolvida uma versão plenamente operacional do FrameDOD para utilização com o .Net *framework*, escrito integralmente em C#. Também foi criada uma versão preliminar da linguagem HOML, que foi a base para a publicação de um artigo no 12ª CONTECSI, ocorrida no período de 20 a 22 de maio de 2015.

Ainda existem melhorias que podem ser implementadas para aprimoramento das ferramentas criadas, porém todos os objetivos propostos foram alcançados satisfatoriamente, conforme descrito abaixo:

Definição de uma linguagem de consulta independente de fonte de dados – Foi criada a linguagem HOML que permite a especificação de expressões de consulta através de uma estrutura completamente independente de qualquer fonte de dados;

Independência de localização – Conforme a proposta original, o FrameDOD permite ao programador interagir com as fontes de dados sem que qualquer informação sobre a sua localização precise ser colocada no código do programa. Em vez disso, a única informação utilizada é o nome da fonte de dados, conforme declarado no arquivo HOML;

Independência de ambiente operacional – Por se tratar de um *framework*, o

FrameDOD é compatível com o mesmo ambiente de execução da linguagem onde ele for utilizado. Desse modo, o ambiente de execução torna-se irrelevante, pois os *drivers* responsáveis pelo funcionamento garantirão o isolamento e serão executados no mesmo ambiente do programa principal de maneira transparente;

Independência de fonte de dados – Toda a comunicação com as fontes de dados ocorre através do *drivers* por meio de uma interface padronizada. Isso significa que o programa principal não possui qualquer interação direta com essas fontes nem possui qualquer informação sobre elas, garantindo a independência;

Independência sintático-semântica – Todas as operações do FrameDOD são definidas sem a utilização de qualquer recurso da fonte de dados hospedeira. Em vez disso, todas as operações são especificadas na forma de árvores de sintaxe que são processadas pelos *drivers*, produzindo os resultados desejados e garantindo a independência sintático-semântica;

Extensibilidade – Um dos pontos focais da criação da HOML foi a garantia da capacidade de expressão do programador. Isso é possível, pois cada *driver* possui um processador de linguagem integrado que permite a criação de extensões da linguagem, agregando novas funcionalidades sem a necessidade de recompilação do *framework*.

Alguns trabalhos futuros ainda serão concluídos ou executados até o final de 2015, a citar:

a) Implementação de uma ferramenta para automatização da criação de mapeamentos em HOML;
b) Disponibilização do *framework* para o publico de modo que seja possível obter retorno dos usuários para aprimoramento da ferramenta;
c) Publicação de artigos em Journal referentes à HOML e ao FrameDOD;
d) Disponibilização do *framework* para a comunidade a fim de obter massa crítica de testes para validação do *framework*.

Futuramente será criada uma versão do *framework* em JAVA, que também será disponibilizada para o público. Esse projeto, no entanto, ainda não possui data prevista.

REFERÊNCIAS

AHMED K. Z. ; UMRYSH C. E. **Developing Enterprise Java Applications with J2EE and UML.** Washington: Addson-Wesley, 2001.368 p. ISBN 9780201738292.

BARBOSA, Á. C. P. et al. **Integração de Dados Heterogêneos em Ambiente Web.** Rio das Ostras: IV Escola Regional de Informática, 2004. 18p. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erirjes/2004/009.pdf>. Acesso em: 14 jan. 2012.

BAUER, C. ; KING G. **Hibernate in Action.** Greenwich: Manning, 2005.

BAILEY, I.; PARTRIDGE, C. **An Analysis of Services.** United Kingdom: Model Futures, 2010. 38p. Disponível em: <http://www.modelfutures.com/file_download/17/MOD+CIO+-+Service+Analysis+Report+-+v1.3.pdf>. Acesso em: 12 dez. 2012.

BHATTI, S.; ABRO, Z.; RUFABRO, F. Performance evaluation of java based object relational mapping tool. **Mehran University Research Journal of Engineering and Technology**, v. 32, n. 2, p. 159-166, 2013. Disponível em: <http://publications.muet.edu.pk/research_papers/pdf/pdf755.pdf>. Acesso em: mar 2014.

BEGIN, C.. **iBATIS Data Mapper:** developer guide. [S.l.]: Apache Foundation, 2006. Disponível em: <https://ibatis.apache.org/docs/java/pdf/iBATIS-SqlMaps-2_en.pdf>. Acesso em: nov. 2011.

BOUGANIM, L. et al. **The Ecobase project: database and web technologies for environmental information systems**. ACM Sigmod Record, v. 30, n. 3, p. 70-75, 2001. Disponível em: <http://www-smis.inria.fr/dataFiles/B01.pdf>. Acesso em: nov 2012.

BUSICHIA, G.; FERREIRA, J. E. **Compartilhamento de módulos de bases de dados heterogêneas através de objetos integradores.** In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 14., 1999, Florianópolis. Anais... 1999. 15p. Disponível em: <http://www.inf.ufsc.br/~sbbd99/anais/SBBD-Completo/33.pdf#http://www.inf.ufsc.br/~sbbd99/anais/SBBD-Completo/33.pdf>. Acesso em: 2 jun. 2011.

CÂMARA, G. et al. TerraLib: **An open source GIS library for large-scale environmental and socio-economic applications.** In: OPEN source approaches in spatial data handling. Berli:n Springer Berlin Heidelberg, 2008. p. 247-270. Disponível em: <http://www.inf.puc-rio.br/~casanova/Publications/Papers/2007-Papers/2007-camara.pdf>. Acesso em: nov. 2011.

CHAUDHURI, S. ; NARASAYYA, V.R.; SYAMALA, M. **Database application developer tools using static analysis and dynamic profiling.** IEEE Data Eng. Bull., v. 37, n. 1, p. 38-47, 2014. Disponível em: <ftp://131.107.65.22/pub/debull/A14mar/p38.pdf >. Acesso em: dez 2014.

CHENGKAI, L. et al. **RankSQL: supporting ranking queries in relational database management systems.** In: VLDB Conference, 2005, Trondheim. **Proceedings...** 2005. 4p. Disponível em: <http://www.vldb.org/conf/2005/papers/p1342-li.pdf>. Acesso em: 14 nov. 2011.

CODD, E. F. **The Relational model for database management:** version 2. Washington: Addison-Wesley, 1990.

CONSTANTIN, C.; DU MOUZA, C.; TRAVERS, N. **Browse your content-based distributed repository**!. 2012. Disponível em <http://cedric.cnam.fr/fichiers/art_2615.pdf>. Acesso em: jan. 2013.

COPELAND, R. **Essential sqlalchemy**. [S.l.]: O'Reilly Media, Inc., 2008.

DE SOUZA, V. C. O.; DOS SANTOS, M. V. C. **Maturing, Consolidation and performance of NoSQL databases-comparative study**. In: ANNUAL CONFERENCE ON BRAZILIAN SYMPOSIUM ON INFORMATION SYSTEMS: INFORMATION SYSTEMS: A COMPUTER SOCIO-TECHNICAL PERSPECTIVE, 1., 2015. **Proceedings…** 2015. p. 32. Disponível em: <http://plutao.sid.inpe.br/col/sid.inpe.br/plutao/2015/06.01.12.27/doc/souza_amadurecimento.pdf>. Acesso em: 10 ago.2015.

FERREIRA, K. R. et al. **Arquitetura de software para construção de bancos de dados geográficos com SGBD objeto-relacionais**. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 17., 2002, Gramado. **Anais...** 2002. 11p. Disponível em: <http://www.lbd.dcc.ufmg.br:8080/colecoes/sbbd/2002/004.pdf>. Acesso em: 13 jan 2012.

GABRICK, K. A.; WEISS, D. B. **Java 2EE and XML Development.** 1st Ed. Greenwich: Manning Publications Co., 2002. 320p. ISBN 1-930110-30-8.

GAMMA, E. et al. **Padrões de projeto**: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, c2000. Reimpressão 2008.

GEHANI, N. H.; JAGADISH, H. V.; ROOME, W. D. OdeFS: a file system interface to an object-oriented database. In: VLDB, 1994. **Proceedings...** 1994. p. 249-260. Disponível em: <http://www.vldb.org/conf/1994/P249.PDF>. Acesso em: 14 nov. 2011.

GRIFFIN, C. **A comparative look at entity framework code first.** 2013. Disponível em <http://digitalcommons.lasalle.edu/cgi/viewcontent.cgi?article=1010&context=mathcompcapstones>. Acesso em ago 2015.

HAMIDEH, A.; KNAPP, D. An extensible object-oriented approach to databases for VLSI/CAD. In: In: VLDB. 2000. **Proceedings...** 2000. p. 623-626.

HEUSER, C. A. **Projeto de banco de dados.** 4. ed. São Paulo: Sagra Luzzatto, 1998. 206 p. (Livros didáticos Universidade Federal do Rio Grande do Sul. Instituto de Informática; 4).

HIROSHI, I.; KAZUMI K. **An active object-oriented database: a multi-paradigm approach to constraint management.** In: VLDB, 19., 1993. Proceedings... 1993. 12p. Disponível em: <http://www.vldb.org/conf/1993/P467.PDF>. Acesso em: 14 nov 2011.

HUSTED, T. Dumoulin, C.; FRANCISCUS, G. Winterfeldt, D. **Struts in action:** building web applications with the leading java framework. Greenwich: Manning, 2003. 664p.

KIMMEL, P. T. **Professional devexpress ASP. NET Controls**. [S.l.]: John Wiley & Sons, 2010.

KIM, W. Object-oriented database systems: promises, reality, and future. In: VLDB, 19., 1993. **Proceedings...** 1993. p. 676-692. Disponível em: <http://www.vldb.org/conf/1993/P676.PDF>. Acesso em: 14 nov. 2011.

KÜHN, Eva. **The zero-delay data warehouse: mobilizing heterogeneous databases.**

In: VLDB, 29., 2003, Berlin. Proceedings... 2003. 6p. Disponível em: <http://www.vldb.org/conf/2003/papers/S33P02.pdf>. Acesso em: 14 nov. 2011.

KUROSE, J. F. **Redes de computadores e a Internet:** uma nova abordagem top-down. 3. ed. São Paulo: Pearson, 2006.

LAW, Yan-Nei; WANG, H.; ZANIOLO, C. **Query languages and data models for database sequences and data streams.** In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 30., 2004. Proceedings... 2004**.** p. 492-503

LINNEMANN, V. et al. **Design and implementation of an extensible database management system supporting user defined data types and functions**. Los Angeles: In: VLDB CONFERENCE, 14., 1988. Proceedings... 1988. 12p. Disponível em: <http://www.vldb.org/conf/1988/P294.PDF>. Acesso em: 14 nov. 2011.

LONEY, K. **Oracle database 11g the complete reference**. [S. l.]: McGraw-Hill, Inc., 2008.

LUO, Q.; NAUGHTON, J. F. **Form-based proxy caching for database-backed web sites**. in: VLDB CONFERENCE, 27., 2001, Roma. Proceedings... 2001. 10p. Disponível em: <http://www.vldb.org/conf/2001/P191.pdf>. Acesso em: 14 nov 2011.

MANOLESCU, I.et al. **Agora: Living with XML and relational.** In: In: VLDB CONFERENCE, 2000. Proceedings... 2000. p. 623-626. Disponível em: <ftp://ftp.cse.buffalo.edu/users/azhang/disc/disc01/cd1/out/papers/vldb/agoralivingwithiodad.pdf>. Acesso em: ago. 2012.

MARGUERIE, F.; EICHERT, S.; WOOLEY, J. **LINQ in Action**. [S.l.]: Manning, 2008. ISBN 1-933988-16-9.

MICROSOFT. **ODBC basics**. United States: Microsoft Corporation, 2010. Disponível em <https://msdn.microsoft.com/pt-br/library/thzzea08%28v=vs.100%29.aspx>. Acesso em: nov. 2011.

MINHAS, U. F. ; RAJAGOPALAN, S.; CULLY, B. **RemusDB: Transparent high availability for database systems.** The VLDB Journal, v. 22, n. 1, p. 29-45, 2011. 11p. Disponível em: <http://www.vldb.org/pvldb/vol4/p738-minhas.pdf>. Acesso em: 14 nov. 2011.

MOLZ, K. W. **Um framework para construção de aplicações OO sobre SGBD relacional.** Porto Alegre: Universidade Federal do Rio Grande do Sul. Instituto de Informática. Curso de Pós-Graduação em Ciência da Computação, 1999. 83p. Disponível em: <#http://hdl.handle.net/10183/25974#>. Acesso em: 17 jan. 2012.

PAN, A. et al. **The denodo data integration platform.** In: VLDB Conference, 28., 2002, Seoul. Proceedings... 2002. p. 986-989. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.19.8146>. Acesso em: nov. 2011.

REESE, G. **Database porgramming with Jdbc and Java.** 2. ed. Florida: O'Reily, 1997.

RICHLY, S.; HABICH, D.; LEHNER, W. GignoMDA: exploiting cross-layer optimization for complex database applications. In: VLDB CONFERENCE, 19., 2006, Seoul. **Proceedings...** 2006. 4p. Disponível em: <http://www.vldb.org/conf/2006/p1251-habich.pdf>. Acesso em: 14 nov. 2011.

SAYÃO, L. F.; MARCONDES, C. H. **O desafio da interoperabilidade e as novas perspectivas para as bibliotecas digitais**. Transinformação**,** v. 20, n. 2, 2012. Disponível em: <http://periodicos.puc-campinas.edu.br/seer/index.php/transinfo/article/download/530/510>. Acesso em: 10 ago.2015.

SHETH, A.P.; LARSON, J. A. **Federated database systems for managing distributed, heterogeneous, and autonomous databases**. ACM Computing Surveys (CSUR), v. 22, n. 3, p. 183-236, 1990.

SHAHRAM, G.et al. **Implementing a language for specifying active database execution models**. In: VLDB Conference, 19., 1993, Dublin. Proceedings... 1993. 5p. Disponível em: <http://www.vldb.org/conf/1993/P441.PDF>. Acesso em: 16 nov. 2011.

SHIVAKUMAR, V. ZHANG, T. **Heterogeneous database query optimization in DB2 universal datajoiner** . In: VLDB Conference, 24., 1998, New York. Proceedings... 1998. 5p. Disponível em: <http://www.vldb.org/conf/1998/p685.pdf>. Acesso em: 14 nov. 2011.

SIIDKARNP, N.; I,INNCMANN, V. **Elimination of views and redundant variables in an sql-like database language for extended NF² structures.** VLDB CONFERENCE, 16., 1990, Brisbane. Proceedings... 1990. 12p. Disponível em: <http://www.vldb.org/conf/1990/P302.PDF>. Acesso em: 16 nov 2011.

SILBERSCHATZ, A; GALVIN, PB; GAGNE, G. **Fundamentos de sistemas operacionais**. Rio de Janeiro (RJ): LTC, 2010.

SILVA, F. A. da; PERRI, Carlos Renato de Souza; ALMEIDA, Leandro Luiz de. **Desenvolvimento de uma ferramenta assistente para criação de aplicações CRUD em Java na Web.** In: COLLOQUIUM EXACTARUM. 2011. Anais... p. 70-82. Disponível em: <http://revistas.unoeste.br/revistas/ojs/index.php/ce/article/download/460/906>. Acesso em: out. 2015.

SOMMERVILLE, I. **Engenharia de software**. São Paulo: Pearson Education do Brasil, 2011.

STEVENSON, J. E. **Extraction tools and relational database schemas for CVS, SVN, and bazaar revision control systems.** 2008. Tese (Doutorado)- Brigham Young University, 2008. Disponível em: <http://repository2mysql.googlecode.com/svn-history/r97/trunk/doc/LaTeX/proposal.pdf>. Acesso em: nov. 2011.

SÜDKAMP, N.; LINNEMANN, V. **Elimination of view and redundant variables in a sql-like database language for Extended NF2 Structures**. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 16., 1990. Proceedings... 1990. p. 302-313.

SUSAN M. **DataBase management systems and the internet**. In: VLDB, 22., 1996, Mumbai (Bombay) Proceedings... 1996. 2p. Disponível em: <http://www.vldb.org/conf/1996/P600.PDF>. Acesso em: 14 nov. 2011.

TANENBAUM, A.S.; WOODHULL, A. S. **Sistemas operacionais:** projeto e implementação. Porto Alegre, RS: Bookman, 2008.

TORRES, A.; GALANTE, R. **Essential notation for object-relational mapping**. 2014. Tese (Doutorado)-Universidade Federal do Rio Grande do Sul. 2014. Disponível em:

<http://www.lume.ufrgs.br/bitstream/handle/10183/97116/000920633.pdf?sequence=1>. Acesso em: 10 ago. 2015.

VENKATARAMAN, S.; ZHANG, T. **Heterogeneous database query optimization in DB2 Universal DataJoiner.** In: VLDB Conference, 24., 1998, New York. Proceedings... 1998. 5p. Disponível em: <#http://www.vldb.org/conf/1998/p685.pdf#>. Acesso em: 14 nov. 2011.

VIDAL, V.et al. **Geração automática de visões de objeto de dados relacionais**. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 20., 2005, Uberlândia. Anais... 2005 15p. Disponível em: <http://www.lbd.dcc.ufmg.br:8080/colecoes/sbbd/2005/011.pdf>. Acesso em: 14 jan. 2012.

APÊNDICE A – FORMAS DE CLASSIFICAÇÃO DAS FONTES DE DADOS

Quadro 3: Classificação das principais fontes de dados quanto à categoria

Classificação	Fontes de dados
Arquivos Estruturados	• CSV • HTML • XML • XSL
Arquivos simples	• Arquivos binários • Arquivos de acesso aleatório • Arquivos sequenciais
Bancos de dados	• Access • DBase • Paradox
Comunicação entre processos	• Broadcast • Memória compartilhada • Multicast • NFC • Unicast
Dispositivos de armazenamento	• Disco rígido • Flash Drives • Unidades SSD
Dispositivos de entrada e saída	• Centrais telefônicas • GPS • Sensores • Servomecanismos
SGBDs	• MySQL • Oracle • PostgreSQL • SQL Server
Serviços	• E-Mail • Sms • WebServices
Frameworks	• Hibernate • Active Records • LINQ
Outros	• Sistema de arquivos

Fonte: autoria própria (2015)

Quadro 4: Classificação das fontes de dados quanto à persistência

Classificação	Categorias
Persistentes	• Arquivos estruturados • Arquivos simples • Bancos de dados • Dispositivos de armazenamento • SGBDs • Sistema de arquivos
Não persistentes	• Comunicação entre processos • Dispositivos de entrada e saída
Independentes de persistência	• Serviços • *Frameworks*

Fonte: autoria própria (2015)

- São consideradas **persistentes** as fontes de dados cujo propósito seja o armazenamento dos dados e a sua posterior recuperação.
- São consideradas **não persistentes** as fontes de dados cujo propósito é a troca, transferência ou conversão de dados, não importando se esses dados serão ou não persistidos.
- São consideradas **independentes de persistência** as fontes de dados cujo proposito é prover a abstração das operações sobre os dados, havendo ou não o suporte para o controle da persistência na camada inferior.

APÊNDICE B – VANTAGENS E DESVANTAGENS DAS FONTES DE DADOS

As vantagens e desvantagens apresentadas neste apêndice levam em consideração os métodos de acesso e manipulação mais frequentemente utilizados pelos programadores. Nos casos específicos das fontes de dados proprietárias, são considerados os mecanismos fornecidos pelos próprios fabricantes. Em ambos os casos foram desprezados os mecanismos ou recursos fornecidos por terceiros, exceto aqueles providos como parte do sistema operacional.

As classificações apresentadas possuem cunho qualitativo, sendo utilizadas as seguintes convenções:

Quanto à abstração:

- **Inexistente**: indica que a fonte de dados não possui estrutura de registros;
- **Baixa**: indica que a fonte de dados possui estrutura de registros, mas a estrutura de campos é inexistente ou depende do programador;
- **Alta**: indica que a fonte de dados possui estrutura de registros com campos bem definidos.

Quanto à legibilidade (leitura por *software*):

- **Ruim**: indica que a leitura do conteúdo através de *software* é complexa, requerendo tratamento muito específico e muito esforço de programação;
- **Boa**: indica que a leitura do conteúdo através de *software* requer pouco esforço de programação, porém a estrutura não pode ser reconhecida automaticamente, requerendo tratamento específico;
- **Excelente**: indica que a leitura do conteúdo através de *software* requer pouco esforço e a estrutura pode ser reconhecida automaticamente.

Quanto à facilidade de manipulação:

- **Simples**: indica que a fonte de dados requer pouco esforço de programação para leitura e escrita;
- **Complexa**: indica que a fonte de dados requer muito esforço de programação para leitura e escrita.

Quanto à ordem:

- **Inexistente**: indica que a fonte de dados não possui suporte à ordenação dos dados;
- **Manual**: indica que a fonte de dados possibilita a ordenação dos dados, mas requer o uso de recursos de programação para prover essa funcionalidade;
- **Personalizável**: indica que a fonte de dados possui suporte nativo à ordenação dos dados, conforme parâmetros definidos pelo programador.

Quanto à padronização:

- **Inexistente**: indica que a fonte de dados não possui padronização definida;
- **Ruim**: indica que a fonte de dados possui padronização limitada;
- **Boa**: indica que a fonte de dados possui uma padronização completa.

Quanto à indexação:

- **Inexistente**: indica que a fonte de dados não possui nenhum suporte à indexação dos dados;
- **Manual**: indica que a fonte de dados requer o uso de recursos de programação para prover a indexação dos dados;
- **Personalizável**: indica que a fonte de dados possui suporte nativo à indexação dos dados, conforme parâmetros definidos pelo programador.

Quadro 5: Vantagens e desvantagens das principais fontes de dados persistentes

Categoria	Descrição	Vantagens	Desvantagens
Arquivo Simples	Arquivos sequenciais	• Bom para armazenamento de texto • Manipulação simples • Requer pouco planejamento	• Abstração inexistente • Acesso lento a dados específicos • Busca ineficiente • Compartilhamento difícil • Estrutura textual • Indexação inexistente • Legibilidade ruim • Ordenação inexistente • Padronização inexistente
	Arquivos de acesso aleatório	• Acesso rápido a registros específicos • Busca pode ser eficiente • Estruturado • Manipulação simples • Ordenação manual • Requer pouco planejamento	• Abstração baixa • Compartilhamento difícil • Eficiência depende do programador • Indexação manual • Legibilidade ruim • Limitado à estrutura predefinida • Ordenação manual • Padronização inexistente
	Arquivos binários	• Acesso rápido a blocos específicos • Acesso rápido a bytes específicos • Busca pode ser eficiente • Estruturação possível • Ordenação manual • Sem estruturas predefinidas	• Abstração inexistente • Compartilhamento difícil • Eficiência depende do programador • Estrutura provida pelo programador • Indexação manual • Legibilidade ruim • Manipulação complexa • Ordenação manual • Padronização inexistente • Requer muito planejamento
Arquivos Estruturados	CSV	• Abstração alta • Estruturado • Legibilidade boa • Manipulação simples • Padronização boa • Requer pouco planejamento	• Acesso lento a dados específicos • Busca ineficiente • Compartilhamento difícil • Estrutura textual • Indexação inexistente • Ordenação manual
	XML	• Abstração alta • Estruturado • Legibilidade excelente • Padronização boa • Requer pouco planejamento • Dados e apresentação separados	• Acesso lento a dados específicos • Busca ineficiente • Compartilhamento difícil • Estrutura textual • Indexação inexistente • Manipulação complexa • Ordenação manual
	HTML	• Estruturado • Legibilidade excelente • Padronização boa	• Abstração baixa • Acesso lento a dados específicos • Busca ineficiente • Compartilhamento difícil • Dados e apresentação mesclados • Estrutura textual • Indexação inexistente • Manipulação complexa • Ordenação inexistente
SGBD	MySQL Oracle PostgreSQL SQL Server Outros	• Abstração alta • Compartilhamento fácil • Acesso rápido a registros específicos • Busca altamente eficiente • Busca eficiente • Estrutura personalizável e flexível • Indexação personalizável • Legibilidade excelente • Manipulação simples • Ordenação personalizável	• Padronização ruim
Bancos de Dados	Access Paradox DBase	• Abstração alta • Acesso rápido a registros específicos • Busca eficiente • Busca altamente eficiente • Estrutura personalizável e flexível • Legibilidade excelente • Indexação personalizável • Manipulação simples • Ordenação personalizável	• Compartilhamento difícil • Padronização ruim

Fonte: autoria própria (2015)

APÊNDICE C – INTERFACES QUE PRECISAM SER IMPLEMENTADAS POR UM DRIVER

1. Interface IRowAdapter: traduz lexemas HOML para lexemas específicos da fonte de dados e converte os resultados em entidades do modelo de negócio.

```
namespace DataLibrary.Interfaces
{
    public interface IRowAdapter
    {
        string CreateSelect(Criterion filter);
        string CreateSelect();
        string   CreateSelect(Criterion.Table   joinTable,   Criterion.ValuedField   leftFilter,   Criterion.Field
JoinTableField);
        string CreateSelect(Criterion valueDefinition, Criterion filter);
        string CreateSelect(Criterion.Field field, Criterion filter);
        string CreateSelectCount(Criterion filter);
        string CreateSelectCount();
        string CreateExec(Criterion.StoredProcedure procedure);
        string CreateInsert(Object item, IDictionary<Criterion.Field, IRelationField> keyFieldsValues);
    }

    public interface IRowAdapter<DataType> : IRowAdapter
    {
        IDataFieldInfo PrimaryKeyField { get; }
        IDictionary<string, IDataFieldInfo> PrimaryKeyFields { get; }
        IDictionary<string, IDataFieldInfo> AllFields { get; }
        IDictionary<string, IDataFieldInfo> NonKeyFields { get; }
        IList<IRelationHelper> Children { get; }
        Criterion.ArgumentList ArgumentList { get; set; }
        bool PreventLock { get; set; }
        ICriteriaParser Parser { get; }

        DataType CreateItem(DataRow row, bool primaryKeyOnly, int treeDepth);
        DataType CreateItem(DataRow row, bool primaryKeyOnly);
        DataType CreateItem(DataRow row);
        DataType CreateItem();

        string CreateInsert(DataType item);
        string CreateInsert(DataType item, IDictionary<Criterion.Field, IRelationField> keyFieldsValues);

        bool HasIdentityField();

        void SetIdentityValue(DataType item, Object value);

        string CreateDelete(Criterion filter);
        string CreateDelete();

        string CreateDelete(DataType filter);

        string CreateUpdate(DataType item, Criterion filter);
        string CreateUpdate(DataType item);
        string CreateUpdate(DataType[] items);

        string CreateSelectCount(DataType filter);

        Criterion CriterionFromItem(DataType filter);

        DataType DefaultNullValue { get; }

        string CreateUpdate(DataType oldValue, DataType newValue);
    }
}
```

2. Interface IColumnAdapter: Auxilia no processamento de campos específicos, permitindo o mapeamento bidirecional entre os dados oriundos da fonte de dados e os atributos da entidade mapeada.

```
namespace DataLibrary.Interfaces
{
    public interface IColumnAdapter<DataType> : IColumnAdapter where DataType : new()
    {
        DataType CreateItem(DataRow row);
        DataType CreateItem();
        void FillItem(ref DataType item, DataRow row);

        Criterion CriterionFromItem(DataType item);
        Criterion.ValuedFieldist GetValuedFields(DataType item);
    }

    public interface IColumnAdapter
    {
        Dictionary<string, Criterion.Field> FieldMappings { get; }
        Criterion.FieldList GetFields();
        Criterion.ValuedFieldist GetValuedFields(Object item);
        Object CreateObject(DataRow row);
        Object CreateObject();
        void FillItem(ref Object item, DataRow row);

        Criterion CriterionFromObject(Object item);

        string GetFieldName(string memberName);
    }
}
```

3. Interface ITypeMapper: Permite definir mapeamentos explícitos entre atributos da entidade mapeada e tipos de dados específicos.

```
namespace DataLibrary.Interfaces
{
    public interface ITypeMapper
    {
        Object ValueToItem(Object value);
        Object ItemToValue(Object Item);
    }

    public interface ITypeMapper<ItemType, ValueType> : ITypeMapper
    {
        ItemType ValueToItem(ValueType value);
        ValueType ItemToValue(ItemType Item);
    }
}
```

4. Interface IDatabaseAdapter: Permite repassar ao núcleo de processamento do *driver* os lexemas específicos devidamente processados pelo RowAdapter.

```
namespace DataLibrary.Interfaces
{
    public interface IDatabaseAdapter
    {
        DataTable Select(string command);
        Object ExecuteScalar(string command);
        int Execute(string command);
        int Execute(string command, out Object identityValue);
    }
```

```
        }
```

5. Interface IDao: Permite o envio de lexemas HOML para o *driver* e permite o recebimento de respostas na forma de tipos numéricos primitivos, entidades isoladas ou coleções de entidades.

```
namespace DataLibrary.Interfaces
{
    public interface IDAO<DataType> : IDAO
    {
        CollectionType Select<CollectionType>(Criterion filter, int count, int treeDepth) where CollectionType : IList<DataType>, new();
        CollectionType Select<CollectionType>(Criterion filter, int count) where CollectionType : IList<DataType>, new();
        CollectionType Select<CollectionType>(Criterion filter) where CollectionType : IList<DataType>, new();
        CollectionType Select<CollectionType>() where CollectionType : IList<DataType>, new();

        void Select(IList<DataType> collection, Criterion filter, int count, int treeDepth);
        void Select(IList<DataType> collection, Criterion filter, int count);
        void Select(IList<DataType> collection, Criterion filter);
        void Select(IList<DataType> collection);

        void Select(IList<DataType> collection, DataType filter, int count, int treeDepth);
        void Select(IList<DataType> collection, DataType filter, int count);
        void Select(IList<DataType> collection, DataType filter);

        void Select(ICollection<DataType> collection);
        void Select(ICollection<DataType> collection, Criterion filter);
        void Select(ICollection<DataType> collection, Criterion filter, int count);
        void Select(ICollection<DataType> collection, Criterion filter, int count, int treeDepth);

        void Select(ICollection<DataType> collection, DataType filter);
        void Select(ICollection<DataType> collection, DataType filter, int count);
        void Select(ICollection<DataType> collection, DataType filter, int count, int treeDepth);

        CollectionType Select<CollectionType>(DataType filter) where CollectionType : IList<DataType>, new();
        CollectionType Select<CollectionType>(DataType filter, int count) where CollectionType : IList<DataType>, new();
        CollectionType Select<CollectionType>(DataType filter, int count, int treeDepth) where CollectionType : IList<DataType>, new();

        CollectionType Select<CollectionType>(IEnumerable<DataType> filter) where CollectionType : IList<DataType>, new();
        CollectionType Select<CollectionType>(IEnumerable<DataType> filter, int count) where CollectionType : IList<DataType>, new();
        CollectionType Select<CollectionType>(IEnumerable<DataType> filter, int count, int treeDepth) where CollectionType : IList<DataType>, new();

        int Update(DataType oldValue, DataType newvalue);
        int Update(DataType value);
        int Update(DataType[] values);
        int Update(DataType value, Criterion filter);

        int Delete(DataType filter);

        int Count(DataType filter);

        int Insert(DataType value);
        Criterion CriterionFromItem(DataType item);
    }

}
```

```
namespace DataLibrary.Interfaces
{
    public interface IDAO
    {
        IRowAdapter<Object> RowAdapter { get; }
        System.Type ItemType { get; }

        Criterion.ArgumentList ArgumentList { get; set; }

        ValueType SelectScalar<ValueType>(Criterion valueDefinition);
        ValueType SelectScalar<ValueType>(Criterion valueDefinition, Criterion filter);
        ValueType SelectScalar<ValueType>(Criterion.Field field);
        ValueType SelectScalar<ValueType>(Criterion.Field field, Criterion filter);
```

```
        DataType SelectOne<DataType>();
        ValueType SelectOne<ValueType>(Criterion filter);
        ValueType SelectOne<ValueType>(ValueType filter);

        Object SelectOne();
        Object SelectOne(Criterion filter);
        Object SelectOne(Criterion filter, int treeDepth);

        Object SelectOne(Object filter);
        Object SelectOne(Object filter, int treeDepth);

        IList Select();

        IList Select(Criterion filter);
        IList Select(Criterion filter, int count);
        IList Select(Criterion filter, int count, int treeDepth);
        IList Select(Criterion.Table joinTable, Criterion.ValuedField leftFilter, Criterion.Field JoinTableField, int
treeDepth);

        IList Select(Object filter);
        IList Select(Object filter, int count);
        IList Select(Object filter, int count, int treeDepth);

        void Select(IList collection);
        void Select(IList collection, Criterion filter);
        void Select(IList collection, Criterion filter, int count);
        void Select(IList collection, Criterion filter, int count, int treeDepth);

        void Select(IList collection, Object filter);
        void Select(IList collection, Object filter, int count);
        void Select(IList collection, Object filter, int count, int treeDepth);

        int Insert(Object value);
        int Insert(IList values);

        int Update(Object oldValue, Object newValue);
        int Update(Object value);
        int Update(Object value, Criterion filter);

        int DeleteAll();
        int Delete(Criterion filter);
        int Delete(Object filter);

        int Count();
        int Count(Criterion filter);
        int Count(Object filter);

        Object ExecuteScalar(string query);
        int Execute(string query);

        Criterion CriterionFromItem(Object item);
    }
}
```

6. Interface IProcedure: Permite a execução de procedimentos armazenados na fonte de dados. Caso a fonte de dados não possua suporte nativo a esse tipo de procedimento, o programador poderá optar por disparar uma exceção do tipo NotSupportedException ou por implementar as funcionalidades necessárias.

```
namespace DataLibrary.Interfaces
{
    public interface IProcedure
    {
        Criterion.ArgumentList ArgumentList { get; set; }

        void StoreBatch();
        void DiscardBatch();
        int ExecuteBatch();
        int Execute();
        object ExecuteScalar();
        object ReadObject();
        ListType ReadList<ListType>() where ListType : IList, new();
        void FillList(IList list);
    }

    public interface IProcedure<DataType, AdapterType, DatabaseAdapterType> : IProcedure
        where AdapterType : IRowAdapter<DataType>, new()
        where DatabaseAdapterType : IDatabaseAdapter, new()
    {
        DataType ReadOne();
        IList<DataType> Read();
    }
}
```

7. Interface IRelationHelper: Permite a definição explicita de relacionamento entre duas entidades.

```
namespace DataLibrary.Interfaces
{
    public interface IRelationHelper
    {
        string MasterPropertyName { get; }
        Criterion.Field SlaveField { get; }
        Criterion.Table SlaveTable { get; }
        Type MasterFieldType { get; }
        IDataFieldInfo MasterFieldInfo { get; set; }
        IDictionary<Criterion.Field, IRelationField> RelatedFields { get; }
        IRowAdapter SlaveRowAdapter { get; }
        IEnumerable SubItems { get; set; }
    }
}
```

8. Interface ICriteriaParser: Permite processar cada um dos lexemas HOML e convertêlos em lexemas específicos da fonte de dados mapeada.

```
namespace DataLibrary.Criteria
{
    public interface ICriteriaParser
    {
        /* Tipos primitivos não numéricos  */
        string Parse(Object value);
        string ParseType(Object value);
        string Parse(string value);

        /* Tipos primitivos numéricos não anuláveis */
        string Parse(UInt16 value);
        string Parse(UInt32 value);
        string Parse(UInt64 value);
        string Parse(Int16 value);
        string Parse(Int32 value);
        string Parse(Int64 value);
        string Parse(Byte value);
        string Parse(SByte value);
        string Parse(Char value);
        string Parse(Boolean value);
        string Parse(DateTime value);
        string Parse(Decimal value);
        string Parse(Single value);
        string Parse(Double value);

        /* Tipos primitivos numéricos anuláveis */
        string Parse(UInt16? value);
        string Parse(UInt32? value);
        string Parse(UInt64? value);
        string Parse(Int16? value);
        string Parse(Int32? value);
        string Parse(Int64? value);
        string Parse(Byte? value);
        string Parse(SByte? value);
        string Parse(Char? value);
        string Parse(Boolean? value);
        string Parse(DateTime? value);
        string Parse(Decimal? value);
        string Parse(Single? value);
        string Parse(Double? value);

        /* Listas */
        string Parse(List<object> value);

        /* Critérios */
        string Parse(Criterion value);
        string Parse(Criterion.Field value);
        string Parse(Criterion.ValuedField value);
        string Parse(Criterion.Table value);
        string Parse(Criterion.IJoin value);
        string Parse(Criterion.JoinType value);
        string Parse(Criterion.NamedItem value);
        string Parse(Criterion.FieldList value);
        string Parse(Criterion.ValuedFielddist value);
    }
}
```