UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DETECÇÃO DE OPORTUNIDADES DE REFATORAÇÃO EM BASES DE DADOS RELACIONAIS

DISSERTAÇÃO DE MESTRADO

Luiz Fogliato Júnior

Santa Maria, RS, Brasil
2015

DETECÇÃO DE OPORTUNIDADES DE REFATORAÇÃO EM BASES DE DADOS RELACIONAIS

Luiz Fogliato Júnior

Dissertação apresentada ao Programa de Pós-Graduação em Informática (PPGI),
da Universidade Federal de Santa Maria (UFSM, RS),
como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

Orientadora: Profª. Drª. Deise de Brum Saccol
Co-orientador: Prof. Dr. Eduardo Kessler Piveta

Santa Maria, RS, Brasil

2015

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

DETECÇÃO DE OPORTUNIDADES DE REFATORAÇÃO EM BASES DE DADOS RELACIONAIS

elaborada por
Luiz Fogliato Júnior

Como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

COMISSÃO EXAMINADORA:

Deise de Brum Saccol, Drª.
(Presidente/Orientadora)

Marcelo Soares Pimenta, Dr. (UFRGS)

Sergio Luis Sardi Mergen, Dr. (UFSM)

Santa Maria, 30 de novembro de 2015.

AGRADECIMENTOS

Durante o processo de produção deste trabalho percebi que as noites são mais úteis do que parecem mesmo após exaustivos dias de trabalho. Também percebi o quanto evoluiu o meu conhecimento em inglês por necessitar ler milhares de páginas de bibliografias que não pertenciam à minha língua nativa, pois a prova de suficiência foi a parte fácil. Por ter passado os últimos anos mais concentrado no mercado de trabalho antes de entrar no mestrado e um pouco distante do meio acadêmico, foi necessário reaprender e coordenar a forma que conduzia o trabalho para que a cientificidade fosse sempre mantida. Por fim, o desafio de desenvolver um trabalho que se dispõe a solucionar problemas que por muitas vezes são complexos e trabalhosos ao ponto de ser um "fardo pesado" para profissionais da área, o questionamento sobre a possibilidade de desistência muitas vezes era iminente devido a estes fatores. Este trabalhou me ensinou muito além de questões técnicas, me ensinou o real sentido de perseverança e sou muito grato por isso. Independentemente do julgamento que este trabalho tenha perante aos leitores e avaliadores, tenho a sensação de dever cumprido e uma enorme gratidão de ter a oportunidade de concretizá-lo.

Agradeço à minha família que sempre me incentivou e me ajudou de diferentes formas para ter os impulsos que precisava, principalmente ao meu pai que mesmo estando próximo à morte durante o processo de construção desta dissertação me falava sobre a importância dela para concluir o mestrado, mesmo sem entender o que eu especificamente estava fazendo. Agradeço especialmente à minha esposa Jaqueline pelos momentos de abdicação da minha companhia, de compreensão com meus objetivos para poder seguir em frente, me incentivando e empurrando à finalização mesmo quando pensava que tinha chegado ao limite. Agradeço à minha filha Ingrid que está por nascer e que serviu de combustível para buscar a conclusão deste trabalho com mais força.

Agradeço à empresa Tellfree por ter me concedido os espaços de tempo que precisava para me capacitar cada vez mais através do mestrado.

Por fim, agradeço imensamente pelo profissionalismo, compreensão e suporte dos meus orientadores Deise e Eduardo que souberam me ajudar a seguir o caminho correto para atingirmos as metas, através do fornecimento de materiais necessários, da análise crítica sobre as minhas entregas, do acompanhamento sobre a evolução deste trabalho e, ao mesmo tempo, determinando o que poderia ser relevante ou não.

De verdade, muito obrigado a todos!

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

DETECÇÃO DE OPORTUNIDADES DE REFATORAÇÃO EM BASES DE DADOS RELACIONAIS

AUTOR: LUIZ FOGLIATO JÚNIOR
ORIENTADORA: DEISE DE BRUM SACCOL
CO-ORIENTADOR: EDUARDO KESSLER PIVETA

Muitos sistemas de informação que trabalham com BDR (Bancos de Dados Relacionais) apresentam problemas no projeto de suas bases de dados. Tais falhas podem ser decorrentes de falhas na construção, mudanças de requisitos ou falta de conhecimento por parte da equipe envolvida para produzir esquemas que proporcionem tratamentos mais evolutivos para tais sistemas. A técnica que possibilita fazer mudanças para corrigir imperfeições em bases de dados que geram os problemas citados é conhecida como refatoração (*refactoring*) em bases de dados. Objetivando identificar possíveis oportunidades de refatoração em bases de dados, este trabalho propõe heurísticas que detectam ou auxiliam na detecção de oportunidades de refatoração. Assim, analistas de domínio e profissionais da área de banco de dados poderão identificar, com maior agilidade, os defeitos compreendidos na estrutura e nos dados de um esquema e também tomar as providências necessárias para solucionar esses defeitos.

Palavras-chave: oportunidades de refatoração, bases de dados e heurísticas.

ABSTRACT

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

DETECTION OF REFACTORING OPPORTUNITIES IN RELATIONAL DATABASES

AUTOR: LUIZ FOGLIATO JÚNIOR
ORIENTADORA: DEISE DE BRUM SACCOL
CO-ORIENTADOR: EDUARDO KESSLER PIVETA

Many information systems that work with RDR (Relational Databases) have problems in the design of their databases. Such failures may result from design specification, requirement changes or even lack of knowledge to produce schemes that provide more evolutionary treatments for such systems. The technique that includes making changes to correct imperfections in databases that generate the mentioned problem is known as database refactoring. In order to identify possible opportunities for refactoring databases, this work proposes heuristics that detect or assist in detecting opportunities for refactoring. Thus, domain analysts and data professionals can identify the structure and data bad smells of a scheme and also take the necessary steps to address these shortcomings.

Keywords: refactoring opportunities, databases and heuristics.

LISTA DE FIGURAS

LISTA DE TABELAS E QUADROS

LISTA DE ABREVIATURAS E SÍMBOLOS

AHP	*Analytic Hierarchy Process*
API	*Application Interface Programming*
BD	Base de Dados
BDR	Bancos de Dados Relacionais
BPMN	*Business Process Modeler and Notation*
CASE	*Computer-Aided Software Engineering*
CRUD	*Create, Read, Update and Delete*
CSV	*Comma-separated values*
DAO	*Data Access Object*
DDD	Discagem Direta à Distância
DLL	*Dynamic-Link Library*
DML	*Data Manipulation Language*
DTO	*Data Transfer Object*
ER	Entidade e Relacionamento
ETL	*Extract Transform Load*
GUID	*Globally Unique Identifier*
IDE	*Integrated Development Environment*
LOVeM	*Line Of Visibility Enterprise Modeling*
ORM	*Object Relational Manager*
Regex	Expressão regular
RG	Registro Geral
SGBDS	Sistemas de Gerenciamento de Banco de Dados
SMS	*Short Message Service*
SQL	*Structured Query Language*
TI	Tecnologia da Informação
UML	*Unified Modeling Language*
WSDL	*Web Service Decription Language*
XML	*Extensible Markup Language*

SUMÁRIO

1. INTRODUÇÃO

Conforme OPDYKE (1992), refatoração (do inglês *refactoring*) é a disciplina que compreende em fazer mudanças em código fonte para aperfeiçoar sua estrutura e, consequentemente, melhorar seus atributos de qualidade. A principal premissa da refatoração é manter o comportamento observável da aplicação. De forma semelhante, refatoração em banco de dados é uma mudança na forma que uma base de dados (BD) foi estruturada para aperfeiçoar o projeto, mantendo seu comportamento e semântica.

Para AMBLER e SADALAGE (2006), este tipo de refatoração consiste em dois aspectos:

a) estrutural: quando consiste em ajustar as definições de tabelas, colunas e visões (*views*);
b) funcional: quando consiste em ajustar as definições de rotinas como *stored functions*, *stored procedures* ou *triggers*.

FOWLER (1997) introduziu o conceito de "*code smells*" que é uma forma de categorizar um problema em código que precisa ser refatorado. Exemplos de c*ode smells*, comumente encontrados em códigos fonte são métodos longos, códigos duplicados, nomenclaturas inapropriadas, entre outros. Semelhantemente, existem alguns "*database smells*" (limitações em base dados) comuns que indicam uma refatoração em potencial (AMBLER, 2003). Tais limitações incluem: colunas com vários propósitos de utilização, tabelas com vários propósitos de utilização, dados redundantes, tabelas com muitas linhas, tabelas com muitas colunas, coluna com granularidade que é quando uma coluna apresenta diferentes conceitos dependendo da posição em que se encontra a informação, dentre outras.

AMBLER e SADALAGE (2006) afirmam que grande parte das ferramentas e metodologias que possam contribuir para a evolução de bases relacionais é oriunda de ferramentas de desenvolvimento. Nesse sentido, alguns trabalhos na área de refatoração de código servirão como referência para alguns métodos de refatoração destinados a banco de dados como FOWLER (1999), BARONI et. al (2005), PIVETA (2009), etc. Outros trabalhos, diretamente ligados ao controle evolutivo, de auxílio a mudanças em bases relacionais e propriamente sobre refatoração foram utilizados como base para este trabalho, como por exemplo, FAROULT et. al (2008), TÂN et. al (2011), WALEK et. al (2012), STOREY et. al (1998), entre outros. Trabalhos com foco em desenvolver métodos para automatizar a detecção de oportunidades de refatoração em bases de dados não foram encontrados na

literatura durante as pesquisas realizadas, entretanto tal foco é o que norteará a proposta desta dissertação.

Para exemplificar, um profissional com conhecimentos básicos sobre modelagem de banco de dados, ao analisar um determinado esquema, pode se deparar com situações semelhantes à estrutura original presente na Figura 1. Nesta situação, pode-se identificar a necessidade de adicionar uma tabela de referência para garantir valores válidos para a coluna "UF" e tomar as providências necessárias para alcançar algum estado similar ao demonstrado na estrutura resultante. Tabelas de referência são aquelas que servem para descrever um conjunto de valores, como por exemplo, uma tabela que descreve as siglas dos estados de um país, siglas de unidades de medida, identificadores de status que usuários podem assumir perante um sistema, etc.

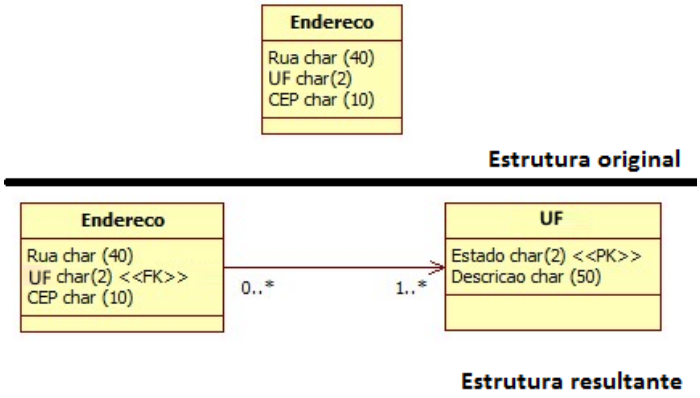

Figura 1 - Exemplo da refatoração "adição de tabela de referência"

Assim como neste exemplo, o problema é que a identificação de oportunidades de refatoração em bases de dados é realizada, majoritariamente, por uma análise humana, onde a falta de ferramentas de apoio às atividades como essas é uma realidade. Desse modo, em bases de dados contendo centenas ou milhares de tabelas, onde algumas tabelas também possuem de centenas a milhões de registros, surge a seguinte questão: como detectar *database smells* de maneira ágil e simplificada? Infelizmente, na atualidade, a resposta para tal questão é normalmente encontrada na ação de profissionais que destinam esforços, muitas vezes manuais e morosos, dependentes do tempo de análise e das habilidades do indivíduo que realiza tal trabalho. Para proporcionar uma resposta que amenize as dificuldades encontradas em cenários como o exposto acima que esta dissertação foi desenvolvida. Para o exemplo aqui citado, bem como de outros *database smells*, nos capítulos seguintes são propostas algumas

heurísticas que auxiliarão na análise dos diferentes elementos (instâncias de tipos de objetos) de bases de dados, visando identificar oportunidades de refatoração e fornecendo informações úteis para alcançar as estruturas resultantes dos problemas detectados.

1.1. Objetivos e Contribuições

O objetivo deste trabalho é a definição de um conjunto de heurísticas para a identificação de oportunidades de refatoração em bases de dados relacionais. Para isso, foram definidas heurísticas para minimizar os seguintes problemas:

- colunas com definições inadequadas, tais como: tipos indevidos, falta de atribuição de valores padrões, dados armazenados com formatos não padronizados, colunas nulas quando deveriam ser não nulas e vice-versa;
- elementos que compõem a estrutura da base de dados e que estejam em desuso;
- tabelas de referência faltantes;
- elementos que tenham nomes que não se enquadram em um padrão de nomenclatura especificado.

Alguns dos problemas citados podem ser detectados durante a fase de projeto, mas a maioria é detectada apenas em bases de dados que já estão em uso. De forma a verificar a viabilidade de implementação de tais heurísticas, foi desenvolvida uma ferramenta chamada "*DatabaseSmellDetector"* e aplicada à base de dados do sistema de *help desk* chamado "Eventum" para avaliar as propostas abordadas.

1.2. Organização do Texto

Esta dissertação está estruturada da seguinte forma:

- capítulo 2 – **Fundamentação teórica** – contextualiza temas importantes que servem de base para o entendimento do que foi proposto. Apresenta alguns trabalhos existentes que estão relacionados ou que possam contribuir, diretamente, à detecção de

oportunidades de refatoração em bases de dados;

- capítulo 3 – **Propostas de heurísticas** – propõe as heurísticas para atender aos objetivos citados anteriormente, juntamente com a diagramação dos fluxos dos processos propostos e algoritmos para automatizá-los;
- capítulo 4 – **Implementação -** propõe uma arquitetura para implementação do sistema e apresenta alguns métodos que viabilizam a execução prática dos algoritmos apresentados no capítulo anterior;
- capítulo 5 – **Experimentos e avaliação de resultados** – descreve os experimentos realizados em uma base de dados de amostra com cada heurística proposta, avaliando sua respectiva credibilidade.
- capítulo 6 – **Conclusões** – apresenta as principais conclusões do trabalho.

2. FUNDAMENTAÇÃO TEÓRICA

A qualidade dos dados tornou-se uma questão-chave em sistemas de gerenciamento. Dados inadequados provocam dificuldades operacionais, bem como perdas financeiras diretas. Além disso, os bancos de dados estão aumentando de tamanho a uma taxa exponencial com a proliferação do comércio eletrônico, a globalização e a economia em tempo real (VASARHELYI e ISSA, 2010). Uma vez que os processos manuais agora são armazenados em bases de dados relacionais, com dados de várias fontes, eles formam ambientes cuja heterogeneidade dá origem a um novo conjunto de problemas como, por exemplo, a entrada incorreta de dados, as informações incompletas, os formatos não padronizados e outras limitações.

Diversas empresas possuem projetos de reestruturação de seus sistemas que necessitam manter suas bases de dados evolutivas e condizentes com seu cenário atual. Ademais, diversos profissionais liberais, estudantes e pesquisadores que desenvolvem projetos de software necessitam que os mesmos não tenham problemas de modelagem. Por estes motivos, é essencial prover métodos que auxiliem no processo decisório acerca do que pode ser refatorado em bases de dados relacionais.

Neste trabalho, as análises dos diferentes tipos de objetos compreendidos por bases de dados (tabelas, colunas, visões, etc.) e dos dados contidos em tais objetos são abordagens utilizadas para identificar oportunidades de refatoração. Conceitos e tecnologias recorrentes são apresentados ao longo deste capítulo, além dos trabalhos relacionados ao tema da pesquisa.

2.1. Noções Gerais de Refatoração em Bancos de Dados

Refatoração em bancos de dados é um processo que envolve mudanças na estrutura da base de dados para aperfeiçoar seu projeto, mantendo o comportamento e a semântica das informações (AMBLER, 2003). Em outras palavras, não se adiciona uma nova funcionalidade ou realiza uma modificação que afete o funcionamento de alguma existente, tampouco se adicionam novos dados ou se modificam seus significados.

A determinação da estratégia de refatoração a ser adotada é proveniente do cenário em que se encontra a base de dados alvo da operação: um ambiente composto por uma aplicação ou por várias aplicações. No cenário de apenas uma aplicação, é possível refatorar e alterar a aplicação e a base de dados simultaneamente, o que possibilita uma aplicação direta dos ajustes desejados. Em cenários compostos de várias aplicações, especialmente daqueles que possuem aplicações que não são de domínio da equipe ou do profissional que está realizando a refatoração, será necessário um período de transição para se chegar à estrutura da base de dados desejada. Em tais casos, o período de transição compreende uma estrutura intermediária, contendo os elementos da estrutura original marcados para serem depreciados (removidos), juntamente com os elementos da estrutura desejada, até que todos os sistemas envolvidos também sejam refatorados para trabalhar com tal estrutura. A remoção dos elementos desnecessários ou obsoletos será realizada com a conclusão da refatoração dos sistemas envolvidos, a fim de evitar uma utilização indevida, confusões e, até mesmo, problemas de desempenho.

É importante estipular as datas para a remoção de elementos obsoletos e desnecessários de acordo com a velocidade de desenvolvimento ou ciclo de *releases* das equipes que adaptarão as aplicações envolvidas. Após o período de transição e a execução suficiente de testes, os elementos indesejados da estrutura inicial e as *triggers* de apoio ao processo de transição (*triggers* que servem para sincronizar a estrutura original com a desejada) deverão ser removidos, resultando assim na estrutura final conforme o exemplo que é citado mais à frente. A Figura 2 demonstra a regra geral para refatorações em cenários com várias aplicações.

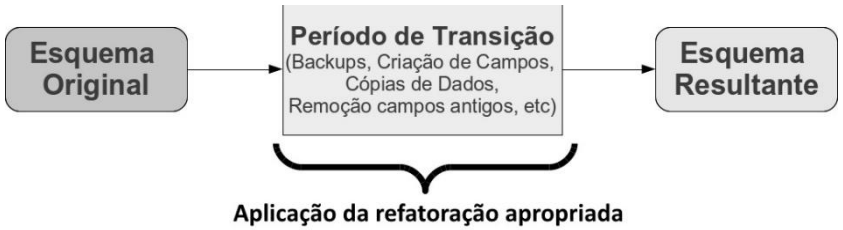

Figura 2 - Regra geral de refatoração
Fonte: Mello 2013

É recomendável que o provimento da regra geral de refatoração seja feito em ambientes apropriados, conhecidos como "*sandboxes*", que são ambientes contendo todas as funcionalidades necessárias para construir, testar e executar um sistema. Por razões de

segurança, é conveniente manter várias *sandboxes* para que o trabalho de um desenvolvedor ou uma equipe não afete, indevidamente, os esforços de outro desenvolvedor ou equipe que esteja trabalhando paralelamente em algum outro projeto que utilize as mesmas estruturas da base e até os mesmos dados. Segundo AMBLER (2002), uma lógica organizacional de distribuição de *sandboxes* para metodologias evolutivas pode ser apresentada da seguinte forma:

a) desenvolvimento: *sandbox* que serve para o desenvolver realizar as alterações e produção do trabalho que envolve a codificação;
b) integração de projetos: normalmente é utilizada para realizar a união de diferentes projetos ou partes de projetos;
c) demonstração: destinada a demonstrações de ajustes e entregas que a equipe ou o desenvolvedor precisa fazer;
d) homologação: deve ter as mesmas configurações do ambiente de produção para que testes sejam realizados da forma mais próxima possível da realidade que o desenvolvimento a ser entregue enfrentará;
e) produção: ambiente no qual os clientes consumirão o que for entregue.

2.1.1. Exemplo

Imagine que você esteja trabalhando com os sistemas de uma instituição bancária e se depare com a estrutura demonstrada na Figura 3. Ao analisá-la, identifica-se que a coluna *saldo* está localizada de maneira equivocada na tabela *cliente*, quando deveria constar na tabela *conta*.

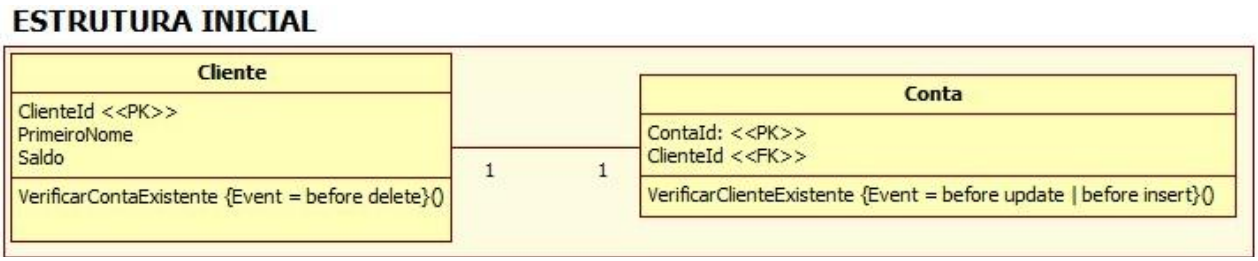

Figura 3 - Estrutura inicial a ser refatorada

Nesse caso, manter a coluna s*aldo* na tabela *cliente* pode trazer transtorno como a impossibilidade de um cliente poder ter várias contas e desejar ver o saldo de cada uma. Produzir mecanismos que detectem refatorações como esta, automaticamente, é praticamente inviável, visto que pode exigir conhecimentos específicos de regras de negócio que motivem tal refatoração. Entretanto, esse exemplo será utilizado para ilustrar como proceder com a aplicação de uma refatoração oportuna.

No cenário de uma aplicação, é recomendado que o desenvolvedor e o administrador de banco de dados realizem uma implementação em par para produzir a solução ou um profissional que tenha conhecimento em ambas as áreas. Essa refatoração, inicialmente, deve ser produzida e testada na *sandbox* de desenvolvimento. Quando as alterações necessárias forem finalizadas deverão ser publicadas e promovidas na *sandbox* de integração de projetos para que a aplicação seja recompilada, testada e corrigida, novamente, se necessário.

Para o problema aqui exposto, deverá ser aplicada a refatoração *Mover Coluna* para se chegar ao resultado da estrutural final conforme a Figura 4. Também é recomendado que os envolvidos nesta operação realizem um desenvolvimento orientado a testes, de forma que todos os testes de aplicações e de banco de dados que validam o campo *Conta.saldo* sejam previamente preparados e executados na *sandbox* de desenvolvimento e depois, postados no controle de gerenciamento de configuração.

Figura 4 - Estrutura final - resultado da refatoração

Após realizar todos os ajustes e testes necessários, por medida de segurança, é importante fazer um *backup* dos dados presentes em *cliente.saldo*, movê-los para *conta.saldo* e promover todo o trabalho realizado na *sandbox* de integração de projetos.

O cenário com várias aplicações é mais complexo porque envolve a publicação de *releases* de diferentes aplicações em uma frequência de tempo variada. O trabalho aqui realizado é semelhante ao exposto no cenário de uma aplicação, exceto pelo fato de que não poderá excluir a coluna *cliente.saldo*, até que todas as aplicações se adequem ao novo

contexto. Durante este período de adequação (transição) será necessário manter as duas colunas com os mesmos dados. Para tanto, as *triggers* “SincronizarSaldoConta” e “SincronizarSaldoCliente” ficarão encarregadas de tal responsabilidade conforme consta na Figura 5.

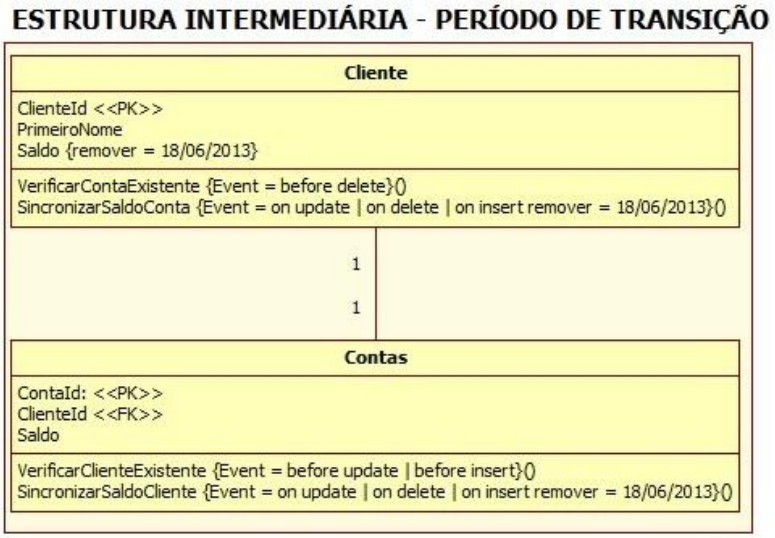

Figura 5 - Estrutura necessária para o período de transição

Após o período de transição e a execução suficiente de testes, a coluna da estrutura inicial e as *triggers* que mantém redundância entre a estrutura inicial e a desejada deverão ser removidas, resultando, assim, na estrutura final ilustrada pela Figura 4.

A estratégia adotada neste exemplo para aplicar a Refatorações “Mover Coluna” poderá ser aplicável a outras refatorações que serão listadas mais à frente. Importante alertar que, na prática, atividades como estas poderão ser ainda mais complicadas quando envolvem sistemas de terceiros cujas decisões para mensurar o período de transição ideal não dependem, exclusivamente, de uma equipe ou organização.

2.1.2. Categorias de Refatoração em Base de Dados

Para organizar os estudos nesta área foram introduzidas seis categorias distintas conforme o Quadro 1 proposto por AMBLER e SADALAGE (2006).

Categoria	**Descrição**	**Exemplo**
Estrutural	Alteração nas definições de uma ou mais tabelas ou *views*	Mover uma coluna de uma tabela para outra
Qualidade de dados	Alteração que aperfeiçoa a qualidade contida na base de dados	Tornar uma coluna não nula para garantir que a mesma sempre conterá um valor
Integridade referencial	Alteração que garante que os valores de um determinado campo da tabela obrigatoriamente existam em outra ou para se certificar que aquele campo não é mais utilizado	Adicionar uma trigger que proporciona a exclusão em cascata entre duas entidades.
Arquitetural	Alteração que aperfeiçoa a maneira como os programas interagem com a base de dados	Substituir um método em Java que está em uma biblioteca compartilhada por uma *stored procedure* para possibilitar que aplicações em outras linguagens possam utilizar tal operação
Rotinas ou métodos	Alteração em uma *stored procedure*, *function* ou *trigger* para aperfeiçoar sua qualidade	Renomear uma *stored procedure* para identificar seu propósito mais facilmente
Transformações (Não é uma refatoração)	Alteração na estrutura da base de dados que têm influência semântica	Adicionar uma nova coluna em uma tabela para adição de novas funcionalidades na aplicação

Quadro 1 - Categorias de refatorações de banco de dados
Fonte: adaptado de AMBLER e SADALAGE, 2006.

Esta categorização não é perfeita, pois algumas refatorações podem coexistir em mais de uma das classificações acima. Também podem ter subclassificações que especifiquem melhor os tipos de refatorações. Mesmo assim, é uma classificação válida e difundida pela literatura de trabalhos nesta área. Este trabalho abrangerá, praticamente, todas as categorias de refatoração, porém com mais enfoque nas seguintes categorias: estrutural, qualidade de dados e integridade referencial.

2.1.3. Refatorações Catalogadas

As refatorações possíveis para bases de dados foram inicialmente sugeridas na obra de AMBLER e SADALAGE (2006) conforme a listagem abaixo:

a) estrutural: Remover Coluna, Remover Tabela, Remover Visão, Mesclar Colunas, Mesclar Tabelas, Mover Coluna, Renomear Coluna, Renomear Tabela, Renomear

Visão, Substituir Coluna, Substituir Relacionamento de Um para Muitos por Tabela Associativa, Substituir Chave Auxiliar por Chave Natural, Fragmentar Coluna, Fragmentar Tabela;

b) qualidade de dados: Adicionar Tabela de Referência, Aplicar Padronização de Código, Aplicar Padronização de Tipo, Consolidar Estratégia de Chaves, Remover Chave Estrangeira e Restrição, Remover Valor Padrão, Modificar Coluna Definida Como não Nula para Nula, Introduzir Restrição e Chave Estrangeira, Padronizar Formato, Introduzir Valor Padrão, Tonar Coluna Não-Nula, Mover Dados, Substituir Tipo de Código por Flag;
c) em métodos da base de dados: Mudar de Interface, Adicionar de Parâmetro, Remover Parâmetro, Renomear Parâmetro, Reordenar Parâmetro, Substituir Parâmetro por Método Explícito, Mudanças Internas;
d) integridade referencial: Adicionar Chave Estrangeira, Remover Chave Estrangeira, Introduzir Exclusão Lógica, Introduzir Exclusão Física.

O trabalho de AMBLER e SADALAGE (2006) norteou o rumo de pesquisas e até mesmo de terminologias adotadas por ferramentas do mercado que tratam da aplicação de refatorações em banco de dados. Um exemplo bastante difundido é o LIQUIBASE (2014), ferramenta que permite controlar o versionamento da base de dados, tudo em conformidade com as necessidades de refatoração que são aplicadas através das operações citadas. Tal listagem também serviu como base para a criação de uma ontologia para especificar refatorações em bases de dados apresentada por FOGLIATO (2014).

Nem todas as refatorações citadas acima serão empregadas nas heurísticas propostas. A resolução de uma limitação da base de dados pode necessitar de uma ou mais refatorações dependo do contexto. O foco deste trabalho é detectar oportunidades de refatoração em vista de imperfeições identificadas pelas heurísticas propostas mais à frente. A indicação das operações a serem executadas para corrigir tais imperfeições é de caráter sugestivo, visando minimizar esforços de análise de dados para produção de *scripts* em *Structured Query Language* (SQL) e demais artefatos que se encarregarão de modificar o que for preciso. Obviamente as indicações apontadas pelas heurísticas não eximem completamente a necessidade de um profissional apto para analisar e julgar a pertinência das mesmas.

2.1.4. Atributos de Qualidade

PIVETA (2009) sugere uma classificação de padrões de refatoração de acordo com atributos de qualidade que podem ser visados ou desejados em um determinado código. Também sugere a criação de perfis de atributos de qualidade para auxiliar na determinação do que deve ser priorizado para aplicar as refatorações encontradas por sua proposta.

De forma semelhante, também é possível determinar algum tipo de classificação para base de dados. No trabalho de PIPINO et. al (2002), ao invés de chamar de atributos de qualidade, os autores classificaram como dimensões de qualidade. As dimensões sugeridas são citadas no Quadro 2.

Dimensão	**Definição**
Acessibilidade	O quanto os dados estão disponíveis, ou fácil e rapidamente acessíveis.
Quantidade apropriada de dados	Até que ponto o volume dos dados é apropriado para a tarefa em questão.
Credibilidade	Até que ponto os dados são creditados como verdadeiros e críveis.
Completude	Até que ponto os dados estão completos e são suficientes tanto em largura quanto em profundidade para a tarefa em questão.
Representação concisa	O quão compacta está a representação dos dados.
Representação consistente	Até que ponto os dados estão sendo representados no mesmo formato.
Facilidade de manipulação	O quão fácil é manipular e aplicar os dados em diferentes tarefas.
Livre de erros	Até que ponto os dados são confiáveis e corretos.
Facilidade de interpretação	Até que ponto os dados estão em linguagens, símbolos e unidades apropriadas e as definições estão claras.
Objetividade	Até que ponto os dados são imparciais.
Relevância	Quão úteis e aplicáveis são os dados para a tarefa em questão.
Reputação	Até que ponto os dados são considerados em termos de seu conteúdo e origem.
Segurança	Até que ponto os dados são restritos para manter sua segurança.
Atualidade	O quão atualizado estão os dados para a tarefa em questão.
Facilidade de entendimento	Até que ponto os dados são facilmente compreendidos.
Valor adicionado	Até que ponto os dados são benéficos e retornam vantagens em seu uso.

Quadro 2 - Atributos de qualidade para elementos de bases de dados
Fonte: adaptado de PIPINO et. al, 2002.

Os atributos de qualidade citados são bons parâmetros para servir de referência para avaliar as necessidades de aplicação das refatorações que serão sugeridas por este trabalho. O profissional encarregado desta atividade deverá contrabalanceá-los com os esforços e os custos necessários para aplicar as refatorações julgadas como pertinentes.

2.1.5. Estratégias de Refatoração em Bancos de Dados

Esta seção tem por objetivo descrever algumas experiências de AMBLER e SADALAGE (2006) que podem ajudar a diminuir esforços em executar o processo. Essas experiências incluem os seguintes pontos:

a) pequenas mudanças são mais fáceis de aplicar;
b) identificar refatorações individuais;
c) implementar uma grande mudança de modo que se torne várias pequenas mudanças;
d) ter uma tabela de configuração da base de dados;
e) priorizar a utilização de triggers para fazer a sincronização de dados ao invés de views e aplicações do tipo consoles (shell scripts);
f) determinar um período de depreciação suficiente;
g) simplificar a negociação com outras equipes;
h) encapsular os acessos à base de dados (BARONI, ABREU e CALERO, 2005).
i) estar pronto para facilmente configurar um ambiente para base de dados.
j) evitar duplicidade de códigos SQL.
k) colocar todas as alterações em um controle de mudanças;
l) tomar cuidados com políticas.

2.1.6. O Processo de Refatoração de Banco de Dados

A Figura 6 mostra um diagrama de atividades em UML para ilustrar o processo de refatoração que será abordado nesta seção. Nele é importante observar a comunicação entre uma atividade e outra e o ciclo como um todo até o momento da publicação de uma

refatoração. A seguir, cada uma das atividades que compõe o diagrama será, brevemente, descrita.

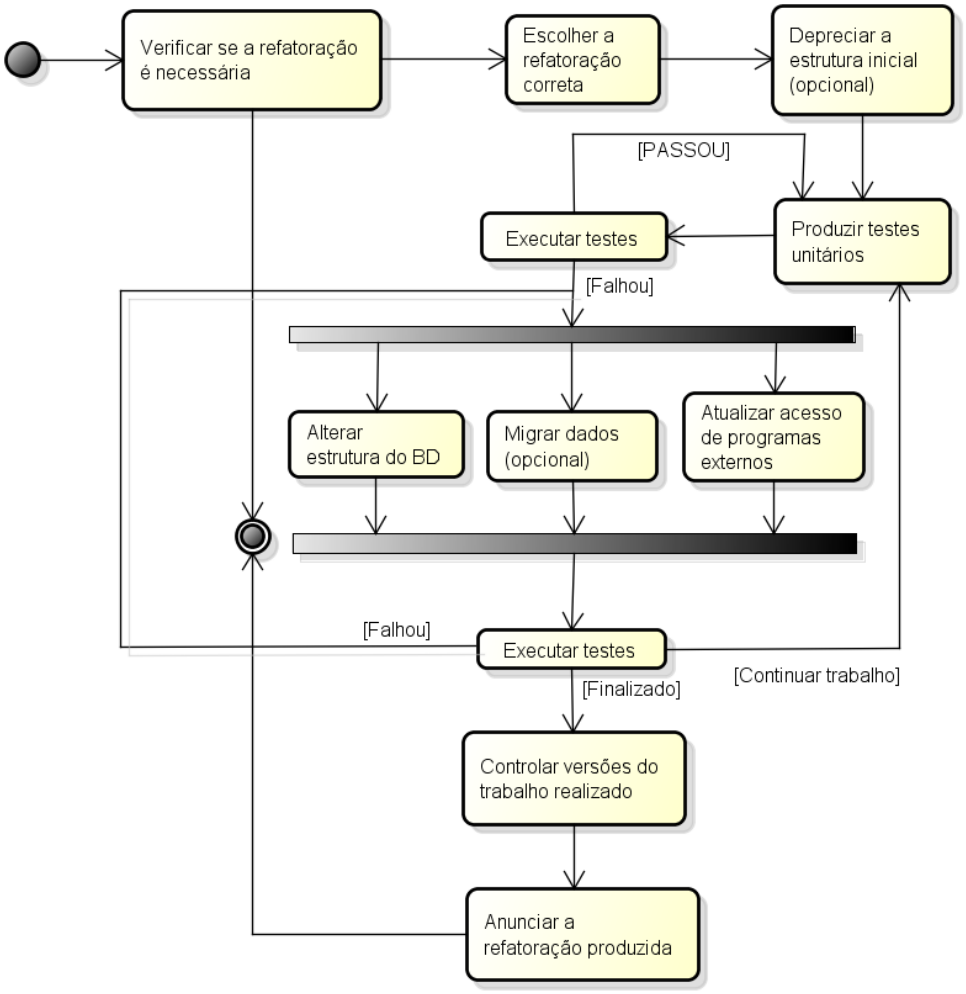

Figura 6 - UML - diagrama de atividades
Fonte: adaptado de AMBLER e SADALAGE, 2006.

O trabalho parte de uma solicitação que o desenvolvedor recebe para corrigir um defeito, e na sequência executa as seguintes atividades:

a) verificar se a refatoração no banco de dados é apropriada;
b) escolher o tipo de refatoração mais apropriado para solucionar o problema de forma que o analise a fundo e constate qual o melhor caminho a ser seguido para chegar à solução;
c) depreciar a estrutura original. AMBLER (2003) alerta que se várias aplicações acessam a base de dados, provavelmente será necessário trabalhar com a hipótese de que não é possível refatorar todas estas aplicações simultaneamente. Por esse motivo, é necessário um período de transição, onde a porção da estrutura que está

sendo alvo da operação deve ser marcada como tal. A Figura 7 demonstra o ciclo de vida para refatorações que necessitam desta providência;

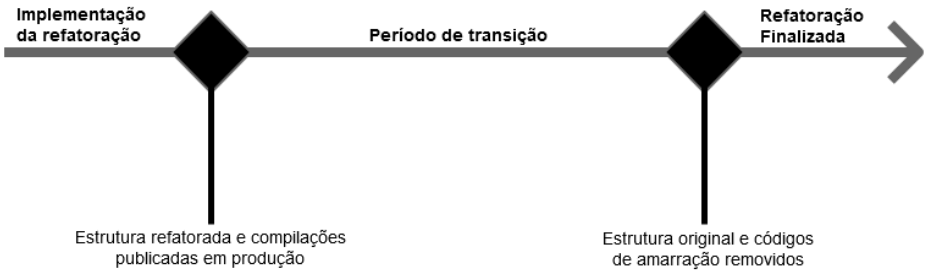

Figura 7 - Ciclo de vida de refatorações em cenários com várias aplicações
Fonte: adaptado de AMBLER e SADALAGE, 2006.

d) testar as alterações antes, durante e depois de todo processo. Provavelmente necessitará escrever os seguintes testes:
   – testes para validar a estrutura da base de dados,
   – testes de validação da migração de dados,
   – testes da codificação de programas externos;
e) modificar a estrutura da base de dados;
f) produzir *scripts* responsáveis pelas mudanças estruturais da base de dados seguindo as convenções do projeto;
g) migrar os dados para a estrutura modificada se necessário*;*
h) modificar o acesso de programas externos;
i) executar testes de regressão;
j) colocar todo o trabalho realizado em um gerenciamento de controle de configuração;
k) anunciar a refatoração produzida.

2.2. Método AHP

Para a identificação de elementos obsoletos foi empregado o método AHP (*Analytical Hierarchy Process*) (SAATY, 1990), de forma semelhante ao exposto por PIVETA (2009), porém no contexto de refatoração em bases de dados. Essa é uma técnica matemática de apoio

às decisões que envolvem várias alternativas para um dado problema. Visto que a atividade de refatorar é situacional, neste trabalho o AHP é usado para expressar o peso que cada situação deve ter sobre um determinado padrão que caracteriza uma oportunidade de refatoração.

Resumidamente, neste método deve seguir os seguintes passos:

1. Definir os pesos ou hierarquia que cada critério deve ter em relação aos demais. Para isso, o método AHP apresenta o Quadro 3 predefinido para auxiliar na definição de tal hierarquia.

Valor	Importância relativa
1	Mesma Importância
2	Ligeiramente mais importante
3	Fracamente mais importante
4	Fracamente para moderadamente mais importante
5	Moderadamente mais importante
6	Moderadamente para fortemente mais importante
7	Fortemente mais importante
8	Largamente mais importante
9	Absolutamente mais importante

Quadro 3 - Quadro predefinido de pesos para critérios do AHP
Fonte: adaptado de Piveta 2009

Imagine que existam os critérios C1, C2 e C3, por exemplo. É possível definir uma ordem de importância, conforme consta no Quadro 4, a partir do Quadro 3.

Valor	Importância
2	C1 é ligeiramente mais importante que C2
3	C3 é fracamente mais importante que C1
4	C3 é fracamente para moderadamente mais importante C2

Quadro 4 - Definição de critérios

2. Organizar os critérios em uma matriz. Por exemplo, supondo que um dado processo decisório utiliza três critérios com os valores exemplificados no Quadro 4, é possível gerar a matriz demonstrada na Figura 8.

$$\mathcal{M} = \overbrace{\begin{bmatrix} 1 & 1/2 & 3 \\ 2 & 1 & 4 \\ 1/3 & 1/4 & 1 \end{bmatrix}}^{c_1 \quad c_2 \quad c_3} \begin{matrix} c_1 \\ c_2 \\ c_3 \end{matrix}$$

Figura 8 - Matriz de critérios
Fonte: adaptado de Piveta (2009)

3. Após, é necessário gerar o vetor que irá determinar o peso de cada critério. Esse vetor é conhecido como "*eigenvector*" (SAATY, 2003). Para gerar este vetor é necessário calcular a matriz quadrada resultante da matriz anterior, somar as linhas e dividir o total da soma de todas as linhas pela soma de cada linha para gerar um vetor que normalize o peso de cada critério. A Figura 9 ilustra o processo.

$$\overbrace{\begin{bmatrix} 3.0000 & 1.7500 & 8.0000 \\ 5.3333 & 3.0000 & 14.0000 \\ 1.1667 & 0.6667 & 3.0000 \end{bmatrix}}^{squared\ matrix} = \overbrace{\begin{bmatrix} 12.7500 \\ 22.3332 \\ 4.8333 \end{bmatrix}}^{sum\ of\ rows} = \overbrace{\begin{bmatrix} 0.3194 \\ 0.5595 \\ 0.1211 \end{bmatrix}}^{normalized}$$

$$\textbf{Total} \quad 39.9165 \quad 1.0000$$

Figura 9 - Normalização para gerar o *eigenvector*
Fonte: adaptado de Piveta (2009)

4. Repetir o processo até que a iteração corrente apresente o mesmo resultado da iteração anterior com uma precisão de dígitos após a vírgula determinada pelo usuário deste método. Para o exemplo em questão foram utilizados apenas quatro dígitos após a vírgula.
5. Depois, é necessário montar uma matriz para cada critério. Essas matrizes devem representar o peso que cada alternativa possui em relação ao critério em questão conforme os exemplos abaixo. Uma observação importante neste ponto é que se pode utilizar valores quantitativos e não somente qualitativos como o que está sendo exemplificado aqui. Imagine que se tenha quatro alternativas para a refatoração e que os critérios de seleção das refatorações desejadas tenham as mesmas definições hierárquicas de C1, C2 e C3 citadas. Assim, os atributos de qualidade definidos como critérios seriam: completude (C1), facilidade de

entendimento (C2) e credibilidade (C3). As alternativas para a refatoração são: adicionar tabelas de referência (A1), padronizar formatos (A2), eliminar elementos obsoletos (A3) e minimizar elementos redundantes (A4). Dessa forma, é possível estabelecer que as alternativas possuem a matriz de pesos sob um determinado contexto representada no quadro 5.

Concluído o passo anterior, deve-se calcular o *eigenvector* de cada critério como demonstrado no Quadro 5.

Critério	Matriz					*Eigenvector*
Completude		$A1$	$A2$	$A3$	$A4$	
	$A1$	1	$1/3$	$1/5$	$1/5$	0.0745
	$A2$	3	1	2	2	0.4004
	$A3$	5	$1/2$	1	1	0.2626
	$A4$	5	$1/2$	1	1	0.2626
Facilidade de entendimento		$A1$	$A2$	$A3$	$A4$	
	$A1$	1	2	$1/3$	4	0.2877
	$A2$	$1/2$	1	3	2	0.3166
	$A3$	3	$1/3$	1	2	0.2983
	$A4$	$1/4$	$1/2$	$1/2$	1	0.0974
Credibilidade		$A1$	$A2$	$A3$	$A4$	
	$A1$	1	7	$1/4$	$1/5$	0.1327
	$A2$	$1/7$	1	$1/8$	$1/9$	0.0356
	$A3$	4	8	1	$1/2$	0.3279
	$A4$	5	9	2	1	0.5038

Quadro 5 - Matrizes de comparações por critério e respectivos *eigenvectors*

6. Para encontrar a melhor alternativa em torno de um determinado processo decisório, monte uma matriz com os vetores (*eigenvector*) gerados no passo anterior. Multiplique tal matriz pelo *eigenvector* do passo 3 e encontrará a melhor alternativa dentre os critérios informados.

$$\begin{array}{c|ccc} & C1 & C2 & C3 \\ A1 & .0745 & .2877 & .1327 \\ A2 & .4004 & .3166 & .0356 \\ A3 & .2626 & .2983 & .3279 \\ A4 & .2626 & .0974 & .5038 \end{array} \times \begin{array}{c} Eigenvector \\ 0.3194 \\ 0.5595 \\ 0.1211 \end{array} = \begin{array}{c} Ranking \\ 0{,}2008 \\ 0{,}3093 \\ 0{,}2904 \\ 0{,}1993 \end{array}$$

Para o exemplo dado, a alternativa vencedora é a A2 (padronizar formatos), pois foi aquela que apresentou a melhor pontuação no *ranking* com base nos critérios informados.

2.3. Trabalhos Relacionados

Trabalhos que tratam, especificamente, de automatizar ou agilizar a detecção de oportunidades de refatoração em bases dados não foram encontrados durante as pesquisas realizadas para esta produção. Ocorrências desse tipo seriam de grande utilidade para servir como referência para comparações do que está sendo proposto. Entretanto, além dos trabalhos já citados nesta fundamentação, outros contribuem diretamente para este trabalho e merecem atenção, como por exemplo, trabalhos na área de *Extract Transform Load* (ETL), deduplicação de dados, análise semântica dos elementos da base de dados, entre outros.

Existem trabalhos na área de limpeza de dados que podem contribuir diretamente com alguns objetivos introduzidos neste trabalho, como por exemplo, RAMAN e HELLERSTEIN (2002), ANDRITSOS et. al (2006) e outras bibliografias sobre ETL na parte de transformação de dados. Entretanto, pouquíssimos fazem referência às abordagens que focam na limpeza através da identificação daqueles elementos em desuso. A limpeza de dados visa detectar e remover anomalias dos dados com o objetivo de aumentar/melhorar a sua qualidade (RAHM e DO, 2000). Para OLIVEIRA et. al (2006), o processo de limpeza de dados não pode ser executado sem o envolvimento de um perito do domínio, uma vez que a detecção e correção de anomalias requer conhecimento especializado. Durante a pesquisa deste trabalho, foram levantadas diversas hipóteses para determinar tabelas em desuso. Uma delas é seguindo estratégias semelhantes à proposta por JUNIOR (2004), onde, no contexto deste trabalho, poderia ser proposto algum recurso que monitorasse a base de dados com frequência de forma a "aprender" sobre aquilo que pode ou não estar em uso. É uma alternativa interessante, entretanto, necessitaria de tempo suficiente para coleta de dados e, consequentemente, não forneceria respostas imediatas, além disso, aumentaria, drasticamente, a complexidade da solução.

Neste trabalho, alguns processos usam expressões regulares, como por exemplo, para identificar irregularidades em formatos de dados e ajustar a nomenclatura de determinados elementos. O emprego de expressões regulares é comum em validadores de formulários web, justamente para verificar se determinados campos estão em conformidade com formatos desejados antes de submetê-los ao servidor. Logo, utilizar expressões regulares, também no contexto deste trabalho, mostra-se uma alternativa apropriada visto que é para mesmo objetivo, ou seja, identificar expressões que não estejam em um formato desejado. Embora

seja uma alternativa apropriada e eficiente, em contrapartida exige do usuário conhecimento de como manipular tais expressões.

Também foram pesquisados diferentes trabalhos correlatos a fim de conseguir embasamentos suficientes para montar alguma proposta de detecção da refatoração de padronização de tipos, proposta por AMBLER e SADALAGE (2006), de forma automatizada. Dentre os trabalhos mais significativos pesquisados estão os que abordam a carga e a conversão de dados. Segundo KIMBAL e CASERTA (2006), definir uma estratégia eficiente de carga e distribuição de dados de forma fragmentada para analisar e/ou transformar dados é essencial para garantir processos que possam incidir sobre grandes massas de dados.

2.4. Considerações Finais

Como o objetivo deste trabalho é auxiliar na detecção de imperfeições e falhas de projetos de bases de dados, propondo as devidas refatorações para torná-los mais evolutivos, é necessário ter os conhecimentos abordados neste capítulo para entender e analisar assuntos que circundam os próximos temas. Grande parte das abordagens dos próximos capítulos detém-se à análise da estrutura de esquemas de bases de dados independentemente do cenário que tais esquemas estejam incluídos. As produções de estratégias para conduzir os períodos de transições, bem como os ajustes arquiteturais necessários para facilitar a execução das refatorações, não pertencem ao escopo principal deste trabalho. Tais assuntos foram citados com o intuito de promover o conhecimento necessário para aplicar as sugestões de refatorações resultantes das heurísticas que serão demonstradas.

As categorias de refatorações trabalhadas nas abordagens desta dissertação são: estrutural, qualidade de dados e integridade referencial. Refatorações arquiteturais, de rotinas ou métodos não serão alvo deste trabalho, pois são refatorações que, normalmente, implicam em regras negócio e conhecimento das aplicações consumidoras da base de dados e são refatorações que também fogem do escopo do trabalho.

Grande parte das operações de refatorações catalogadas é empregada nas propostas heurísticas para alcançar os atributos de qualidade que cada uma visa. Entretanto, não são utilizadas em sua totalidade e, sim, conforme a demanda necessária para os processos propostos. Por fim, os trabalhos relacionados serão mais aprofundados ao longo desta dissertação através das propostas em que os mesmos fazem parte da formulação de soluções.

3. HEURÍSTICAS PARA BUSCAS DE OPORTUNIDADES DE REFATORAÇÃO EM BANCOS DE DADOS RELACIONAIS

Heurística é definida como qualquer técnica que melhora o desempenho de execução sobre uma determinada tarefa de resolução de problemas e, geralmente, é uma aproximação do conhecimento (RUSSELL e NORVIG, 2002). Sendo assim, neste capítulo serão propostas técnicas que visam melhorar a detecção de oportunidades de refatoração em bancos de dados relacionais a partir de determinadas imperfeições.

São propostas heurísticas para as seguintes imperfeições: elementos em desuso, falta de tabelas de referência, elementos com a nomenclatura imprópria para um padrão especificado e colunas com definições inapropriadas, como por exemplo, tipos e tamanhos inadequados, colunas com dados não padronizados, colunas que podem ter algum valor padrão definido, colunas definidas como não nulas, quando deveriam ser nulas e vice-versa.

Elas foram escolhidas, porque quando solucionadas, melhoram diversos atributos de qualidade de uma base de dados. Em cada proposta, são descritos os atributos de qualidade visados, mas não serão detalhados, pois a análise de impacto não é o objetivo deste trabalho. Também foram escolhidas porque foram aquelas passíveis de alguma espécie de automatização durante a execução deste trabalho.

Foram feitas algumas investigações em outras heurísticas para detectar oportunidades de refatoração, resultantes de outras imperfeições que poderiam ser adicionadas a este trabalho. Por exemplo, identificar estruturas não normalizadas, ou seja, aquelas que não se encontram na primeira, segunda ou terceira forma normal. Porém durante as pesquisas, não surgiram alternativas para solucionar as questões mencionadas de forma a estabelecer processos que não necessitem intensamente da intervenção de um analista de domínio e que reconheçam todas as operações de refatorações necessárias para atingir os diferentes níveis de normalização.

Também foram feitas algumas pesquisas para minimizar elementos redundantes, mas foi retirado do escopo do projeto porque este tipo de imperfeição necessita de técnicas de análise de similaridade, o que aumentaria a dimensão do trabalho. Questões nesse sentido serão abordadas na sessão de trabalhos futuros.

As heurísticas apresentadas são descritas em um formato focado na detecção das oportunidades de refatoração e, simultaneamente, abordam questões técnicas necessárias para viabilizar tais detecções. A preocupação com verificações necessárias e que envolvam

grandes volumes de dados é uma questão que deve ser refletida pelo usuário das propostas aqui abordadas em todas as heurísticas. Por esse motivo, deve-se considerar a possibilidade de processamentos paralelos e distribuídos para os algoritmos propostos. Por exemplo, para que sejam viáveis as verificações em colunas de tabelas que possuem grandes volumes de registros, a divisão do processo de análise dos registros em *threads* na implementação do algoritmo de detecção de tamanhos e tipos de colunas inadequados pode ser uma necessidade.

Por fim, foi escolhido este conjunto de propostas porque tratam de problemas comumente encontrados em sistemas de diferentes proporções e cenários. Além disso, a concretização das oportunidades de refatorações apontadas resulta em bases de dados com menor incidência de informações indevidamente representadas, com mais integridade referencial, com um estado mais condizente com seu propósito de uso, entre outros aspectos que beneficiam sua utilização.

Tais técnicas são descritas no formato de processos que objetivam aproximar-se de soluções que sejam suficientemente aplicáveis a diferentes cenários. Os processos propostos são apresentados usando *Business Process Model and Notation* (BPMN) (BROCKE e ROSERMANN, 2010). Foi utilizado o BPMN porque possibilita uma diagramação apropriada de processos de forma simples, detalhada e com uma boa apresentação. Cada processo diagramado é detalhado neste capítulo, também, os algoritmos que servem para auxiliar a automatização de tais processos são detalhados para que, no capítulo seguinte, sejam abordadas algumas implementações.

3.1. Identificar Elementos em Desuso

As operações de refatoração “Remover Tabela” e “Remover Coluna” normalmente são motivadas pelo desuso de tais elementos e por se tornarem obsoletos. Elementos em desuso são aqueles elementos que não são utilizados na base de dados por diferentes razões, como por exemplo, as regras de negócio das aplicações mudaram, os projetos de software que utilizariam tais elementos não foram postos em práticas ou não foram implementados conforme o estipulado durante as atividades de projeto, etc.

Os tipos de objetos da base de dados que são foco do processo de busca desta heurística são tabelas, *views* e colunas. Outros tipos de objetos como *stored procedures, triggers, stored function,* etc., não são analisados pelo processo proposto neste trabalho, pois

requerem uma análise funcional e também, porque durante as pesquisas realizadas, não foram encontrados embasamentos suficientes para serem considerados por esta heurística.

A solução para esse tipo de imperfeição resulta, principalmente, em operações categorizadas como estruturais. Manter apenas o que é necessário e útil em uma base de dados poderá apresentar diversos benefícios, contribuindo, direta ou indiretamente com as seguintes dimensões de qualidade: acessibilidade, quantidade apropriada de dados, credibilidade, representação concisa, facilidade de manipulação, livre de erros, facilidade de interpretação, relevância, reputação, atualidade e facilidade de entendimento.

3.1.1. Regra Geral

Como regra geral, considerando que *X* é um limiar de tempo informado pelo usuário do processo, para definir que um elemento *E* esteja em um estado de desuso pelo menos uma das seguintes condições deve ser satisfeita:

a) *E* não possui dados;
b) *E* não possui dados e não foi alvo de exclusões a partir de *X*;
c) *E* possui dados, mas não possui dados a partir de *X*;
d) *E* não possui dados a partir de *X* e não é alvo de inserções, exclusões ou atualizações a partir de *X*;
e) caso a condição “c)” e “d)” seja satisfeita e *E* não é alvo de seleções de dados a partir de *X*.

O limiar de tempo deve ser determinado por um conhecedor do domínio que deseja verificar os elementos obsoletos. A condição “a)” servirá para identificar aqueles elementos normalmente criados sem necessidade, que muitas vezes não chegaram a ser utilizados. A condição “e)” deve servir como um reforço à condição “c)” e “d)” e não decisiva, pois podem haver casos de elementos que são alvos de seleções de dados, porque as aplicações que os utilizam ainda não foram adaptadas para desconsiderá-los em seus códigos. Em contrapartida, também pode ser alguma tabela de referência que é utilizada, propositalmente, apenas para seleção de dados, como por exemplo, uma tabela para armazenar os estados de um país, onde pode ficar anos sem sofrer inserção ou atualização de registros.

Cada condição possui um determinado peso para determinar sua relevância. Para o cálculo do peso de cada uma foi utilizando o método AHP a partir da matriz que contém a hierarquia entre uma condição e outra, representada no Quadro 6.

	a)	b)	c)	d)	e)
a)	1	1/3	2	1/2	1/5
b)	3	1	5	2	1/3
c)	½	1/5	1	1/3	1/5
d)	2	1/2	3	1	1/2
e)	5	3	5	2	1

Quadro 6 - Matriz da hierarquia das regras de elementos obsoletos

O vetor de pesos resultante (*eigenvector*) para a matriz do Quadro 3 é a=0,0985; b=0,2466; c=0,0563; d=0,1656; e=0,4330. Os pesos deste vetor servem para medir a relevância das oportunidades de refatoração de elementos obsoletos. Por exemplo, determina-se que, em um dado esquema de bases de dados, deva-se marcar como elementos obsoletos apenas aquelas oportunidades com peso maior ou igual a 0,09. Logo, aquelas tabelas que possuem dados, mas que não possuem dados a partir de uma data *X* (condição “c)” não serão levadas em consideração.

3.1.2. Processo de Detecção

De acordo com a proposta deste trabalho, o processo de detecção de elementos obsoletos parte de uma lista de elementos que deve ter sua obsolescência analisada a partir da verificação do que está especificado na regra geral para determinar a identificação deste tipo de imperfeição. A construção desta lista pode ser aleatória, porém, se este processo está sendo executado por alguém que conhece os elementos da base de dados e suspeita quais elementos não são úteis na base de dados, é aconselhável que tal lista contenha apenas tais elementos. Dessa forma, o processo terá uma execução menos custosa e servirá de apoio à decisão para confirmar desconfianças neste sentido. As atividades que comõem tal processo são demonstradas através da Figura 10.

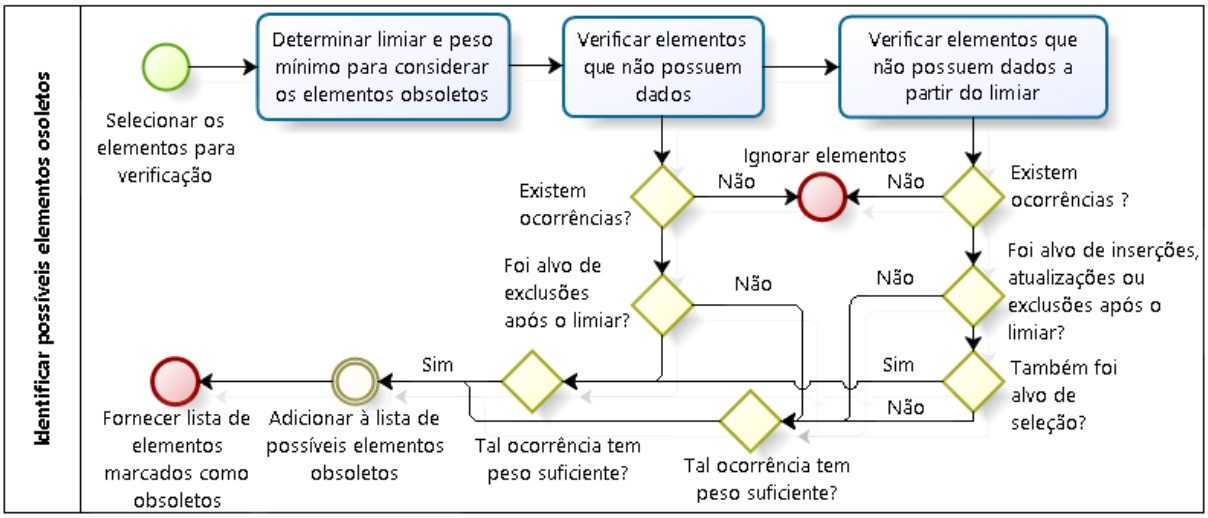

Figura 10 - BPMN - identificação de elementos obsoletos

A determinação de um limiar e peso mínimo são critérios essenciais para considerar a relevância das oportunidades de acordo com o cenário que se deseja aplicar tal processo. Uma vez definidos esses valores, basta verificar as regras sobre a lista definida inicialmente. Esse processo não necessariamente precisa ser executado através de um algoritmo, pode ser realizado manualmente, constatando aqueles que não possuem dados (regras “a)” e “b)”) ou que não possuem a partir do limiar determinado (regras “c”, “d)” e “e)”) a partir da lista inicial.

Após, deve-se conferir a característica de cada ocorrência, ou seja, verificar, por exemplo, se uma tabela sem registros não foi alvo recentemente de inserções e exclusões para garantir que nenhuma aplicação ainda esteja a utilizando para alguma carga de dados temporária. Tal situação acentua o desuso da tabela e serve também, de indicador para fazer uma remoção com menores impeditivos. Para cada ocorrência e suas respectivas características, deve-se identificar qual o peso correspondente de acordo com o *eigenvector* e tal peso é maior ou igual ao que foi definido para, então, ser adicionados à lista de possíveis elementos obsoletos. Obviamente, é um processo moroso de ser feito manualmente. Por este motivo, o Algoritmo 1 fornece agilidade para a identificação desse tipo oportunidade.

3.1.3. Algoritmo

Já que cada condição que determina um elemento como possivelmente em desuso tem um peso, um dos parâmetros de entrada no algoritmo que verifica tais condições é o peso mínimo desejável. Como pode ser observado no Algoritmo 1, cada teste condicional que trabalha com a variável de peso *weight* serve para a verificação de uma condição citada anteriormente e que, consequentemente, faz a chamada ao método responsável por obter tal tipo de valor através do método *AHPDeprecated*. O método *AHPDeprecated* possui a implementação necessária do AHP para obter o peso de cada regra.

Algoritmo 1: Detectar elementos obsoletos

Entrada: Elements {lista de elementos que terão sua obsolescência verificada}
lim {limiar de tempo para determinar a obsolescência de um elemento}
p{peso mínimo para considerar um elemento obsoleto}

Saída: dep {lista de itens marcados como possivelmente obsoletos}

```
1. for each <elem> in Elements do
2.   if elem.Type  = "Table" or elem.Type = "Column" or elem.Type = "View" then
3.     weight ← 0
4.     if !ElemHaveData(elem) then
5.       weight ← AHPDeprecated(NoData)
6.       if !ElemHaveUpdateInsertOrDelete(elem, lim) then
7.         weight ← AHPDeprecated (NoDataNoUpdates)
8.       end if
9.     else
10.      haveDataAfterTreshold ← ElemHaveDataAfterMoment(elem, lim)
11.      if !haveDataAfterTreshold then
12.        weight←AHPDeprecated(NoDataAfterTreshold)
13.        if !ElemHaveUpdateInsertOrDelete(elem, lim) then
14.          weight ← AHPDeprecated(NoDataAfterTresholdAndNoUpdates)
15.          receiveSelect ← ElemReceiveSelectAfterLim(elem, lim)
16.          if (!receiveSelect) then
17.            weight ← AHPDeprecated(NoDataAfterTresholdAndNoAcess)
18.          end if
19.        end if
20.      end if
21.    end if
22.    if weight >= p then
23.      dep.add(elem)
24.    end if
25. end for
26. Retorna elementos obsoletos
```

Os métodos que servem para fazer verificações de existência de dados, incidência de atualizações ou seleção de dados a partir de um determinado momento, como por exemplo, *ElemHaveData*, *ElemHaveUpdateInsertOrDelete*, *ElemReceiveSelectAfterLim*, respectivamente, e outros, têm sua implementação dependente do tipo de banco de dados, pois alguns oferecem funções apropriadas para tais propósitos e outros exigirá do desenvolvedor a produção de tais funções.

3.2. Detectar Possíveis Tabelas de Referência Faltantes

A adição de uma tabela de referência parte da necessidade de restringir os valores que uma coluna pode assumir em um determinado conjunto e, principalmente, descrever o significado destes valores. Os elementos alvos desse processo são colunas que apresentam recorrência nos valores armazenados e que não possuem referência (chave estrangeira) para uma tabela que represente o conjunto de valores possíveis que tal coluna pode assumir.

A adição de tabela de referência, embora envolva a operação de adicionar uma tabela, é categorizada como uma refatoração de qualidade de dados porque objetiva a descrição dos dados contidos em uma determinada coluna. A adição de tabelas de referência pode contribuir com atributos de qualidade como: credibilidade, representação concisa, representação consistente, facilidade de interpretação, facilidade de entendimento, quantidade apropriada de dados, completude e valor adicionado.

3.2.1. Regra Geral

Para evitar que tabelas de referência sejam sugeridas de maneira indevida é necessário que seja levado em consideração o reaproveitamento de tabelas existentes, pois pode haver alguma tabela no esquema em questão que satisfaça o conjunto de valores para uma determinada coluna, porém não existe uma chave estrangeira envolvendo tal coluna que formalize o relacionamento. Isso pode ocorrer pelo esquecimento durante o projeto da base de dados e devido à falta de preocupação com a integridade referencial ou qualquer outro motivo. Assim, tal possibilidade é prevista na regra proposta.

Como uma tabela de referência possui um conjunto restrito de valores, a maneira de identificar a oportunidade da adição de uma tabela de referência é verificar os agrupamentos que podem ser gerados a partir de uma coluna alvo de análise do processo.

Por exemplo, imagine uma tabela destinada a armazenar usuários de um sistema que tenha milhões de registros e uma coluna chamada “status”. Ao realizar uma consulta SQL, que agrupe por tal coluna, observa-se que são gerados apenas quatro agrupamentos com valores 1, 2, 3 e 4 respectivamente. Da mesma forma, ao verificar o código da aplicação que trabalha com tal coluna, um enumerador que descreve os valores é identificado como: 1 = ativo, 2 = cancelado, 3 = bloqueado e 4 = inativo. Logo, uma tabela de referência pode ser criada para que essa informação seja registrada na base de dados e para que fique garantido que a coluna possa assumir valores entre um e quatro.

Para situações como essas, alguns bancos de dados dão suporte ao tipo “ENUM”, que funciona, justamente, como um enumerador na base de dados. Entretanto, como nem todos os bancos de dados apresentam esta possibilidade, se for necessário descrever melhor um determinado conjunto de valores, a utilização de tabelas de referência pode ser uma alternativa mais apropriada.

Com base na situação exemplificada e, supondo que uma coluna *C* participe deste processo de análise, *R* seja a quantidade de registros da tabela que contém *C*, um limiar *X* que define o número máximo de agrupamentos gerados e *G* como sendo o número de agrupamentos gerados a partir de *C*, é possível definir as seguintes premissas para identificar uma oportunidade de adição de tabela de referência:

a) *C* não é uma chave estrangeira;
b) *C* não é do tipo ENUM;
c) *C* não é de um tipo temporal (*date*, *timestamp*, *datetime*, etc....)
d) $G <= X$;
e) $G < R$
f) $G > 2$

Para verificar se uma tabela *T* pode ser uma possibilidade de referência para *C*, primeiramente, é necessário definir o percentual máximo *P* de valores em *C* que não possuem suas respectivas ocorrências na chave primária de *T*. Então, considerando que *Z* é a quantidade de valores não encontrados na chave primária de *T*, *Y* é a quantidade total de registros que *T* possui e *K* é o percentual de registros não encontrados em *T*. Logo, também é possível definir a seguinte regra para determinar a possibilidade de reaproveitamento de uma tabela existente como referência de *C*:

a) $K = (Z*100) / Y$;

b) $K <= P$;

Obviamente, o fato de os valores de uma determinada coluna coincidirem com os valores da chave primária de outra tabela não é determinante para considerar tal tabela para reaproveitar como referência, mas é um forte indício para um analista de domínio avaliar a possibilidade. Conferir se o nome de *T* possui alguma relação com *C* e se de fato os dados *T* descrevem a referência presente em *C*, são tarefas complementares que deverão ser realizadas neste processo para garantir a decisão.

3.2.2. Processo de Detecção

A detecção de tabelas de referência faltantes, representada na Figura 11, parte da determinação dos parâmetros de entrada especificados no Algoritmo 3. Neste passo inicial, quando se têm conhecimento sobre o esquema alvo desta análise, para poupar verificações e demais processos necessários, é importante determinar apenas as colunas que se têm suspeita de que possam representar a necessidade de adição de tabela de referência. Na sequência, devem ser retiradas de tal lista aquelas colunas que são chaves primárias ou chaves estrangeiras, para que os esforços deste processo incidam especificamente sobre aquelas colunas que não representam vínculos relacionais.

Para restringir ainda mais a listagem definida inicialmente, deve ser verificada a quantidade de agrupamentos gerados por cada coluna. Isso pode ser feito através de seleções semelhantes ao exemplificado no método *CountGroups* do Algoritmo 3. Esta verificação é essencial para descartar aquelas colunas que não apresentam algum padrão em seus valores e que consequentemente não representam uma possibilidade de adicionar uma tabela de referência. Por exemplo, uma coluna que pertença a uma tabela com mais de vinte mil registros e tenha uma diversidade de mais dezenove mil agrupamentos é pouco provável que tenha chances de necessitar de uma tabela de referência. Isso porque elas normalmente representam um conjunto restrito de tipos de valores que podem ser assumidos e raramente atingem tal volume de registros.

Com a conclusão desta atividade se tem em mãos o principal artefato deste processo que é uma listagem com as possíveis colunas que precisam de um detalhamento melhor sobre os dados que podem representar. Basta agora verificar se cada uma dessas colunas pode

referenciar alguma tabela existente no esquema ou se realmente necessitamos de novas tabelas de referência. Logo, as próximas atividades servem para mapear ambas as possibilidades sobre cada coluna pertencente à lista.

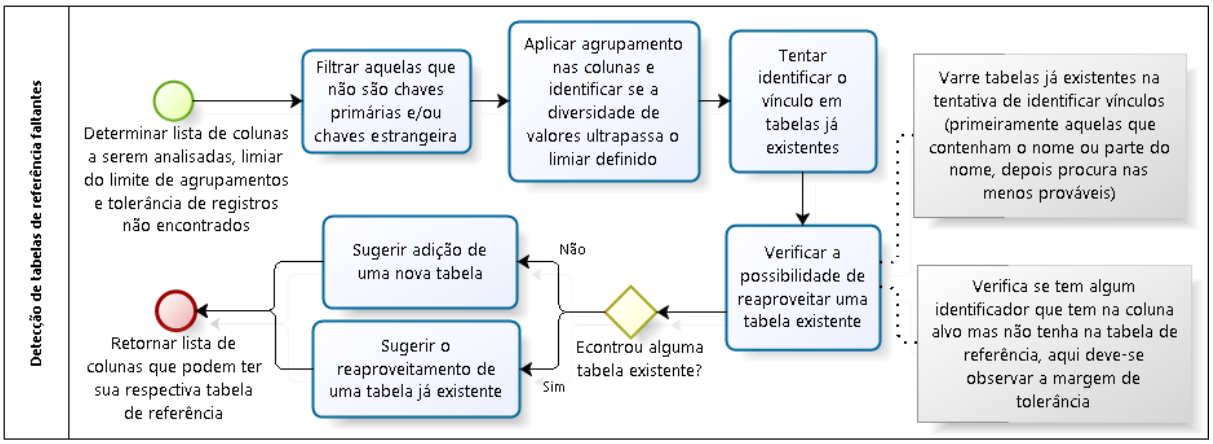

Figura 11 - BPMN - detecção de tabelas de referência faltantes

A verificação da possibilidade de reaproveitamento de uma tabela existente pode ser feita através de instruções SQL semelhantes à apresentada no método *ComparePrimaryKeyWithColumn* (linha 7 do Algoritmo 2). É de suma importância neste processo não se deter apenas aos resultados de tais comandos, pois pode haver comparações com tabelas que não pertençam ao contexto da coluna em questão, mas que não tenham ocorrência de registros encontrados por simples coincidência. Por motivos como este, é necessário avaliar o nome das tabelas e informações nelas contidas para que alguma relação seja identificada com a coluna alvo do processo, e por consequência determinar se uma tabela pode ser reaproveitada.

3.2.3. Algoritmo

O propósito do Algoritmo 2 é identificar oportunidades de adição de tabelas de referência e, simultaneamente, verificar a possibilidade de reaproveitar tabelas existentes. Este algoritmo consiste em percorrer uma lista de colunas verificando se a quantidade de agrupamentos gerados para cada coluna é condizente para a criação de uma tabela de referência. Em caso positivo, as tabelas que compõem o esquema relacional ao qual a coluna

pertence são percorridas através de um laço de repetição, verificando a possibilidade de reaproveitamento de acordo com a regra geral.

Algoritmo 2 : Detectar adição de tabelas de referência

Entrada: lc {lista de colunas que terão seus dados avaliados}
X {limiar que determina a quantidade máxima de agrupamentos}
P{percentual para tolerar registros não encontrados}

Saída: refTab {lista de colunas que podem ter sua respectiva tabela de referência}

```
1. for each <col> in lc do
2.   if (col.key = null) and (col.type != Type.Enum) and  (!isTemporal(col.type) then
3.     G ← CountGroups(col)
4.     if (G <= X) and (G < GetCountFromTable(col.tableName) then
5.       tables ← GetTablesFromSchema(col.schema)
6.       for each <tab> in tables do
7.         Z ← ComparePrimaryKeyWithColumn(tab, col)
8.         K ← (Z*100)/GetCountFromTable(tab.name)
9.         if (K <= P) then
10.          possibleReferenceTables.add(tab)
11.        end if
12.      end for
14.        refTab.add(col, possibleReferenceTables)
15.   end if
16.  end if
17. end for
18. Retorna lista de colunas e tabelas de referência existentes
```

Caso não seja encontrada a possibilidade de reaproveitamento de uma tabela existente e se a coluna em questão tiver uma quantidade de agrupamentos que represente um conjunto de valores que podem ser referenciados, a coluna em questão é marcada para a adição de uma nova tabela de referência, como pode ser observado na linha 14 do Algoritmo 2.

Como é possível observar, normalmente as heurísticas propostas neste trabalho devem ser aplicadas sob um determinado esquema de base de dados. Entretanto, de acordo com a linha 5 do Algoritmo 2 é possível aplicar o processo em colunas de diferentes esquemas, pois o carregamento das tabelas do esquema ao qual a coluna faz parte acontece dinamicamente.

Para atender a lógica os seguintes métodos devem ser implementados:

a) *CountGroups* (linha 3): é um método que recebe como parâmetro uma coluna e verifica quantos agrupamentos são gerados a partir da mesma. Por exemplo, para verificar a quantidade de agrupamentos da coluna *status_id* pertencente a tabela *usuarios*, esse método executará a seguinte instrução SQL e contará a quantidade de

linhas retornadas:

SELECT COUNT(id) AS QuantidadeDeAgrupamentos **FROM** usuarios **GROUP BY** status_id;

b) GetTablesFromSchema (linha 5): serve para obter as tabelas do esquema que a coluna alvo pertence;

c) *ComparePrimaryKeyWithColumn* (linha 7): obtém a quantidade de registros não encontrados na chave primária da tabela alvo da verificação da possibilidade de reaproveitamento. Por exemplo, para verificar se uma tabela chamada *usuarios_status* pode servir de referência para a coluna *status_id*, tal método deve verificar a quantidade de registros retornada pelo seguinte comando SQL:

SELECT COUNT(id) **AS** QuantidadeDeRegistros **FROM** usuarios_status **WHERE** id **NOT IN** (**SELECT** status_id **FROM** usuarios **GROUP BY** status_id).

O retorno obtido pelo método *ComparePrimaryKeyWithColumn* é transformado em percentual e comparado com *P*, pois a ocorrência de algum registro não encontrado não necessariamente significa que a tabela não pode ser utilizada como referência, isto porque talvez seja necessário verificar questões de inconsistências devido justamente à falta de integridade referencial. Obviamente incidências deste tipo devem ter uma prioridade menor no processo de análise sobre aquelas ocorrências com a inexistência desta questão. Por este motivo, este percentual, assim como o limiar de agrupamentos, deve ser muito bem mensurado pelo usuário deste algoritmo. A implementação do método *GetTablesFromSchema*, também dependerá do tipo de banco de dados que se esteja trabalhando.

3.3. Identificar Colunas com Tipos ou Tamanhos Inadequados

Esta heurística visa identificar colunas que possam estar com uma definição de tipo que não seja a ideal para o seu propósito ou para a dimensão dos dados que ela armazena. Comumente, uma alteração de tipo de dados ou tamanhos de um determinado tipo de coluna se categoriza como qualidade de dados, porque objetiva atender uma determinada necessidade de representatividade de informações.

Assim como em qualquer refatoração, antes desse processo, os *backups* dos dados e da estrutura original devem ser providenciados. Isso é importante, essencialmente, naqueles

casos em que o banco de dados não suporta comandos que possibilitem uma alteração direta do tipo da coluna e forcem a operação de remoção seguida da adição da coluna para concretizar a alteração.

Definições de tipos e seus respectivos tamanhos de maneira adequada, além de proporcionarem significativos ganhos de desempenho interno da base de dados, tráfego de informações gerado pela mesma e demais questões, também contribuem com todos os atributos de qualidade comumente destacados na literatura, mesmo que indiretamente.

3.3.1. Regra Geral

A identificação de tipos e de tamanhos de campos inadequados consiste em realizar testes de conversões e avaliação do tamanho que pode ser aplicado a tais campos de acordo com os dados existentes.

Uma coluna qualquer *Ci*, pertencente a uma tabela *T* pode ter seu tipo *T_Ci* ajustado se pelo menos duas das seguintes condições:

a) se o tipo *T_Ci* da coluna *Ci* for mais abrangente, ou seja, *longtext*, *text*, *tinytext*, *varchar*, *char*, etc., é necessário executar os seguintes testes condicionais adicionais, visto que pode armazenar dados de outros tipos:
   – se a maior parte dos registros contidos em *T* puder ser convertida de *T_Ci* para *Tx_Ci*, onde *Tx_Ci* é XML, é sugerida a operação de substituir um campo LOB por tabelas;
   – se a maior parte dos registros contidos em *T* puder ser convertida de *T_Ci* para *Te_Ci*, onde *Te_Ci*, é um tipo mais específico (numérico, booleano ou datas);
b) se o tipo *T_Ci* da coluna *Ci* for específico (numérico ou datas), tentativas de conversões para um tipo *Te_Ci*, onde *Te_Ci* é um tipo mais específico ainda como, por exemplo, um campo decimal que poderia ser um inteiro;
c) se a maior parte dos registros contidos em *T.T_Ci* for utilizada como *flag* a seguinte condição deve ser satisfeita:
   – se os dados de *Ci* quando agrupados geram apenas dois grupos (*A* e *B*), logo, por exemplo, *A* pode corresponder a *True* e *B* a *False*, possibilitando a conversão de *T_Ci* para *Tb_ci*, onde *Tb_ci* é um tipo booleano;
d) se for possível aplicar o tipo *Tm_Ci* ao invés de *T_Ci*, onde *Tm_ci*, é um tipo de

mesmo propósito, porém de menor tamanho/capacidade, como por exemplo, um *bigint* quando poderia ser um *smallint* ou *char(100)* quando poderia ser *char(2)*;

e) se alguma das condições anteriores for satisfeita e se *Ci* possuir dados.

As condições necessárias para a identificação de tamanho de campo servem de auxílio na condição "d)" para ajustar o tipo de campo. Para identificar um tamanho de campo condizente com o estado atual das informações contidas na coluna, é necessário que o usuário deste processo informe um limiar em percentual de tolerância *Y* em relação ao tamanho do maior registro atualmente existente *Z*. Por exemplo, suponha que a tabela *T* tenha dois milhões de registros. A coluna *Ci* está definida como *text* e a maior ocorrência desta coluna possua 100 caracteres; se o usuário definir um limiar de tolerância *Y* de 10%, então será sugerido um tipo que contemple o tamanho adequado *X* de cento e dez caracteres, ou seja, essa coluna poderá ser definida como *varchar*(110). Portanto, é possível definir a seguinte função para identificar o tamanho correto: $X = Z+(Y*100/Z)$.

3.3.2. Processo de Detecção

O processo de identificação de tipos ou tamanhos inadequados em grandes esquemas de bases de dados sem ferramentas de apoio é um processo moroso por demandar bastante tempo de análise. Entretanto, existem meios para minimizar esforços.

Além do cuidado necessário sobre uma especificação condizente do percentual de tolerância de conversões falhas, a especificação do percentual de determinação do tamanho de novos registros *Y* também deve ser analisada para que os tamanhos alocados para cada tipo não sejam condizentes apenas com estado atual da base de dados, mas, principalmente, com uma prospecção correta sobre o tamanho que os dados contidos na coluna possam ter futuramente.

Durante este trabalho, constatou-se que isso pode ser feito através de uma análise que identifique o crescimento das informações contidas em colunas sobre um período de tempo, como por exemplo, um ano. Ao analisar especificamente uma coluna, outra alternativa é estabelecer a mesma proporção para os registros futuros igual à diferença entre o tamanho da maior ocorrência em relação à segunda maior ocorrência.

Essas são apenas sugestões. Resumidamente, é importante ter uma referência para especificar este parâmetro de forma a não oferecer riscos de ocasionar falhas em futuras

inserções por falta de capacidade e, ao mesmo tempo, não ter um tamanho desproporcional ao uso destino à coluna.

As verificações em torno de cada categoria de tipo (textual, temporal ou numérica) nada mais são que testes de conversão para tipos mais específicos que possam representar as informações contidas nas colunas com uma maior precisão. Os tipos mensurados para cada categoria no diagrama ilustrado na Figura 12 são apenas exemplos mais conhecidos, pois cada tipo de banco de dados possui seu conjunto específico de tipos suportados.

Alguns bancos de dados possibilitam a utilização de CAST para se testar a possibilidade de conversão dos dados de uma determinada coluna para outro tipo. Para aqueles que não possuem tal suporte, é necessário carregar os dados contidos em tais colunas a partir de algum método implementado em uma linguagem "tipada" e executar as instruções propostas no método *TryToConvertToMoreSpecificType* (linhas 9 e 13 do Algoritmo 3).

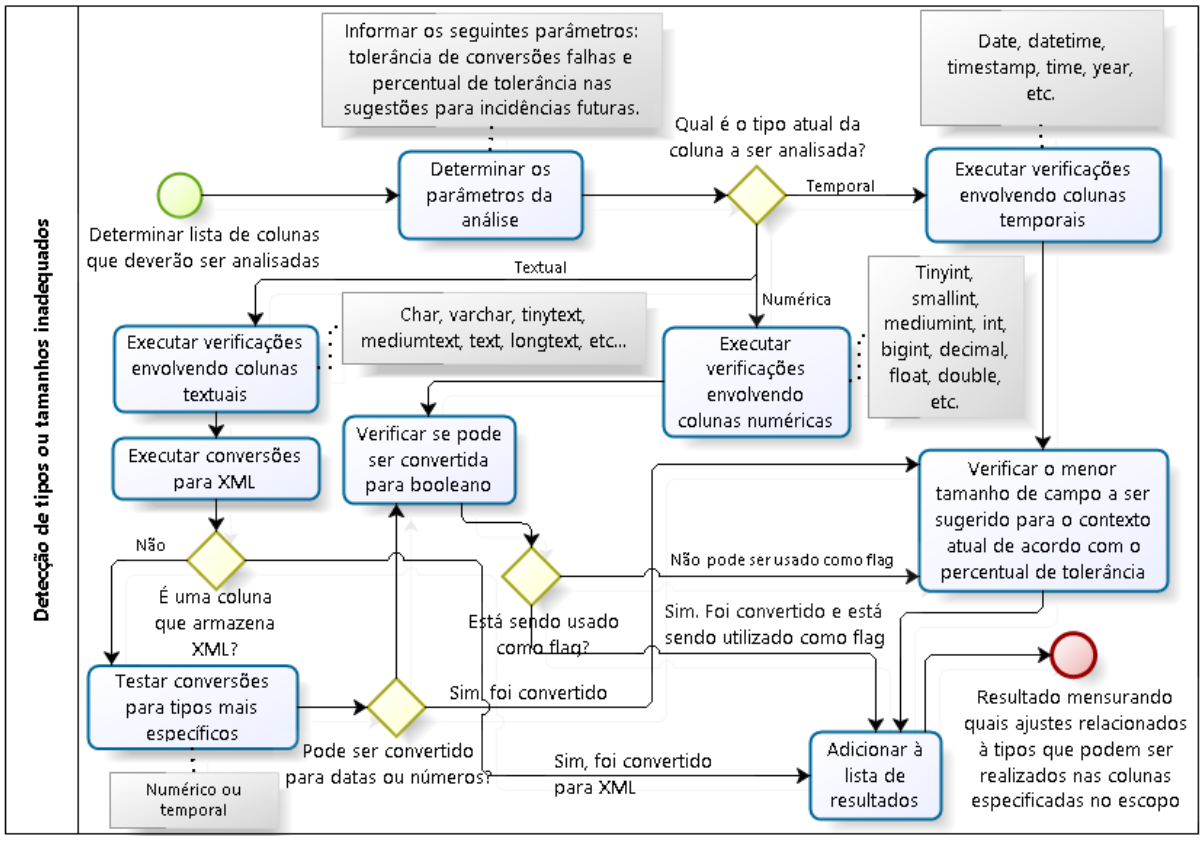

Figura 12 - BPMN - detecção de tipos ou tamanhos inadequados

Da mesma forma, a atividade de verificar o menor tamanho de campo a ser sugerido para o contexto atual, de acordo com o que foi especificado em *Y*, poderá necessitar de uma ferramenta de apoio, caso o banco de dados em questão não ofereça meios. A aplicação da

redução de tamanho deve ser feita com cautela, pois normalmente os sistemas de gerenciamento de banco de dados (SGBDS) não aplicam validações em tal operação. Isso faz com que, por exemplo, em uma coluna do tipo *varchar* (100) e com registros de até noventa caracteres, ao aplicar uma redução para *varchar* (60) ocasionará corte de até trinta caracteres e consequentemente perda de dados.

Para verificar se uma coluna pode ser substituída por uma *flag* deve-se seguir a mesma estratégia descrita no método *CanBeAFlag* (linha 16 do Algoritmo 3). Por exemplo, se a coluna "status" de uma tabela *usuarios* apresentar apenas valores "1" (*ativo*) e "4" (*inativo*), exclusivamente, tal coluna será substituída por outra do tipo booleano chamada "ativo" e quando verdadeira representa que um determinado usuário ativo em determinado sistema e falso quando o usuário inativo.

Por fim, de acordo com o processo proposto na Figura 12, cada coluna que foi passível de algum tipo de conversão deve ser adicionada a uma listagem com o de tipo ou tamanho, a fim de que a aplicação de tais oportunidades seja avaliada mais tarde por um especialista.

3.3.3. Algoritmo

O algoritmo proposto para auxiliar na detecção de tipos ou de tamanhos inadequados consiste em realizar testes de conversões de tipos sobre os dados armazenados em cada coluna mensurada na lista de entrada.

Para agilizar o processo e tornar tal algoritmo computacionalmente viável para verificações das regras apontadas em colunas com grande volume de dados, o parâmetro de entrada *PTC* é obrigatório. Ele serve para determinar tolerância de conversões falhas em relação ao total de registros contidos na tabela que contém a coluna alvo do processo sob um determinado tipo. Por exemplo, dada uma coluna *Ci* do tipo *VarChar*(100) pertencente a uma tabela *T* com 10 milhões de registros, se for determinado que *PTC* é igual a dez por cento do total de registros de *T*, logo, os testes de conversões para o tipo *Decimal* não precisam continuar se atingir um milhão de conversões falhas.

Logicamente, a determinação do percentual de conversões falhas também deve ser avaliada com atenção sobre o esforço necessário para providenciar solução para tais casos. Para o exemplo citado, dez por cento de tolerância é muito, pois significa que será necessário

providenciar ajustes em um milhão de registros para que se possa aplicar a refatoração indicada.

Algoritmo 3 : Detectar colunas com tipos ou tamanhos inadequados

Entrada: lc {lista de colunas que serão verificadas}
PTC {percentual de tolerância para conversões falhas}
Y { percentual de determinação do tamanho de novos registros }

Saída: CTA {lista de colunas e seus respectivos ajustes}

```
1. for each <col> in lc do
2.   if (ElementHaveData(col)) then
3.     continue;
4.   end if
5.   if (GetCategoryOfType(col.type)=CategoryType.Textual) then
6.     if (TryToConvertToXML(col, PTC)) then
7.       CTA.add(col, "Substituir LOB por tabelas ou converter para o tipo xml")
8.     else
9.       newType ← TryToConvertToMoreSpecificType(col, PTC)
10.    end if
11.  end if
12.  if ((GetCategoryOfType(col.type)=CategoryType.Numeric) or
       (GetCategoryOfType(col.type)=CategoryType.Temporal)) then
13.    newType ← TryToConvertToMoreSpecificType(col, PTC)
14. end if
15. if ((!CTA.contains(col)) and (newType = null)) then
16.   if (CanBeAFlag(col)) then
17.     CTA.add(col, "pode ser convertida para flag/booleano")
18.   else
19.     newType ← TryToReduceSize(col, Y);
20.   end if
21.  end if
22.  if (newType != null) then
23.    CTA.add(col, "Converter para o tipo "+newType)
24.  end if
25. end for
26. Retorna lista de colunas e a respectiva sugestão de ajuste
```

Por esse motivo, é aconselhável que seja considerada a tolerância mínima possível para que as verificações em torno de tipos menos prováveis para atribuição da coluna sejam descartados com maior agilidade e, para que o tipo mais apropriado seja realmente sugerido.

Chamadas ao método *GetCategoryOfType* (linhas 5 e 12) servem para conferir se o tipo da coluna alvo do processo trata-se de um tipo textual, numérico ou temporal, com suporte do banco de dados em questão.

O método *TryToConvertToXML* (linha 6) verifica se os dados contidos na coluna podem ser convertidos para XML. Caso isso ocorra, a sugestão de refatoração é substituir um *large object* (LOB) por uma tabela para que as informações concentradas de maneira textual fiquem de maneira relacional, ou seja, com tabelas e colunas que representem adequadamente os nós, atributos e suas relações existentes na base de dados para facilitar a construção de consultas e demais operações sobre tais dados. Em alguns tipos de bancos de dados, como por exemplo, *SQL Server*, é possível atribuir o tipo XML propriamente dito e isso possibilita, inclusive, o uso de instruções XPath em conjunto com SQL.

O método *TryToConvertToMoreSpecificType* (linhas 9 e 13) encapsula a lógica que percorre as linhas da tabela, tentando converter os dados da coluna para um tipo mais específico. Para que este processo tenha precisão, as conversões falhas são contabilizadas para encontrar o tipo que menos apresentou falhas de conversão e que, no mínimo, se enquadre dentro do limite de tolerância especificado.

Caso não encontre um tipo mais específico, é verificada se a coluna pode ser convertida para um tipo booleano. Para tal, uma seleção agrupada pela coluna deve ser executada pelo método *CanBeAFlag* (linha 16) e verifica-se se o resultado foi divido em dois grupos de valores para determinar a possibilidade de substituição da coluna atual por uma *flag*.

Se as verificações anteriores falharam em encontrar alguma sugestão de ajuste, o método *TryToReduceSize* (linha 19) tenta encontrar o tamanho ideal da coluna. Para isto é necessário exe cutar uma consulta SQL que obtenha o maior registro aplicando a função especificada nas regras anteriores. Por fim, nenhum dos métodos ou verificações citadas será executado se a coluna não possuir dados.

3.4. Detectar Valores Padrões

A operação de adição de valor padrão, normalmente, é motivada a partir da percepção da predominância (maior quantidade de registros) de um determinador valor. Geralmente a definição de um valor padrão exige profundidade no conhecimento sobre o domínio e são definições, muitas vezes, identificadas de maneira empírica sobre o que predominará em uma determinada coluna durante o processo de modelagem da base de dados

A solução para este tipo de imperfeição é categorizada como estrutural, podendo haver influência sobre a categoria de qualidade de dados pelo fato de normalmente, alimentar a coluna com um valor conhecido e válido. Este tipo de oportunidade de refatoração contribui com as seguintes dimensões de qualidade: credibilidade, completude, livre de erros, objetividade, relevância e reputação.

3.4.1. Regra Geral

Como esta heurística parte da identificação da predominância de um determinado valor, basta agrupar os registros de uma tabela *T* pela coluna *Ci* e definir um limiar *Y*, onde *Y* é um percentual mínimo de predominância para um valor a ser considerado padrão em relação ao total de registros *Z* em *T*. Partindo da premissa de que *VP* é a quantidade de ocorrências do valor *V*, que compreende a maior quantidade de registros agrupados por *Ci* em *T*, é possível identificar o percentual de predominância *P* utilizando a função abaixo:

*P = (VP*100) / Z*

Logo, se *P >=Y*, então *V* pode ser considerado como padrão.

Por exemplo, uma base de dados possui uma tabela destinada a armazenar os produtos ofertados e tem uma coluna chamada *TipoDeServico*. Ao fazer uma seleção na tabela, agrupando por tal coluna, conclui-se que 80% dos registros possuem o valor “pré-pago”. Se o limiar de predominância for 60%, é possível inferir que o valor “pré-pago” pode ser definido como padrão da coluna *TipoDeServico*.

3.4.2. Processo de Detecção

Este processo não apresenta muitos detalhes além do que está especificado na regra geral, exceto pelo fato de que aquelas ocorrências que não atingirem o percentual mínimo de predominância obviamente precisam ser ignoradas. Também, nem sempre a razão motivadora da definição de um valor padrão será a sua predominância. Entretanto, isso é pouco usual e por isso outras razões não são levadas em consideração neste trabalho. A Figura 13 ilustra o processo proposto.

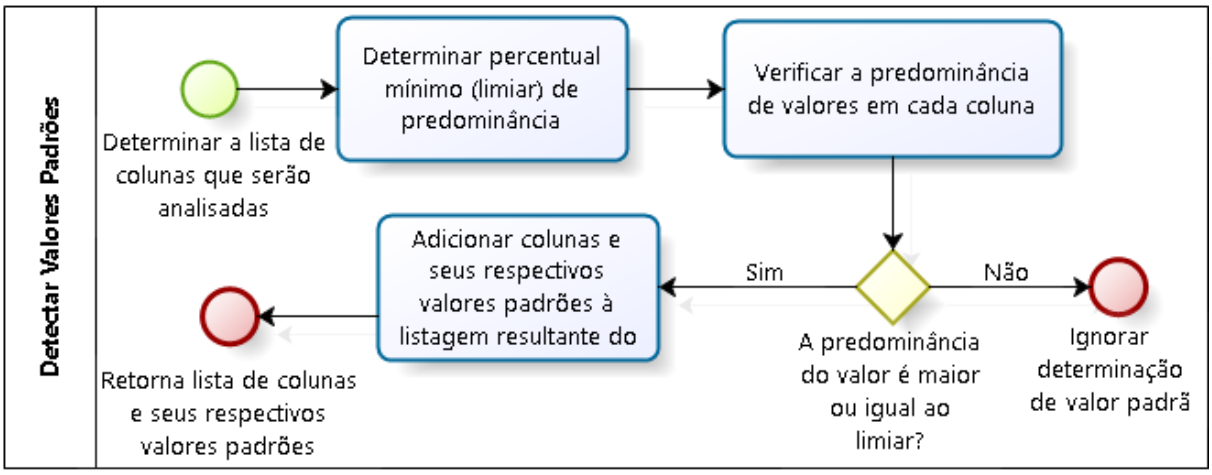

Figura 13 - BPMN - detectar valor padrão

Uma determinação de percentual de predominância mínimo adequado depende da quantidade de grupos de valores que a coluna possui. Por exemplo, se uma coluna *Ci* apresentar apenas os valores *A* e *B*, aquele valor que representa mais de cinquenta por cento do total de registros será o predominante. Da mesma forma, se a coluna possuir os valores *A, B, C e D*, aquele valor que ultrapassar ¼ do total de registros (vinte e cinco por cento) poderá ser predominante. Assim, seria possível adicionar uma lógica ao Algoritmo 2 para calcular, automaticamente, o percentual mínimo de predominância, eliminando a necessidade de tal parâmetro de entrada para este processo. Entretanto, pode-se desejar, por algum motivo, que o percentual mínimo de predominância não seja proporcional ao conjunto de valores que o agrupamento possui.

3.4.3. Algoritmo

A definição de uma lógica que automatize a detecção deste tipo de oportunidade não é complexa, pois consiste em percorrer uma lista de colunas que se deseja fazer tal verificação e aplicar o cálculo definido na regra geral, conforme pode ser observado no Algoritmo 4.

Algoritmo 4: Detectar valor padrão

Entrada: lc {lista de colunas que terão a predominância de valores verificada}
p {limiar de predominância mínima que o valor deve ter em percentual}

Saída: def {lista de colunas passíveis de atribuir valor padrão e seu respectivo valor}

```
1. for each <col> in lc do
2.   value ← GetValueWithMoreOccurences(col)
3.   Z ← GetCountFromTable(col.TableName)
4.   VP ← (value.GetCount()*100)/Z
5.     if (VP >= p) then
6.       def.add(col, value)
7.     end if
8. end for
9. Retorna lista de colunas e seus respectivos valores padrões
```

O método *GetValueWithMoreOccurrences* (linha 2) serve para obter o valor com maior predominância. Para tal, esse método pode executar a chamada de um comando SQL, que agrupa os valores da tabela para a coluna em questão ordenando pela quantidade de ocorrências. Por exemplo, suponha uma tabela *usuarios* com vinte mil registros que possui uma coluna chamada *status*, onde os valores podem variar entre *ativo, cancelado, aguardando ativação, inativo e bloqueado*. É possível efetuar uma consulta SQL similar à seguinte para obter tal informação: ***SELECT*** *status*, ***COUNT(id) AS*** *occurrences* ***FROM*** *usuarios* ***GROUP BY*** *status* ***ORDER BY*** *occurences* ***DESC LIMIT*** *1*.

3.5. Padronizar Formatos

Este é um processo focado, essencialmente, nos dados contidos em colunas e serve, especificamente, para padronizá-los. Por exemplo, uma coluna chamada “fone” destinada a armazenar números de telefones, onde em determinados momentos um mesmo número pode se apresentar nos seguintes formatos: (55) 9120-9789, (55) 9120 9789, 55-9120-9789, etc. Em um cenário com várias aplicações, onde cada aplicação possui seu formato de apresentação, é conveniente que, na base de dados, esse número seja armazenado em um formato neutro para que tais aplicações utilizem a máscara desejada para a exibição em suas interfaces. Neste caso a refatoração de padronizar formato pode ser aplicada para que fique em um formato como “55912097899”.

As soluções para tal imperfeição pertencem à categoria de qualidade de dados e podem contribuir com as seguintes dimensões de qualidade diretamente ou indiretamente: quantidade apropriada de dados, credibilidade, representação concisa, representação consistente, facilidade de manipulação, facilidade de interpretação, facilidade de entendimento e valor adicionado.

3.5.1. Regra Geral

A definição da regra geral para identificar a oportunidade de padronizar formato possui apenas a seguinte condição: se em uma dada coluna *Ci* houver registros que não correspondam ao formato *X*, onde *X* é representado por uma expressão regular que define o formato desejado para os dados armazenados, então, *Ci* pode ser marcada para receber a refatoração de padronizar formato.

3.5.2. Processo de Detecção

Como é possível observar, o fluxo do processo ilustrado na Figura 14 é uma transposição dos principais passos executados pelo Algoritmo 5. Entretanto, é possível executar todo este processo sem, necessariamente, precisar do apoio de uma ferramenta que implemente o algoritmo proposto se o banco de dados tiver funções que deem suporte às validações com expressões regulares. Por exemplo, suponha uma tabela chamada *usuarios_telefones*, que possui a coluna *numero*, destinada a armazenar números de telefones de usuário do sistema com DDD no seguinte formato: (99) 9999-9999. Tal formato pode ser definido pela expressão regular "^\([1-9]{2}\)\ [2-9][0-9]{3}\-[0-9]{4,5}$". Logo, é possível construir a seguinte consulta SQL para capturar aqueles registros fora do padrão desejado: ***SELECT*** * ***FROM*** *usuarios_telefones* ***WHERE*** *id* ***NOT IN (SELECT*** *id* ***FROM*** *usuarios_telefones* ***WHERE*** *numero* ***REGEXP*** *"^\\([1-9]{2}\\)\\ [2-9][0-9]{3}\\-[0-9]{4,5}$")*

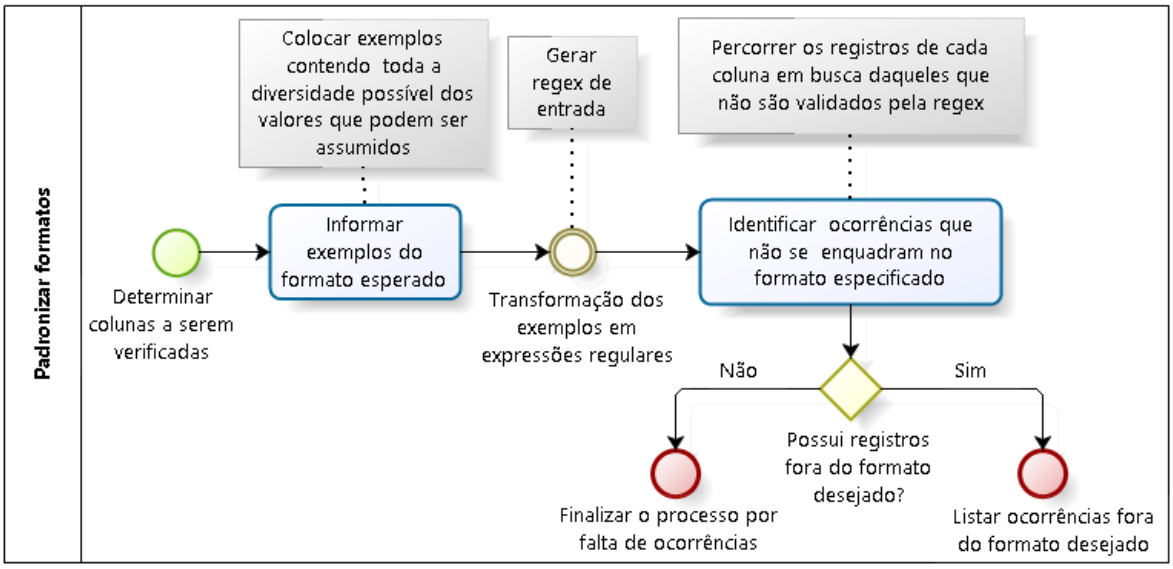

Figura 14 - BPMN - detectar colunas cujos dados podem ser padronizados.

A função *REGEXP* é uma função implementada no banco MySQL, mas funções semelhantes a estas estão presentes em outros tipos de banco de dados. Claramente, a execução manual (sem apoio de ferramentas que implementem o algoritmo) do processo sugerido requer o uso de tais tipos de função e também de conhecimento sobre expressões regulares para a construção das instruções SQLs necessárias. Mesmo para uma execução manual do processo, a atividade de informar/levantar exemplos do formato esperado é importante para a construção de expressões regulares, já que o usuário pode fazê-la diretamente, se tiver conhecimento, para que as reflexões sobre todas as possibilidades de valores sejam avaliadas.

3.5.3. Algoritmo

O Algoritmo 5 realiza as chamadas necessárias para converter os exemplos informados em uma expressão regular que represente o formato desejado. Além disso, percorre os registros das colunas informadas, confrontando com tal expressão regular, para identificar aquelas colunas que precisam ter o seu formato padronizado. Assim, como a definição da regra geral, o algoritmo para a detecção deste tipo de refatoração também é bastante sucinto.

Entretanto, os requisitos devem ser atendidos com cautela para que este processo tenha a precisão desejada para identificar os registros que estejam fora do formato. É essencial que o parâmetro de entrada que contém a lista de exemplos de registros no formato desejado compreenda uma amostragem que reflita a diversidade de valores possíveis que as colunas podem assumir.

Por exemplo, se o alvo destas verificações são colunas destinadas a armazenar números de telefones do Brasil com Discagem Direta à Distância (DDD), é importante que os exemplos informados tenham dez e onze dígitos, se colunas destinadas a armazenar Registro Geral (RG), não se deve considerar exemplos que tenham apenas números, mas, também, deve-se levar em consideração àqueles casos que possuem letras.

Algoritmo 5: Detectar registros fora do formato

Entrada: le {lista de exemplos de registros no formato desejado}
lc {lista de colunas que terão seus dados confrontados com o formato dos exemplos}

Saída: cof {lista de colunas fora do formato e seus respectivos registros}

```
1. X ← ConvertSamplesToRegex(le)
2. for each <col> in lc do
3.    datasOfColumn ← SelectColumnData(col)
4.    regoff.clear()
5.    for each <reg> in datasOfColumn do
6.       if !Regex.IsMatch(reg, X) then
7.          regoff.add(reg)
8.       end if
9.    end for
10.   if (regoff.count > 0) then
11.      cof.add(col, regoff)
12.   end if
13.end for
14. Retorna lista de colunas que devem ser padronizadas e seus respectivos registros fora do
formato desejado
```

O método *ConvertSamplesToRegex* (linha 1) é o responsável pela geração da expressão regular utilizada para identificar os registros que se enquadram no formato desejado. Durante as pesquisas deste trabalho, não foi encontrada nenhuma proposta ou implementação que gerasse uma expressão regular a partir de uma lista de *strings*. Por esse motivo, desenvolvemos uma implementação detalhada no próximo capítulo deste trabalho.

O método *SelectColumnData* (linha 3) serve para selecionar os registros contidos na coluna representada pelo objeto em questão através de comando SQL, gerado a partir da seguinte *string*: "*SELECT* "+col.Name+" *FROM* "+ col.TableName;

A variável *regoff* (linha 7) serve para armazenar, em memória, a lista de registros que não se enquadram no formato desejado. Ela é zerada antes de percorrer os registros da coluna corrente do processo. O método *IsMatch* (linha 6) verifica se o registro em questão é válido para o formato especificado e retorna verdadeiro em caso positivo e falso em caso negativo. O valor que não corresponder ao formato desejado é adicionado à variável *regoff*. Normalmente, a maioria das linguagens de programação oferta nativamente métodos ou funções para a validação de dados através de expressões regulares.

Por fim, se a coluna tiver alguma ocorrência de registro não válido para o formato desejado, configura-se uma oportunidade de refatoração e, portanto, a mesma é adicionada na lista de saída do algoritmo.

3.6. Transformar Colunas Nulas em Não Nulas e Vice-Versa

A transformação de uma coluna nula em não nula, normalmente, parte da necessidade de garantir que a mesma seja sempre preenchida. Entretanto, durante o processo de modelagem da base de dados, pode-se definir a possibilidade de uma coluna ser nula, porém o preenchimento da mesma é garantido por parte das aplicações. Isso pode ser observável mesmo desconhecendo como as aplicações tratam tal coluna, quando apresentar valores em todos os registros na coluna e a tabela, da qual faz parte, possuir um tempo considerável de uso.

Por outro lado, a transformação de uma coluna não nula em nula é motivada para uma melhor representatividade das informações. A necessidade deste tipo de transformação é observável quando um campo textual for vazio em vários registros. Já quando for um valor numérico, é comum o preenchimento com zero apenas para satisfazer a necessidade de informar algo no campo que é requerido nas inserções ou atualizações. Entretanto, o valor zero, em alguns registros, pode ser válido dependendo do contexto. Esse contraponto deve ser severamente relevado na hora de converter os registros zerados em nulos. Como o foco deste trabalho é utilizar apenas as informações residentes na base de dados, a definição da regra geral e do algoritmo será sob as circunstâncias apresentadas.

Esse tipo de refatoração é pertencente à categoria de qualidade de dados e contribui com as seguintes dimensões de qualidade: quantidade apropriada de dados, credibilidade,

representação concisa, representação consistente, facilidade de interpretação, objetividade, facilidade de entendimento e valor adicionado.

3.6.1. Regra Geral

Considerando uma coluna *Ci*, o tipo de tal coluna *T_Ci*, um limiar *X* que define o tempo mínimo de existência da tabela para ser considerada na transformação, a quantidade de registros nulos, vazios ou zerados *Y*. Considerando, também, a data de criação da tabela *CD* em que *Ci* está contida e a data corrente *ND*, a regra para a identificação de oportunidades deste tipo de refatoração, em vista da descrição anterior, pode ser definida sob as seguintes condições:

a) se *Ci* está definida como nula, então *Ci* pode ser marcada para transformação em não nula se satisfazer as seguintes condições:
   — $Y = 0$;
   — $X >= ND - CD$;
b) se *Ci* está definida como não nula, pode ser marcada como nula sob as seguintes condições:
   — $Y > 0$;
   — se *T_Ci* for numérico e os valores zerados são não são válidos, ou seja, servem apenas para cumprir a obrigatoriedade de preenchimento.

Um valor apropriado para *X* é essencial na aplicação dessa regra para desconsiderar aquelas tabelas que foram criadas recentemente e que, consequentemente, apresentam pouco tempo de uso para que sejam colocadas em evidência para este tipo de transformação.

3.6.2. Processo de Detecção

A Figura 15 ilustra o fluxo do processo sugerido para a detecção de colunas definidas como nulas e que podem ser transformadas em não nulas e vice-versa.

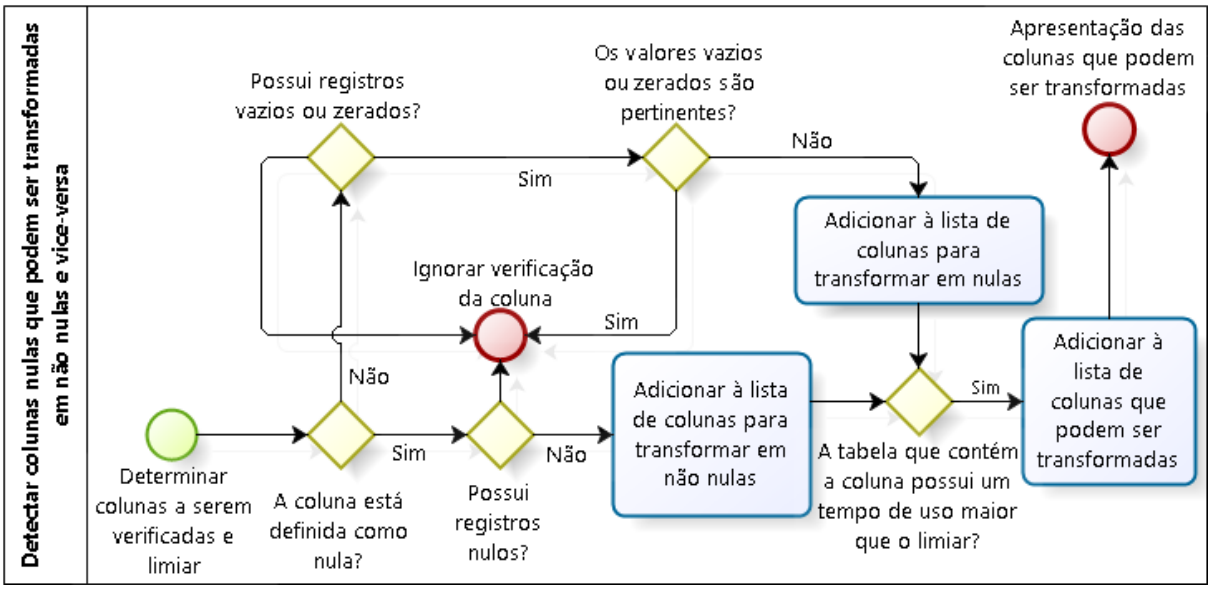

Figura 15 - BPMN - detectar colunas que podem ser transformadas.

A execução deste processo é uma aplicação direta da regra geral apresentada e do algoritmo proposto para a detecção deste tipo de imperfeição, exceto pela atividade "verificar a pertinência dos valores zerados". Essa atividade compreende fazer a verificação sobre a validade/pertinência dos valores zerados informados. Ou seja, verifica se as ocorrências possuem este valor apenas para satisfazer o requerimento de preenchimento para que a possibilidade de transformação de uma coluna não nula em nula seja relevante. Essa não é uma verificação que se consiga fazer através de algoritmos simples, normalmente, este tipo de verificação exige conhecimento sobre os valores que fazem sentido a coluna assumir.

3.6.3. Algoritmo

O algoritmo visa demonstrar uma aplicação direta das regras da seção 3.6.1 para uma implementação simples. Para que tal aplicação seja viável é necessário o desenvolvimento e utilização de alguns métodos que obtenham determinadas informações, como por exemplo, o método *CountNullableTuples* (linha 3), que é responsável por contabilizar a quantidade de registros nulos sob a coluna alvo das verificações. Esse método deve desencadear a chamada a um comando SQL semelhante ao seguinte comando:

***SELECT COUNT(*) FROM** tabela **WHERE** coluna **IS NULL**;*

Onde *coluna* é o nome da coluna alvo da verificação e *tabela* é nome da tabela que tal coluna esteja contida.

Algoritmo 6: Detectar transformação de colunas nulas em não nula e vice-versa

Entrada: lc {lista de colunas que serão verificadas}
X {tempo mínimo de existência da tabela para ser considerada no processo}

Saída: lcTrans {lista de colunas passíveis de transformação}

```
1. for each <col> in lc do
2.   if (col.IsNullable) then
3.     Y ← CountNullableTuples(col)
4.     if (Y = 0) then
5.       tableLifetime ← DateTime.Now() – GetCreationDateOfTable(col.TableName)
6.       if (X <= tableLifetime) then
7.         lcTrans.add(col, “Transformar em não nula”)
8.       end if
9.     end if
10.  else
11.    Y ← CountEmptyOrZero(col)
12.    if (Y > 0) then
13.      lcTrans.add(col, “Transforma coluna nula”)
14.    end if
15.  end if
16. end for
17. Retorna lista de colunas que podem ser alvo de transformação
```

Para saber o tempo de existência da tabela que contém a coluna em questão, representada pela variável *tableLifetime* (linhas 5 e 6), é necessário obter a data corrente através de uma chamada semelhante ao *DateTime.Now*() (linha 5), que corresponde a variável *ND* das regras, menos a data de criação da tabela através do método *GetCreationDateOfTable* (linha 5), equivale a variável *CD* da regra. A implementação do método *GetCreationDateOfTable* depende do tipo de banco de banco de dados, pois alguns armazenam tal informação nativamente. Para aqueles que não têm esse recurso, será necessário providenciar alguma fonte de dados alternativa que disponibilize a relação das tabelas contidas no esquema e sua respectiva data de criação.

O método *CountEmptyOrZero* (linha 11) funciona de forma semelhante ao *CountNullableRegister* (linha 3), porém verifica aqueles registros vazios ou zerados ao invés de nulos. Por exemplo, se campo for do tipo textual irá executar a seguinte consulta SQL:

SELECT COUNT(*) FROM *tabela* ***WHERE*** *coluna = "";*

Se o campo for de um tipo temporal, deve-se fazer a consulta com base no valor mínimo que o campo pode assumir. Por exemplo, se for do tipo *DateTime*, normalmente é “01/01/0001 00:00:00”. Isso porque é uma prática comum alguns desenvolvedores preencherem campos do tipo temporal com o valor de data mínimo apenas para satisfazer o requisito de preenchimento.

3.7. Normalizar a Nomenclatura

É comum que as organizações definam determinados padrões sobre os artefatos, também da mesma forma que muitos tipos de linguagens de programação e *frameworks* definem padrões de nomes para os diferentes elementos do código (métodos, parâmetros de métodos, classes, atributos, etc.). É necessário estabelecer regras que definam como os diferentes tipos de objetos da base de dados devem ser nomeados. Logo, aqueles elementos que possuem nomes que fogem das regras de nomeação definidas podem representar uma oportunidade para ter sua nomenclatura normalizada.

Comumente, as correções necessárias para imperfeições deste tipo resultam em operações pertencentes à categoria estrutural. Também desencadeiam, incondicionalmente, a necessidade de alterações em demais recursos que estejam acoplados (aplicações, outras bases de dados, *frameworks* de persistência, etc.), diferentemente, por exemplo, da transformação de uma coluna não nula para nula que normalmente têm maior impacto no próprio banco de dados. Uma base de dados com a nomenclatura normalizada contribui com os seguintes atributos de qualidade: credibilidade, completude, representação consistente, facilidade de interpretação, reputação, facilidade de entendimento e valor adicionado.

3.7.1. Regra Geral

A especificação de um padrão de nomenclatura para os diferentes tipos de objetos da base de dados pode partir de três passos básicos para que esta heurística atinja os objetivos. Estes passos são os seguintes:

a) especificar como será a separação das palavras que compõem os nomes;

b) especificar se será adotada a singularização ou a pluralização das palavras;
c) especificar prefixos e/ou sufixos para rotular cada tipo de objeto.

Para a especificação da separação de palavras, existem alguns padrões conhecidos como *PascalCase* e *camelCase* exemplificados por MAMONE (2011), onde no *PascalCase* cada palavra que compõe o nome começa em maiúsculo e no *camelCase* também, exceto a primeira. Além disso, é bastante comum a utilização do caractere "_" (*underscore*) como separador.

Quanto à especificação das palavras apresentarem-se no singular ou no plural, ainda não existe uma convenção forte sobre qual das formas se deve trabalhar. Por esse motivo, é necessário que o usuário da heurística determine o modo desejado.

A utilização de prefixos e/ou sufixos para rotular os tipos de objetos também é uma prática comum no mercado para que fique fácil de identificar os tipos de objetos envolvidos a partir de uma leitura direta da instrução SQL. A utilização desse artifício é exemplificada no Quadro 7.

Tipo de objeto	**Prefixo**	**Sufixo**
Tabela	Tbl	
Coluna		
View	Vw	
Trigger	Tg	
Stored Procedure	Sp	
Stored Funcion	Sf	
Eventos	Evt	

Quadro 7 - Exemplo de definição de prefixos e sufixos por tipo de objeto

Desta forma, se for definido que a separação for *PascalCase* e no plural podem ter tabelas nomeadas como: *TblUsuariosTelefones*, *TblClientes*, *TblUsuariosPermissoes*, etc. Views nomeadas como: *VwRelatoriosFinanceiros*, *VwResumosDeAcessos*, etc. Portanto, como regra geral, um elemento da base de dados *E* representa uma oportunidade de refatoração se satisfazer pelo menos uma das condições:

a) *E* não possui separadores conforme a especificação;
b) *E* possui palavras no singular quando a definição for no plural e vice-versa;
c) *E* não apresenta o prefixo e/ou sufixo definido para o seu respectivo tipo de objeto;

No Quadro 7 não foram utilizados sufixos nos exemplos, tampouco foram utilizados sufixos e prefixos no tipo de objeto coluna, por não ser uma prática comum. Entretanto, o

usuário é livre para estabelecer as regras de nomeação para fornecer os parâmetros de entrada do algoritmo e executar o processo como bem entender.

3.7.2. Processo de Detecção

Resumidamente, esse processo consiste em especificar a morfologia dos nomes dos diferentes tipos de objetos da base de dados e em verificar aqueles elementos que não se enquadram em tal especificação para que, posteriormente, sejam renomeados. Em posse da lista de elementos que deve ter a nomenclatura verificada devem-se executar os passos mencionados na regra geral, conforme diagrama ilustrado na Figura 16.

A atividade de identificar as ocorrências que não se enquadram nas especificações torna-se morosa sem o apoio de uma ferramenta que implemente algo semelhante ao Algoritmo 7. Entretanto, as verificações necessárias para identificar tais ocorrências podem ser facilitadas naqueles tipos de bancos de dados que partem de um esquema que armazena informações sobre os diferentes elementos de outros esquemas, normalmente nomeados como *information_schema*. Por exemplo, supondo que as tabelas devem ser prefixadas com *Tb_*; com um comando SQL, semelhante ao abaixo, fica fácil de identificar aquelas que devem ter a nomenclatura normalizada por não terem o prefixo especificado nas regras de nomeação:

SELECT * FROM *information_schema.Tables* ***WHERE*** *name* ***NOT LIKE*** *"Tb_%";*

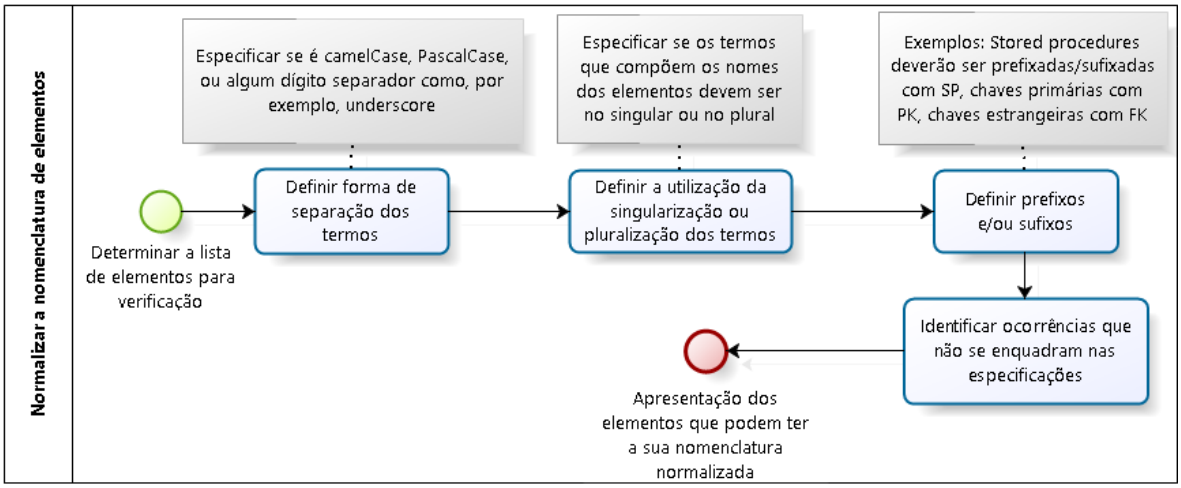

Figura 16 - BPMN - normalizar a nomenclatura de elementos

Em bancos de dados sem meios para extração de informações do esquema, a conferência pode ser feita de maneira visual, o que dependerá de uma maior atenção do usuário que estiver desempenhando esta atividade. Evidentemente, existem algumas estratégias para automatizar a identificação de situações como estas, como por exemplo, organizar a lista dos elementos a serem analisados em listas menores por tipo de objeto, colocar cada lista menor em eletrônica e organizar a lista em ordem alfabética. Deste modo, ficará fácil de identificar, visualmente, aqueles elementos que não possuem o prefixo desejado.

Por fim, uma lista dos elementos a serem renomeados deve ser gerada como artefato final para que as providências de aplicação da refatoração sejam tomadas. Por exemplo, acriação de *scripts* SQLs para as alterações dos nomes errados e criação de outros elementos intermediários no intuito de alcançar a estrutura final desejada até que as aplicações e/ou outras bases de dados que referenciam os elementos listados sejam alteradas.

3.7.3. Algoritmo

Embora as condições para determinar que um elemento necessite ter sua nomenclatura normalizada sejam objetivas, conforme consta na regra geral, existem dependências que precisam ser resolvidas até chegar ao ponto da verificação de tais condições que acabam tornando o Algoritmo 7 um pouco extenso, mas nem por isso complexo. Dentre as dependências estão os parâmetros de entrada, que devem ser definidos com atenção para que eles representem o cenário desejado. No parâmetro de entrada *separationT* deve ser especificado, também, o caractere que servirá para separar palavras para o modo customizado como, por exemplo, "_". Os parâmetros *prefixes* e *suffixes* devem conter os prefixos e sufixos utilizados por tabelas, colunas, *views*, *triggers*, etc., respectivamente conforme foi exemplificado no Quadro 7.

Inicialmente, é necessário separar os termos que compõem o nome de cada elemento de acordo com o tipo de separação de palavras especificado, viabilizando assim a conferência de cada condição da regra. O método "*SeparateWords*" (linha 2) é responsável por executar tal separação, onde a partir dele, é possível obter o prefixo, o sufixo e demais termos que compõem o nome.

Algoritmo 7: Detectar elementos com a nomenclatura inadequada

Entrada: le {lista de elementos que serão verificados}
separationT {tipo de separação das palavras: camelCase, PascalCase ou customizado}
isPlural {define se os nomes dos elementos devem estar no plural}
prefixes {lista de prefixos que devem ser utilizados para cada tipo de objeto}
suffixes {lista de sufixos que devem ser utilizados para cada tipo de objeto }

Saída: leToRename {lista de elementos que devem ser renomeados}

```
1. for each <elem> in le do
2.    words ← SeparateWords(elem.name, separationT)
3.    if prefixes != null and GetPrefixOrSuffix(e.ElementType, prefixes) != prefix then
4.      leToRename.add(elem)
5.    else if suffixes != null and GetPrefixOrSuffix(e.ElementType, sufixes) != sufix then
6.      leToRename.add(elem)
7.     else
8.       for each word in words
9.         if (isPlural and WordIsSingular(word)) or (!isPlural and !WordIsSingular(word))
             and word != prefix and word != suffix then
10.          leToRename.add(elem)
11.        else
12.          case (separationT.TypeOfConvention)
13.            Convention.PascalCase : begin
14.              if !Regex.IsMatch(word, "^([A-Z]+[a-z]+[0-9]+)$") then
15.                leToRename.add(elem)
16.              end if
17.            end
18.            Convention.camelCase : begin
19.              if words[0] = word then
20.                if !Regex.IsMatch(word, "^[a-z]+[0-9]$") then
21.                  leToRename.add(elem)
22.                end if
23.              else
24.                if !Regex.IsMatch(word, "^([A-Z][a-z]+[0-9]+)$") then
25.                  leToRename.add(elem)
26.                end if
27.              end if
28.            end
29.            Convention.Custom : begin
30.              if !Regex.IsMatch(word, "^[a-z]+[0-9]+$") then
31.                leToRename.add(elem)
32.              end if
33.            end
34.        if leToRename.Contains(elem) then
35.          break
36.        end if
37.      end for
38.    end if
39. end for
40. Retorna lista de elementos que devem ser renomeados
```

O método “GetPrefixOrSuffix” (linhas 3 e 5) é encarregado de obter o prefixo ou sufixo para o tipo de objeto alvo da verificação a partir dos parâmetros de entrada. Como pode ser observado, isso é necessário para conferir se o elemento em questão possui o prefixo ou sufixo esperado. Caso não possua, isso representará uma oportunidade de refatoração para que o elemento tenha sua nomenclatura normalizada.

Caso o nome do elemento não apresente problemas de prefixos ou sufixos, os demais termos são verificados se estão no plural de acordo com o parâmetro “isPlural” (linha 9). O método “WordIsSingular” (linha 9) é responsável por verificar se o termo está no singular. A implementação deste método pode ser feita através do uso de dicionários no formato de API (Interface de programação de aplicações) que retornam o significado para palavras no singular em conjunto com verificações em torno das terminações das palavras.

Por exemplo, se este método receber como parâmetro de entrada a palavra “casa” e a requisição a API retornar um significado, logo esta palavra está no singular. Se receber como parâmetro de entrada a palavra “casas”, retira-se o “s” do final e a consulta na API é feita para se certificar que a palavra é válida e está no plural (retorno falso). Evidentemente, tal método também exige o tratamento de palavras terminadas em “ães”, ”ões” e “éis”. Exemplos de API com este propósito são Thesauros, desenvolvido por ALTERVISTA (2015) e o Dicionário-aberto desenvolvido por GOMES (2015).

Por último, uma conferência sobre a correta separação dos termos é feita, onde para cada caso de separação é validado através de sua respectiva expressão regular. Por exemplo, se o tipo de separação dos termos for customizado (“custom”), cada termo que compõe o nome poderá ter apenas letras minúsculas ou números. Não pode haver no termo nenhum caractere maiúsculo para não misturar padrões de nomenclatura ao estilo *PascalCase* ou *CamelCase.* Também não pode haver outro caractere separador como “-”, “.”, etc.

O algoritmo proposto não é perfeito, pois podem existir elementos nomeados erroneamente e que não seja possível identificar. Por exemplo, suponha que o tipo de separação de palavras escolhido seja *PascalCase* e tenha uma tabela nomeada como *Usuariostelefones*, quando o correto seria *UsuariosTelefones*. Esse tipo de situação é muito difícil de ser resolvido através de uma implementação. Primeiramente, esta situação não seria possível detectar através do uso de expressões regulares. Depois, para identificar tais casos, seria necessária uma lógica/algoritmo que criasse todas as possibilidades de palavras que poderiam ser formadas a partir do termo através de *substrings* e que consulte a real existência de cada termo através de alguma API de dicionário citada, o que, computacionalmente, pode ser extremamente custoso quando for necessário avaliar grandes quantidades de elementos.

3.8. Considerações Finais

Neste capítulo, o entendimento sobre as regras gerais, algoritmos e processos propostos são de suma importância, pois são as principais contribuições deste trabalho. Para cada tipo de imperfeição de base de dados que este trabalho se propõe a resolver, foi feita uma breve contextualização para informar a quais categorias de refatoração as operações resultantes pertencem e para citar quais as dimensões de qualidade são atingidas quando solucionadas tais imperfeições.

Em cada regra geral, foram demonstradas as principais condições para identificar cada tipo de imperfeição utilizada como base para a produção dos algoritmos e processos propostos. Cada algoritmo exposto tem por objetivo propor uma lógica que seja capaz de identificar oportunidades de refatoração para um dado tipo de imperfeição a partir de alguns parâmetros de entrada. Já, nos processos, são explicados e diagramados os passos necessários para identificar as oportunidades de refatoração a partir de uma imperfeição sem, necessariamente, depender da implementação do respectivo algoritmo.

Por fim, tais algoritmos servem como base para o próximo capítulo, onde são abordadas algumas implementações que foram feitas para certificar a viabilidade de cada proposta, juntamente com a sugestão de uma arquitetura de software modular para a produção de uma ferramenta que se responsabilize pela implementação das diferentes heurísticas aqui propostas.

4. IMPLEMENTAÇÃO

Os capítulos anteriores abordaram as definições das heurísticas que visam auxiliar na detecção de oportunidades de refatoração e, ao mesmo tempo, fornecem alguns algoritmos que podem ser implementados para auxiliar na automatização dos processos envolvidos. Com base nas propostas anteriores, este capítulo aborda a arquitetura sugerida para a construção de uma ferramenta que execute tais heurísticas. Tal ferramenta foi denominada de "*DatabaseSmellDetector*", já que tem por objetivo detectar oportunidades de refatoração a partir de imperfeições em bases de dados (*Database Smells*).

Tal ferramenta foi desenvolvida em C# utilizando o framework .NET 4.5 da Microsoft e, inicialmente, voltada para MySQL. Embora ela possua a implementação de todos os algoritmos sugeridos e dos principais métodos que viabilizam as heurísticas propostas, é um protótipo.

Além das explicações sobre a arquitetura sugerida para a construção de tal ferramenta, as subseções deste capítulo abordam implementações dos algoritmos presentes no capítulo anterior.

4.1. A Arquitetura do *DatabaseSmellDetector*

A Figura 18 apresenta a arquitetura global através de um diagrama UML de componentes que é detalhado para fornecer uma visão geral sobre toda a solução proposta. Este diagrama UML e os demais expostos neste capítulo foram gerados a partir de uma engenharia reversa sobre a solução desenvolvida no IDE Visual Studio 2013 demonstrada na Figura 17.

Embora a ferramenta abordada tenha sido desenvolvida utilizando as tecnologias mencionadas, a arquitetura proposta é passível de implementação em outras linguagens de programação como, por exemplo, Java. Cada componente diagramado na Figura 18 corresponde a uma camada da aplicação que, em .Net C#, é representada, normalmente, no formato de projeto do tipo biblioteca de classes (*class library)* e que, da mesma forma, poderiam corresponder à pacotes (*packages*) se fossem desenvolvidos em Java. Assim como

em qualquer arquitetura modular, cada componente possui uma determinada responsabilidade para que as classes fiquem organizadas apropriadamente.

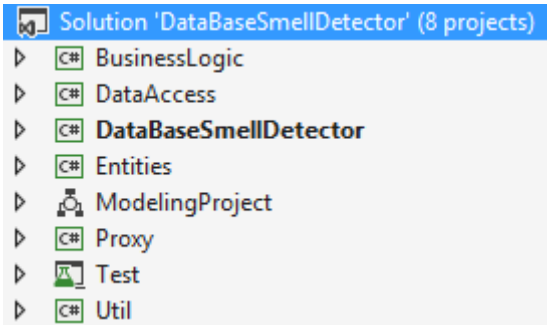

Figura 17 - Conjunto de projetos da solução

Os componentes que compõem a arquitetura proposta podem ser divididos da seguinte forma:

a) *Entities*: formado por classes que representam as diferentes entidades utilizadas na aplicação, normalmente no formato de *data transfer objects* (DTO) ou *structs*. Aqui, também são colocados os arquivos de mapeamentos ORM, os enumeradores e as interfaces que devem ser implementadas no sistema;

b) *DataAccess*: possui as classes responsáveis pela comunicação direta com as bases de dados;

c) *BusinessLogic*: responsável pela lógica de negócio. Aqui, ficam as classes responsáveis pela implementação dos algoritmos propostos.

d) *Util*: possui classes com métodos utilitários que podem ser utilizados nas diferentes camadas da aplicação, como por exemplo, manipulação de datas, manipulação de expressões regulares, cálculos e validações comuns, etc...

e) *Proxy*: possui as classes responsáveis pela comunicação direta com *webservices* e API de terceiros que a aplicação possa necessitar como, por exemplo, dicionários. Os mapeamentos de *webservices* que tenham *webservice description language* (WSDL) também devem ser realizados nesta camada.

f) *Test*: aqui, devem ser centralizados os métodos de testes unitários das classes de outros componentes.

g) *DataBaseSmellDetector*: essa é a camada de apresentação da ferramenta, onde ficam as interfaces com os formulários para a interação com a lógica de negócio. Neste trabalho, foram utilizados *windows forms* (interfaces *desktops*) na construção das telas

para a manipulação das heurísticas.

No diagrama ilustrado na Figura 18, é possível observar as dependências que cada componente possui para entender como se relacionam entre si. Neste trabalho, foram produzidos métodos de testes unitários para a camada de acesso a dados (*DataAccess*), lógica de negócio (*BusinessLogic*) e utilitários (*Util*). Logo, o componente de testes (*Test*) possui dependência apenas destes 3 componentes.

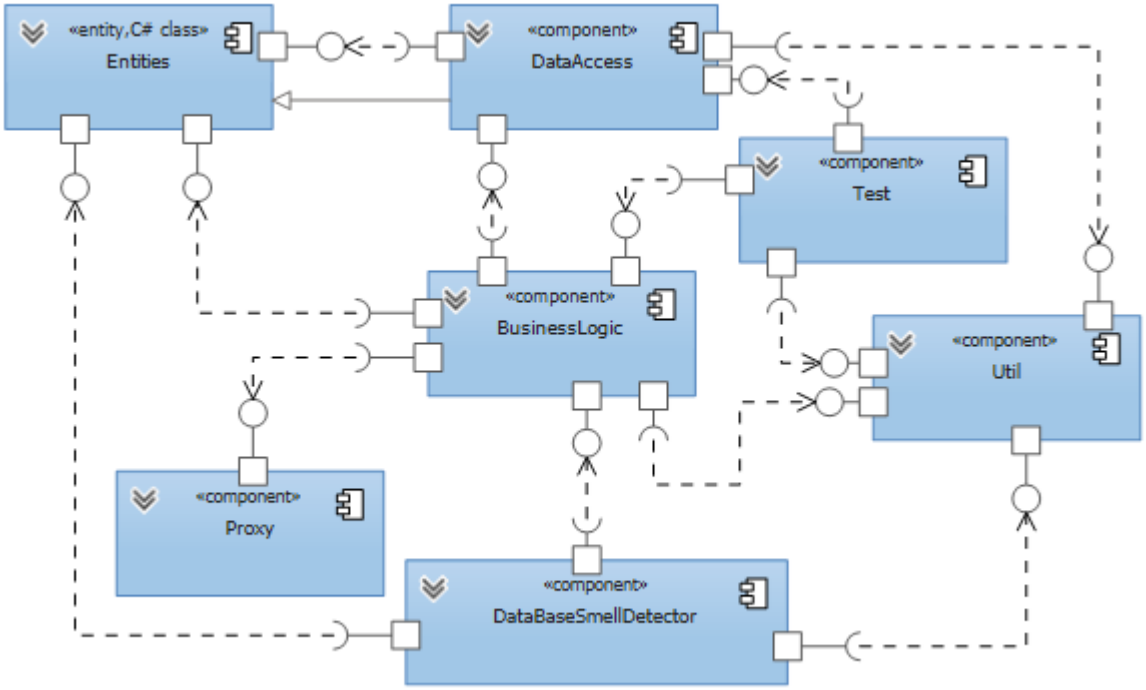

Figura 18 - UML - diagrama de componentes

O componente de entidades (*Entities*) não depende de nenhum outro e a grande maioria depende dele por, justamente, prover os tipos de classes que transitarão entre os diferentes componentes. O componente *Util* também não possui dependências e é referenciado pelos demais componentes com exceção do *Proxy* e *Entities* por ter métodos personalizados para operações, comumente, utilizadas e requisitadas apenas pelos demais componentes. Já o componente *BusinessLogic* é o que possui maior quantidade de dependências por ser a camada centralizadora da arquitetura. Ele é diretamente requisitado pela camada de apresentação *DatabaseSmelldetector* para processar as entradas fornecidas pelo usuário das heurísticas. A próxima subseção abordará o diagrama de classes dos componentes *Entities*, *DataAccess* e *BusinessLogic*, por possuírem as principais classes das implementações realizadas neste trabalho.

4.1.1. Classes dos Principais Componentes

As entidades que compõem este projeto foram baseadas na ontologia sugerida por FOGLIATO (2014). A maioria é destinada a representar elementos da base de dados e a classe utilizada como base é a *DatabaseElementDto*, pois, normalmente, todo elemento da base de dados tem a possibilidade de ter um comentário, seu tipo, nome e a especificação do esquema que o mesmo pertence. Essas relações e outras podem ser observadas na Figura 19, que apresenta o diagrama de classes do componente *Entities*, com exceção das *interfaces* que devem ser implementadas.

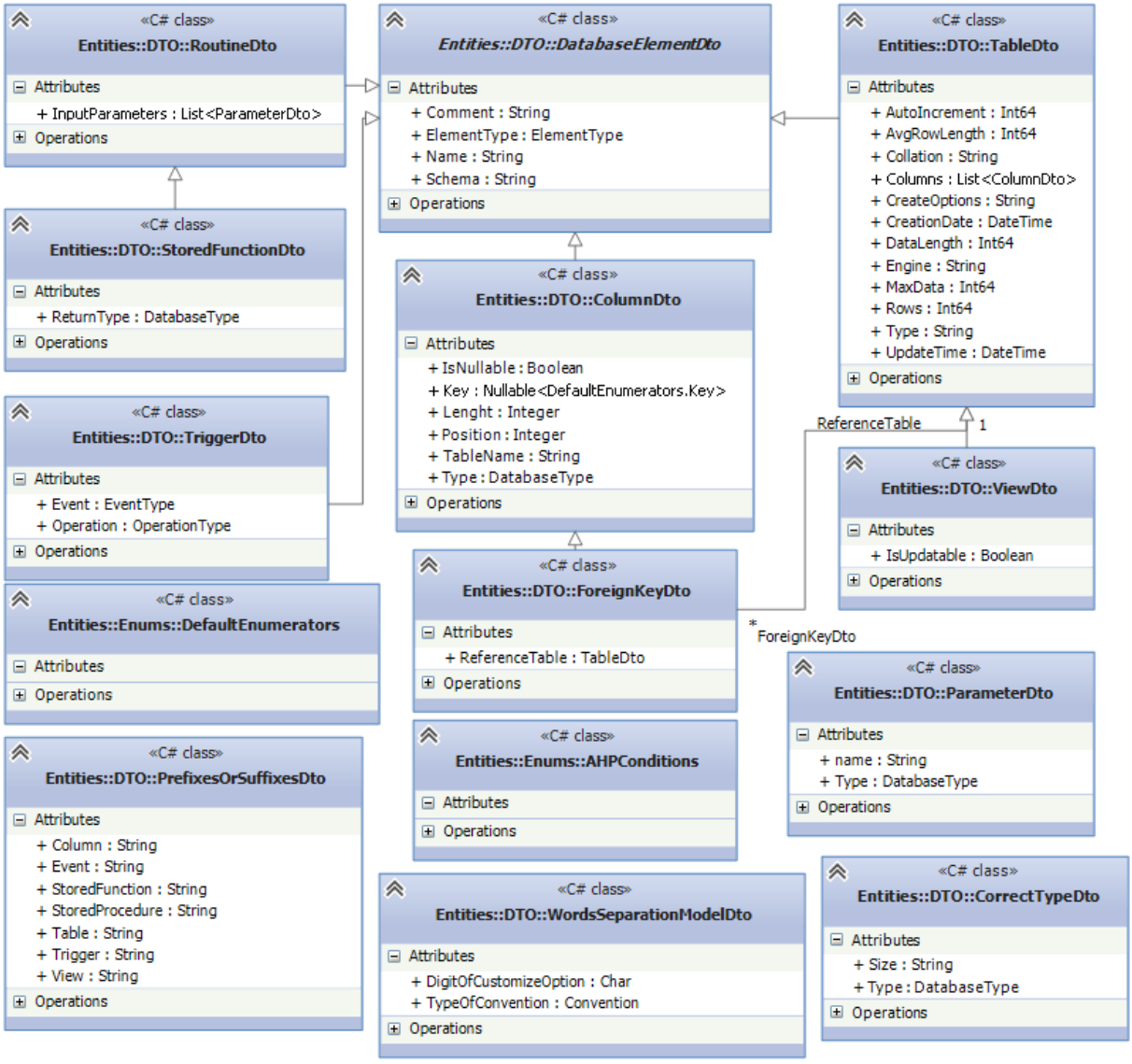

Figura 19 - UML - diagrama de classes - entidades

A classe *DefaultEnumerators* possui os principais enumeradores utilizados na lógica de negócio. As demais classes também são destinadas a complementar tipos de objetos requeridos por tal componente. A Figura 20 apresenta o diagrama de classes da camada de acesso a dados que foi desenvolvida, utilizando o padrão de projeto F*actory Method* (GAMMA et. al., 1999), onde a interface declarada no componente de entidades é implementada na camada de acesso a dados. Dessa forma, é possível fazer a implementação dos métodos necessários para interação com a base de dados para cada tipo de banco de dados.

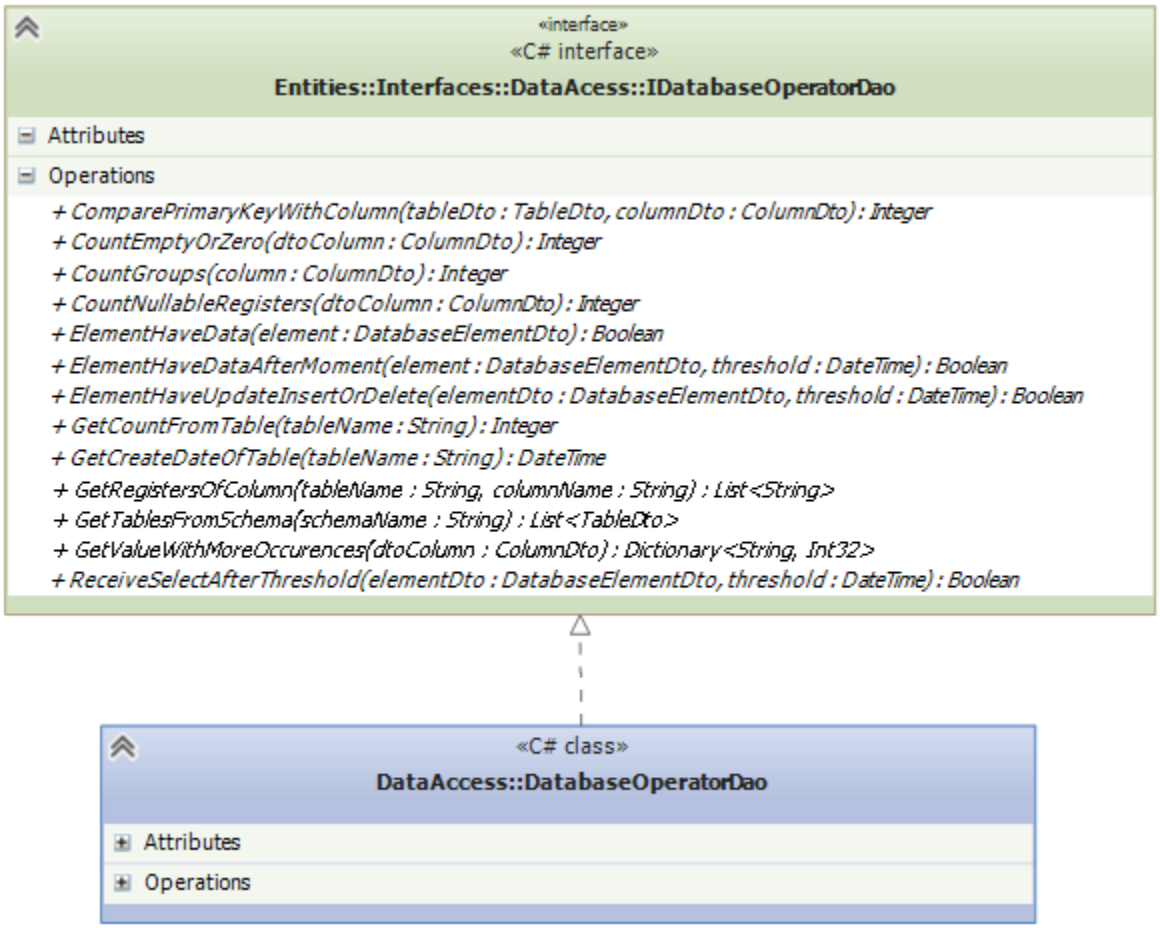

Figura 20 - UML - diagrama de classes - acesso à dados

Neste trabalho, foram implementados métodos para o banco de dados MySQL, mas poderiam ser usados outros SGBDRS, com por exemplo, SQL Server, Oracle, PostgreSQL, etc. Como este projeto foi desenvolvido em .Net C#, quando implementar os métodos para outros tipos de bancos de dados, basta substituir a *dynamic-link library* (DLL) do componente *DataAcess* pela versão que contém o tipo de banco de dados desejado para verificação de

oportunidades de refatoração. Essa camada da aplicação possui apenas a classe *DatabaseOperatorDao*, pois a implementação dos métodos declarados na interface mencionada é suficiente para atender a demanda de acesso à base de dados requisitada pela *BusinessLogic*.

A Figura 21 e a Figura 22 ilustram o diagrama de classes do componente *BusinessLogic*. Nele é possível observar que na implementação feita durante a execução deste trabalho é declarada uma instância da *DatabaseOperatorDao* como atributo de cada classe da lógica de negócio, na classe *InadequateTypesDetector* isto pode ser melhor observado. Neste diagrama também se observa que existe uma classe para cada tipo de imperfeição, cuja detecção este trabalho se propõe a auxiliar. Isto está disposto dessa forma porque cada classe de detecção de imperfeição, normalmente, necessita de outros métodos para que a implementação não fique procedural e, também, para que tenha uma representação que facilite a localização da implementação de cada heurística proposta. Desse modo, as classes que compõem este componente podem ser descritas da seguinte forma:

a) *DeprecatedDetector*: responsável por detectar elementos em desuso. O método *DetectDeprecatedElements* dessa classe é o que implementa o Algoritmo 1;
b) *InadequateTypeDetector*: responsável pela identificação de colunas com tipos ou tamanhos inadequados. Possui o método "*DetectInadequateType*" que implementa o Algoritmo 3;
c) *NormalizeNamesDetector*: implementa o algoritmo 7 no método *FindElementsNormalize* para encontrar elementos que estejam com a nomenclatura fora do padrão desejado;
d) *StandardizeFormatDetector*: implementa o Algoritmo 5 através do método *DetectColumnsWithValuesOutOfFormat* para detectar colunas com valores fora de um formato especificado;
e) *NullColumnToNotNulAndViceVersadetector*: implementa o algoritmo 6 através do método *DetectNullColumnToNotNullAndViceVersa* para detectar colunas nulas que podem ser transformadas em não nulas e vice versa.
f) *DefaultValuesDetector*: implementa o Algoritmo 4 no método *DetectDefaultValues* para identificar a atribuição de possíveis valores padrões em colunas.
g) *ReferenceTableDetector*: implementa o Algoritmo 2 através do método *DetectReferenceTables* para detectar tabelas de referência faltantes ou reaproveitar alguma tabela existente.

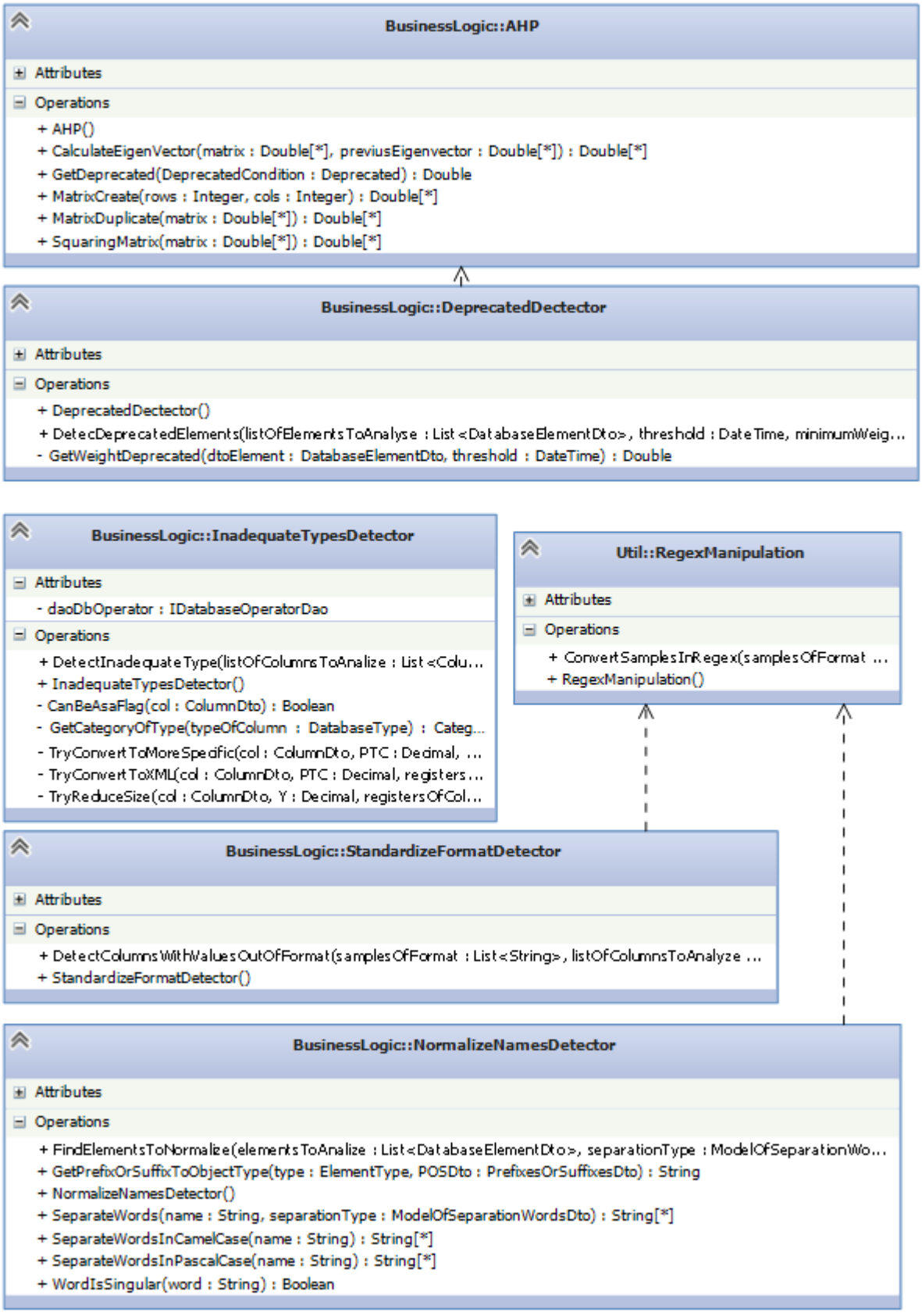

Figura 21 - UML - diagrama de classes - lógica de negócio – parte 1

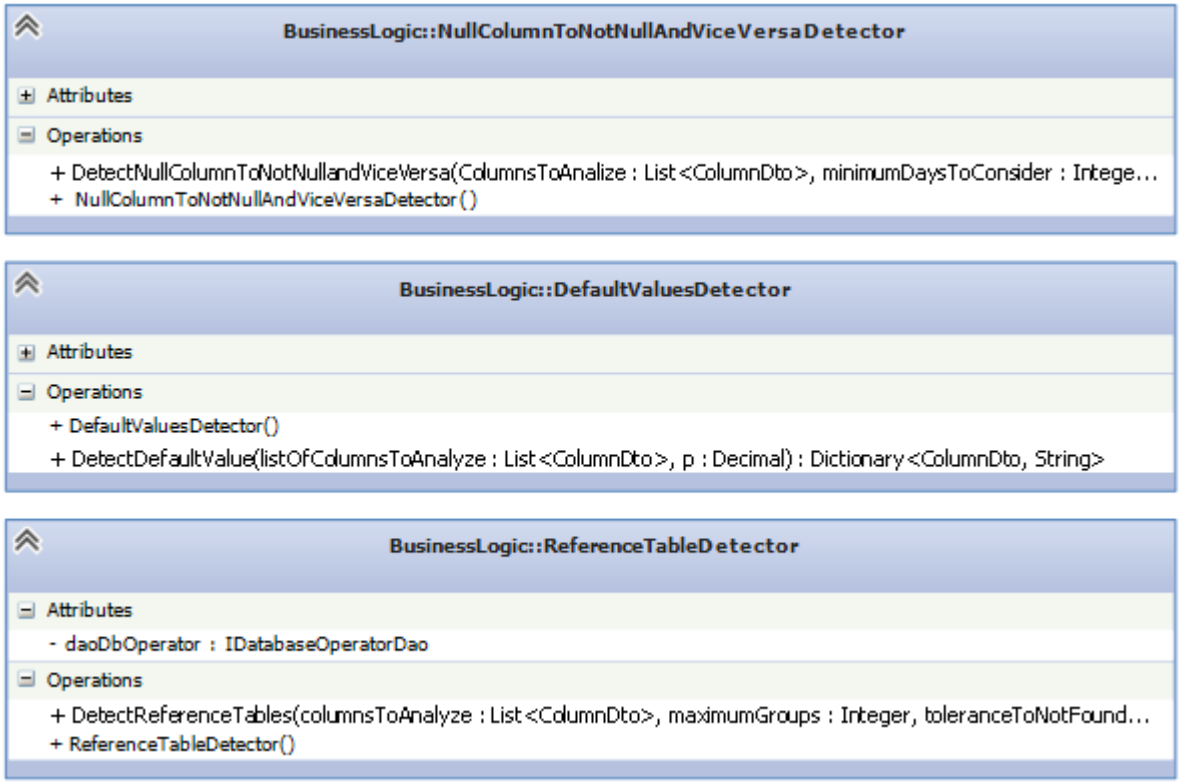

Figura 22 - UML - diagrama de classes - lógica de negócio – parte 2

No diagrama de classes da Figura 21 e 22 também é importante observar a parametrização dos métodos que implementam os algoritmos, pois possuem os mesmos parâmetros de entrada especificados no capítulo anterior. Porém, devem ser previstos os tipos necessários para que sejam implementados em C# de maneira clara. Por fim, vale salientar que, para determinar melhor as responsabilidades de cada classe na implementação feita, a utilização do padrão de projeto *Singleton* (GAMMA et. al., 1999), que também poderá ser utilizado para evitar declarações desnecessárias da classe *DatabaseOperatorDao* nesta camada de aplicação.

4.2. Alguns Métodos Necessários para Viabilizar as Heurísticas Propostas

Para viabilizar os algoritmos propostos neste trabalho foi necessária a implementação de alguns métodos que não foram encontrados durante as pesquisas. O método para calcular *eigenvector* e a conversão de uma lista de *string* que pertençam a um determinado formato em expressão regular são exemplos disto. Obviamente, existem outros métodos neste trabalho

que se enquadram nesta situação, porém nesta seção serão abordados apenas estes, pois são aqueles que apresentaram maior dificuldade para o desenvolvimento e são fundamentais para identificar elementos obsoletos e padronizar formatos respectivamente.

Durante as pesquisas deste trabalho, ferramentas que trabalham com a execução do método AHP não foram difíceis de encontrar, até porque é um método bastante difundido. Foi possível, também, encontrar fontes de algumas implementações como, por exemplo, a realizada por SAATY (2007). Entretanto, não foi possível encontrar algo que implementasse o cálculo do *eigenvector* de forma explícita e que realmente funcionasse. Por esse motivo e, principalmente, para suprir a necessidade da implementação do método "DetectDeprecatedElements", o método chamado "CalculateEigenvector", demonstrado na Figura 23, foi desenvolvido.

```
01. public double[] CalculateEigenVector(double[,] matrix, double[] previusEigenvector = null)
02. {
03.     double[] eigenvector = new double[matrix.GetLength(0)];
04.     double[,] squaredMatrix = SquaringMatrix(matrix);
05.     int precision = Convert.ToInt32(ConfigurationManager.AppSettings.Get("AplicationServerAddress"));
06.     bool again = false;
07.     for (int i = 0; i < squaredMatrix.GetLength(0); i++)
08.     {
09.         for (int j = 0; j < squaredMatrix.GetLength(1); j++)
10.         {
11.             eigenvector[i] += squaredMatrix[i, j];
12.         }
13.     }
14.     double total = eigenvector.Sum();
15.     if (previusEigenvector == null)
16.         again = true;
17.     for (int i = 0; i < eigenvector.GetLength(0); i++)
18.     {
19.         eigenvector[i] = Math.Round(eigenvector[i] / total, precision);
20.         if (again == false)
21.             if (eigenvector[i] != previusEigenvector[i])
22.                 again = true;
23.     }
24.     if (again)
25.         eigenvector = CalculateEigenVector(squaredMatrix, eigenvector);
26.     else
27.         Console.WriteLine("Eigenvector found");
28.     return eigenvector;
29. }
```

Figura 23 - Método para calcular *eigenvector*

O código da Figura 23 é uma transposição direta dos passos 3 e 4 explicados na seção 2.3. Inicialmente, o método recebe como parâmetro apenas a matriz de critérios, calcula a matriz quadrada da mesma (linha 4), soma as linhas da matriz resultante (linha 11), normaliza o vetor resultante da soma das linhas (linha 19) e confere se o vetor atual é diferente da iteração anterior (linha 21) para que o processo seja repetido através de uma chamada

recursiva (linha 25). No método em questão, foi utilizada uma precisão de quatro dígitos e considera-se encontrado o *eigenvector*, quando a iteração anterior for igual à interação corrente. Para aumentar ou diminuir a precisão de dígitos, basta alterar o parâmetro de configuração da aplicação que é carregado na linha 5.

A Figura 24 exemplifica a chamada do método *CalculateEigenvector* para obter os pesos para cada condição de obsolescência a partir da mesma matriz apresentada no Quadro 6.

```
01. public double GetDeprecated(Entities.Enums.AHPContions.Deprecated DeprecatedCondition)
02. {
03.     /* MATRIX OF AHP METHOD TO MODERATE DEPRECATED OPTIONS*/
04.     double[,] m = MatrixCreate(5, 5);
06.     m[0, 0] = 1; m[0, 1] = 0.33; m[0, 2] = 2; m[0, 3] = 0.5; m[0, 4] = 0.33; //first line
07.     m[1, 0] = 3; m[1, 1] = 1; m[1, 2] = 5; m[1, 3] = 2; m[1, 4] = 0.33; //second line
08.     m[2, 0] = 0.5; m[2, 1] = 0.2; m[2, 2] = 1; m[2, 3] = 0.33; m[2, 4] = 0.2;// third line
09.     m[3, 0] = 2; m[3, 1] = 0.5; m[3, 2] = 3; m[3, 3] = 1; m[3, 4] = 0.5;//fourth line
10.     m[4, 0] = 5; m[4, 1] = 3; m[4, 2] = 5; m[4, 3] = 2; m[4, 4] = 1;//fifth line
11.     return CalculateEigenVector(m)[(int)DeprecatedCondition];
12. }
```

Figura 24 - Método que obtém o peso da condição que considera um elemento obsoleto

O método *GetDeprecated* recebe, como parâmetro, um enumerador contendo a condição que se deseja obter o respectivo peso, monta a matriz de critérios da linha 04 até a linha 10 e, após, consome o método que calcula o *eigenvector*, retornando o peso para a respectiva condição recebida (linha 11).

Durante as pesquisas deste trabalho, também foram levantadas diversas ferramentas e algoritmos de manipulação de expressões regulares, mas não foi encontrado algo que fosse capaz de gerar expressões regulares a partir de uma lista de dados. Isso é essencial para que o método *DetectColumnWithValuesOutOfFormat* identifique aquelas ocorrências que não estão em um formato adequado, de maneira que o usuário da heurística não precise ter conhecimentos sobre expressões regulares. Com um recurso que cumpre esta tarefa, basta informar exemplos dos formatos desejados para que os indesejados sejam identificados.

Basicamente, o método da Figura 25 identifica o tipo de dígito para cada posição dos diferentes exemplos fornecidos e traduz tal dígito para a representação na expressão regular correspondente.

```
01. public string ConvertSamplesInRegex(List<string> samples)
02. {
03.     string regex = string.Empty; string digits;
04.     string separators = "-!\"#$%&'()*+,./:;<=>?@[\\]_`{|}~";
05.     string numeric = "0123456789";
06.     string letter = "qwertyuiopasdfghjklçzxcvbnm";
07.     string letterUC = "QWERTYUIOPASDFGHJKLÇZXCVBNM";
08.     int formatSize = samples.Max(x => x.Length);
09.     bool isNum, isLetter, isLetterUC, isSeparator, isOther, isSpace, isSameChar;
10.     for (int i = 0; i < formatSize; i++)
11.     {
12.             isNum = isLetter = isLetterUC = isSeparator = isOther = isSpace = false;
13.             digits = string.Empty; isSameChar = true;
14.             foreach (string sample in samples)
15.             {
16.                     if (sample[i].Equals(' '))
17.                             isSpace = true;
18.                     else if (separators.Contains(sample[i]))
19.                             isSeparator = true;
20.                     else if (numeric.Contains(sample[i]))
21.                             isNum = true;
22.                     else if (letter.Contains(sample[i]))
23.                             isLetter = true;
24.                     else if (letterUC.Contains(sample[i]))
25.                             isLetterUC = true;
26.                     else
27.                             isOther = true;
28.                     digits += sample[i];
29.             }
30.             if (isSeparator)
31.             {
32.                     if (isNum | isLetterUC | isLetter)
33.                             regex += "(.*)";
34.                     else
35.                     {
36.                             for (int k = 1; k < samples.Count; k++)
37.                             {
38.                                     if (!digits[0].Equals(digits[k]))
39.                                             isSameChar = false;
40.                             }
41.                             regex += (isSameChar ? "\\" + digits[0] : regex += "\\W");
42.                     }
43.             }
44.             else if (isLetter & isLetterUC & isNum)
45.                     regex += "\\w";
46.             else if (isLetter & isLetterUC)
47.                     regex += "([A-Z]|[a-z])";
48.             else if (isLetter)
49.                     regex += "[a-z]";
50.             else if (isLetterUC)
51.                     regex += "[A-Z]";
52.             else if (isNum)
53.                     regex += "\\d";
54.             else if (isSpace)
55.                     regex += "\\s";
56.             else
57.                     regex += "(.*)";
58.           if (digits.Length < samples.Count)
59.              {
60.                     if ((samples.Count - digits.Length == 1) | i == formatSize - 1)
61.                        regex += "?";
62.                     else
63.                        regex += "+?";
64.              }
65.     }
66.     return "^" + regex + "$";
67. }
```

Figura 25 - Método para gerar expressões regulares.

Os tipos de dígitos são controlados através das seguintes variáveis:

a) *isNum*: quando é um número. Os números considerados nessa implementação são aqueles contidos na variável *numeric*;
b) *isLetter*: quando é uma letra minúscula. As letras minúsculas consideradas na implementação, estão contidas na variável *letter*;
c) *isLetterUC*: quando é uma letra maiúscula. As letras maiúsculas consideradas na implementação, estão contidas na variável *letterUC*;
d) *isSeparator*: quando é um dígito separador. Os dígitos separadores considerados na implementação estão contidos na variável *separators*;
e) *isSpace*: quando é um espaço;
f) *isSameChar*: quando é sempre o mesmo caractere que ocupa a posição;
g) *isOther*: quando não se enquadra em nenhum caso anterior.

No bloco de repetição da linha 14 até a linha 29, é analisado o tipo de dígito para a posição do formato de acordo com as possibilidades especificadas nas variáveis *numeric*, *letter*, *letterUc* e *separators*. No laço de repetição da linha 36 até a linha 40 é analisado se é sempre o mesmo caractere que ocupa a posição.

As próximas linhas servem basicamente para fazer a transformação para a expressão regular correspondente ao dígito. Por fim, na linha 69 é retornada a expressão regular gerada. Obviamente, a expressão regular gerada não está na forma reduzida e o método proposto pode ser melhorado neste sentido. Isso não foi feito neste trabalho porque o objetivo principal era suprir a necessidade de um recurso que atingisse o objetivo de gerar expressões regulares a partir de uma lista de exemplos fornecidos. A Figura 26 contém um método pertencente ao componente *Test* que exemplifica o uso do método *ConvertSamplesToRegex*. Além da constatação da correta geração das expressões regulares geradas por *ConvertSamplesToRegex* através deste método de teste, também pode ser utilizadas outras ferramentas, como por exemplo, a desenvolvida por SKINNER (2008).

```
01. [TestMethod()]
02. public void ConvertSamplesInRegexTest()
03. {
04.     List<string> fones = new List<string> { "(55) 3221-9198", "(11) 9120-9789", "(54) 8784-3305",
        "(11) 9844-55215" };
05.     string regexFone = new Util.RegexManipulation().ConvertSamplesInRegex(fones);
06.     bool positive = Regex.IsMatch("(55) 3307-2712", regexFone);
07.     bool negative = Regex.IsMatch("55 5412-ab28", regexFone);
09.     Assert.AreEqual(true, positive,"Correct format.");
10.     Assert.AreEqual(false, negative, "Wrong format.");
10. }
```

Figura 26 - Método de teste para conversão de exemplos em expressões regulares (regex)

Basicamente, é fornecida uma lista de exemplos de números de telefones sob um determinado formato a partir da linha 4, após é gerada a expressão regular na linha 05. Por fim, na linha 6 é testado um valor no formato correto e na linha 07 um valor no formato errado.

4.3. Considerações Finais

A divisão dos componentes sugerida neste trabalho viabiliza uma implementação bem modularizada, onde cada componente possui sua respectiva responsabilidade perante a ferramenta. Isso facilita o entendimento para melhorias e novos desenvolvimentos em cada parte da arquitetura proposta. Os padrões de projetos empregados, além de facilitarem a manutenção da ferramenta desenvolvida, proporcionam uma flexibilização para que seja possível realizar os mesmos processos em cada tipo de banco de dados, onde é necessário apenas implementar a classe de acesso à dados correspondente.

Para a criação de outras heurísticas que não foram contempladas neste trabalho, basta adicionar classes que encapsulem a lógica de negócio de tais heurísticas no componente *BusinessLogic*. Caso estas heurísticas precisem coletar informações do banco de dados por métodos que não estejam contemplados na camada de acesso a dados, basta adicionar a assinatura destes novos métodos na interface *IDatabaseOperatorDao* e produzir sua respectiva implementação. Da mesma forma, se tais heurísticas precisarem utilizar APIs de terceiros, basta adicionar mapeamentos ou até mesmo as implementações necessárias para realizar a integração com tais APIs.

O componente *Test* serve de apoio para criação de testes unitários das classes dos diferentes componentes apresentados, onde também basta adicionar as classes e respectivos métodos daquilo que se deseja testar. Desse modo, é possível concluir que a arquitetura proposta possui escalabilidade para a expansão do projeto que este trabalho está tratando.

Alguns métodos desenvolvidos neste trabalho podem ser utilizados pela comunidade de desenvolvedores para outros fins como, por exemplo, o método que cria expressões regulares *ConvertSamplesToRegex*, que pode ser empregado em qualquer outra solução que precise gerar expressões regulares a partir de exemplos. Assim, consequentemente, deve abstrair a necessidade de conhecimento do usuário sobre tal assunto.

Por fim, este capítulo reforça a viabilização das propostas do capítulo anterior através da demonstração prática da implementação de um protótipo e também serve para auxiliar na compreensão dos resultados de alguns experimentos que são detalhados no próximo capítulo.

5. EXPERIMENTOS E AVALIAÇÃO DOS RESULTADOS

Os experimentos foram realizados em uma base de dados MySQL do sistema de *help desk,* desenvolvido pela ORACLE (2009), chamado Eventum, que foi utilizado por uma empresa por mais de cinco anos. Cada heurística tem sua respectiva experimentação em tal base de dados e seus resultados são avaliados para medir a eficiência neste cenário. A maior parte dos experimentos foi executada com base nas regras e nos processos das heurísticas e alguns deles foram executados utilizando as implementações produzidas até o momento. A avaliação dos resultados consiste na comparação das ocorrências apontadas na execução de tais heurísticas em tal base de dados com o que, de fato, é pertinente, quais são os falsos positivos e quais oportunidades estas heurísticas deixaram de detectar.

No Quadro 8 consta a lista de tabelas do esquema alvo dos experimentos. Este quadro foi extraído da base de dados *information_schema*, base esta que possui dados sobre os diferentes esquemas contidos em servidores MySQL. Tal quadro também é utilizado para identificar oportunidades de refatoração de elementos em desuso, normalizar nomenclatura, tabelas de referência inexistentes e para verificações de demais heurísticas. A base de dados, objeto da análise, foi gerada quando o sistema Eventum foi instalado em 20/11/2009. Essa base de dados foi alvo de uma migração de servidores, na qual foi necessário recriá-la em 23/05/2014. Por esse motivo, muitas tabelas possuem essa mesma data de última atualização. Portanto, o limiar de tempo considerado nas heurísticas é de, no mínimo, um ano e dois meses, pois os experimentos foram realizados a partir de 29/07/2015.

O MySQL possui vários tipos de motores para a criação de suas tabelas. Os mais utilizados são *myisam* e *innodb*. A diferença é que o primeiro provê melhor desempenho por não possuir suporte à integridade referencial (o segundo possui tal suporte). A base de dados do Eventum utiliza o padrão *myisam*, o que faz com que não tenha chaves estrangeiras e, portanto, apresente uma extrema deficiência de tabelas de referência. Não foi encontrada uma razão lógica para que o sistema em questão utilize tal motor; a hipótese mais provável é que se optou por utilizá-lo por ser o motor mais primitivo encontrados em bases de dados MySQL.

Como é um esquema que possui 67 tabelas e 469 colunas, alguns experimentos utilizam parte deste esquema. Se fosse utilizada a totalidade do esquema em todos os experimentos, iria demandar mais tempo para a execução das heurísticas e, até mesmo, estender de forma desnecessária este trabalho.

(continua)

Nome	Registros	Criação	Atualização
evt_columns_to_display	703	20-11-2009 08:42	02-06-2015 15:56
evt_custom_field	3	20-11-2009 08:42	02-06-2015 15:57
evt_custom_field_option	0	20-11-2009 08:42	17-03-2015 16:27
evt_custom_filter	19	20-11-2009 08:42	18-06-2015 10:58
evt_customer_account_manager	1	20-11-2009 08:42	23-05-2014 08:42
evt_customer_note	1	20-11-2009 08:42	23-05-2014 08:42
evt_email_account	3	20-11-2009 08:42	23-07-2015 09:49
evt_email_draft	34	20-11-2009 08:42	13-05-2015 10:04
evt_email_response	8	20-11-2009 08:42	17-03-2015 16:30
evt_faq	14	20-11-2009 08:42	23-05-2014 08:42
evt_faq_support_level	0	20-11-2009 08:42	23-05-2014 08:42
evt_group	0	20-11-2009 08:42	20-11-2009 08:42
evt_history_type	60	20-11-2009 08:42	23-05-2014 08:42
evt_irc_notice	0	20-11-2009 08:42	23-05-2014 08:42
evt_issue	16855	20-11-2009 08:42	27-07-2015 17:12
evt_issue_association	28	20-11-2009 08:42	23-05-2014 08:43
evt_issue_attachment	4979	20-11-2009 08:42	27-07-2015 14:49
evt_issue_attachment_file	7027	20-11-2009 08:42	27-07-2015 14:49
evt_issue_checkin	0	20-11-2009 08:42	23-05-2014 08:46
evt_issue_custom_field	6355	20-11-2009 08:42	27-07-2015 15:36
evt_issue_history	198789	20-11-2009 08:42	27-07-2015 17:12
evt_issue_quarantine	0	20-11-2009 08:42	23-05-2014 08:46
evt_issue_requirement	0	20-11-2009 08:42	23-05-2014 08:46
evt_issue_user	17582	20-11-2009 08:42	27-07-2015 17:12
evt_issue_user_replier	1021	20-11-2009 08:42	24-07-2015 16:30
evt_link_filter	0	20-11-2009 08:42	23-05-2014 08:46
evt_mail_queue	126401	20-11-2009 08:42	27-07-2015 17:13
evt_note	18390	20-11-2009 08:42	27-07-2015 15:53
evt_phone_support	1	20-11-2009 08:42	23-05-2014 08:51
evt_project	3	20-11-2009 08:42	23-07-2015 09:49
evt_project_category	20	20-11-2009 08:42	23-07-2015 10:40
evt_project_custom_field	4	20-11-2009 08:42	18-06-2015 10:58
evt_project_email_response	32	20-11-2009 08:42	17-03-2015 16:30
evt_project_field_display	152	20-11-2009 08:42	23-05-2014 08:51
evt_project_group	0	20-11-2009 08:42	23-05-2014 08:51
evt_project_link_filter	0	20-11-2009 08:42	23-05-2014 08:51
evt_project_news	0	20-11-2009 08:42	23-05-2014 08:51
evt_project_phone_category	14	20-11-2009 08:42	23-05-2014 08:51
evt_project_priority	113	20-11-2009 08:42	02-10-2014 11:05
evt_project_release	0	20-11-2009 08:42	23-05-2014 08:51
evt_project_round_robin	6	20-11-2009 08:42	16-06-2014 17:08
evt_project_status	25	20-11-2009 08:42	23-07-2015 09:50

(conclusão)

evt_project_status_date	0	20-11-2009 08:42	23-05-2014 08:51
evt_project_user	52	20-11-2009 08:42	23-07-2015 09:49
evt_reminder_action	0	20-11-2009 08:42	23-05-2014 08:51
evt_reminder_action_list	0	20-11-2009 08:42	23-05-2014 08:51
evt_reminder_action_type	4	20-11-2009 08:42	23-05-2014 08:51
evt_reminder_field	8	20-11-2009 08:42	23-05-2014 08:51
evt_reminder_history	0	20-11-2009 08:42	23-05-2014 08:51
evt_reminder_level	4	20-11-2009 08:42	16-06-2014 17:07
evt_reminder_level_condition	0	20-11-2009 08:42	23-05-2014 08:51
evt_reminder_operator	8	20-11-2009 08:42	23-05-2014 08:51
evt_reminder_priority	7	20-11-2009 08:42	23-05-2014 08:51
evt_reminder_requirement	4	20-11-2009 08:42	16-06-2014 17:07
evt_reminder_triggered_action	0	20-11-2009 08:42	23-05-2014 08:51
evt_resolution	10	20-11-2009 08:42	23-05-2014 08:51
evt_round_robin_user	6	20-11-2009 08:42	27-07-2015 15:12
evt_search_profile	854	20-11-2009 08:42	27-07-2015 14:47
evt_status	11	20-11-2009 08:42	21-10-2014 13:43
evt_subscription	40738	20-11-2009 08:42	27-07-2015 17:12
evt_subscription_type	157433	20-11-2009 08:42	27-07-2015 17:12
evt_support_email	33354	20-11-2009 08:42	27-07-2015 17:08
evt_support_email_body	42628	20-11-2009 08:42	27-07-2015 17:08
evt_time_tracking	852	20-11-2009 08:42	27-07-2015 11:08
evt_time_tracking_category	4	20-11-2009 08:42	23-05-2014 08:55
evt_user	211	20-11-2009 08:42	16-07-2015 11:14
evt_user_alias	0	20-11-2009 08:42	23-05-2014 08:55
evt_version	1	20-11-2009 08:42	23-05-2014 08:55

Quadro 8 - Tabelas do esquema Eventum

Por fim, nas sessões seguintes, as execuções das heurísticas são relatadas conforme as experiências obtidas perante as situações enfrentadas na prática.

5.1. Elementos em Desuso

Ao aplicar as regras nas tabelas do Quadro 8 e em suas colunas e determinar a data de 24/05/2014 como limiar, é possível reportar oportunidades de refatoração de elementos obsoletos da seguinte forma:

a) tabelas consideradas em desuso por não possuírem dados (Quadro 9):

evt_custom_field_option	evt_faq_support_level	evt_irc_notice
evt_issue_checkin	evt_issue_quarantine	evt_issue_requirement
evt_link_filter	evt_reminder_trigered_action	evt_project_group
evt_project_link_filter	evt_reminder_level_condition	evt_reminder_history
evt_project_news	evt_project_release	evt_project_status_date
evt_user_alias	evt_reminder_action	evt_reminder_action_list
evt_reminder_action_list	evt_group,	

Quadro 9 - Tabelas em desuso por não possuirem dados

Comando de seleção de dados executado para capturar as ocorrências:

SELECT *TABLE_NAME* ***FROM*** *information_schema.TABLES* ***WHERE*** *TABLE_SCHEMA = 'eventum'* ***AND*** *TABLE_ROWS = 0;*

b) colunas em desuso por não possuírem dados considerando as vinte primeiras tabelas, e que não se enquadram no item anterior (Quadro 10):

evt_custom_field.fld_backend	evt_custom_filter.cst_keywords
evt_custom_filter.cst_iss_pre_id	evt_email_account.ema_use_routing
evt_custom_filter.cst_closed_date	evt_issue.iss_contact_person_fname
evt_custom_filter.cst_show_notification_list	evt_custom_filter.cst_created_date
evt_custom_filter.cst_closed_date_end	evt_issue.iss_developer_est_time
evt_custom_filter.cst_first_response_date_end	evt_custom_filter.cst_is_global
evt_custom_filter.cst_created_date_filter_type	evt_custom_filter.cst_closed_date_end
evt_custom_filter.cst_created_date_time_period	evt_issue.iss_contact_person_lname
evt_issue.iss_customer_id	evt_issue.iss_customer_contact_id
evt_issue.iss_customer_contract_id	

Quadro 10 - Lista de colunas em desuso por não possuírem dados

c) tabelas que não possuem dados e que não são alvo de exclusões e atualizações a partir de 24/05/2014 (Quadro 11):

evt_customer_account_manager	evt_customer_note	evt_faq
evt_issue_association	evt_phone_support	evt_project_field_display
evt_project_phone_category	evt_reminder_field	evt_version
evt_reminder_action_type	evt_resolution	evt_reminder_priority
evt_time_tracking_category	evt_history_type	evt_reminder_operator

Quadro 11 - Tabelas em desuso por não sofrerem atualizações a partir de 24/05/2015

Comando de seleção de dados executado para capturar as ocorrências:

SELECT *TABLE_NAME* ***FROM*** *information_schema.TABLES* ***WHERE*** *TABLE_SCHEMA* ***LIKE*** *'eventum'* ***AND*** *TABLE_ROWS > 0 AND UPDATE_TIME <= '2014-05-24';*

d) colunas que não possuem dados e que não são alvo de exclusões e atualizações a partir de 24/05/2014 , considerando as 20 primeiras tabelas que não se enquadram no item "a)" e "c)", e colunas que não pertençam às ocorrências do item "b)" (Quadro 12) apenas para filtrar de forma mais objetiva:

evt_email_draft.emd_unknown_user	evt_issue.iss_contact_person_fname
evt_issue.iss_impact_analysis	evt_issue.iss_contact_person_lname.

Quadro 12 - Colunas em desuso por não sofrerem atualizações a partir de 24/05/2015

Exemplo de comando para identificar as tabelas que tenham as colunas alvo deste tipo de verificação:

SELECT TABLE_NAME **FROM** information_schema.TABLES **WHERE** TABLE_SCHEMA **LIKE** 'eventum' **AND** TABLE_ROWS > 0 **AND** UPDATE_TIME >= '2014-05-24' LIMIT 20

Exemplo de comando para verificar a obsolescência de uma coluna para a regra em questão:

SELECT * **FROM** eventum.evt_email_draft **WHERE** emd_updated_date >= '2014-05-24' **AND** emd_unknown_user **IS NOT NULL**

Se a consulta acima retornasse resultados a coluna não seria marcada como em desuso.

e) elementos dos itens "c)" e "d)" que não são alvo de seleção de dados:

Tabelas (Quadro 13):

evt_customer_account_manager	evt_customer_note	evt_issue_association
evt_reminder_priority	evt_phone_support	evt_project_phone_category
evt_reminder_action_type	evt_version	evt_time_tracking_category
evt_reminder_operator	evt_reminder_field	

Quadro 13 - Tabelas que não são alvo de seleção de dados

Colunas: nenhuma das ocorrências mensuradas no item "d)".

5.1.1. Avaliação dos Resultados

Dos 57 elementos apontados como em desuso pela aplicação da heurística, 52 são pertinentes após uma análise sobre o domínio. Isso representa uma precisão de 91,23% no cenário em questão. Os 8,77% de desvio não necessariamente representam falsos positivos, porque a regra "e)" da seção 3.1.1 serve, justamente, para reforçar a verificação daqueles elementos que embora não sofram adição, atualização ou exclusão de dados, mas ainda são utilizados pela aplicação apenas como fonte de consulta. Exemplos disso foram as seguintes tabelas marcadas pela regra "d)" (experimento "c)") e que não podem ser excluídas por serem utilizadas pela aplicação da regra "e)":

a) *evt_resolution*: tipo de resolução dada aos chamados;
b) *evt_project_field_display*: possui os campos que devem ser mostrados para a abertura de chamados;
c) *evt_history_type*: tipo de histórico sobre um determinado chamado;
d) *evt_time_tracking_category*: tipo de atendimento realizado sobre um chamado;
e) *evt_version*: possui a versão do sistema.

Em contrapartida, todas as colunas mensuradas no item "d)" da experimentação ainda são alvo em consultas, mas isso acontece porque a aplicação não foi ajustada pra desconsiderá-las. Entretanto, a ferramenta Eventum proporciona a personalização de quais campos devem ser utilizados nos formulários de abertura e edição de chamados. Esse recurso da ferramenta nada mais faz do que, justamente, ignorar colunas da tabela "evt_issue" para carregamento e manipulação dos registros. Infelizmente, o sistema não possui esta possibilidade para as demais partes do sistema. Recursos como esses facilitam a verificação da pertinência da existência de um determinado elemento em alto nível e em qualquer sistema.

5.2. Tabelas de Referência Faltantes

Como o intuito é demonstrar de forma simples e direta uma experimentação do processo sugerido na seção 3.2.2, foram consideradas as colunas das primeiras quinze tabelas do Quadro 8. Da mesma forma, como o motor utilizado nas tabelas do esquema alvo dos

experimentos é *myisam*, a aplicação das oportunidades de refatoração apontadas nesta seção implica em mudar o motor das tabelas para *innodb*, para que as chaves estrangeiras necessárias sejam adicionadas.

Estabelecendo um limiar de, no máximo, trinta agrupamentos gerados por coluna para determinar a possibilidade de necessitar uma tabela referência e, além disso, realizar verificações sob tais critérios, é possível inferir que as colunas que enquadram nesta situaão são as que constam no Quadro 14.

evt_columns_to_display.ctd_page	evt_custom_filter.cst_reporter	evt_faq.faq_usr_id
evt_columns_to_display.ctd_rank	evt_email_draft.emd_iss_id	evt_issue.prj_id
evt_issue.iss_last_public_action_type	evt_issue.iss_duplicated_iss_id	evt_issue.prc_id
evt_custom_field.fld_report_form	evt_email_draft.emd_usr_id	evt_issue.iss_sta_id
evt_columns_to_display.ctd_field		

Quadro 14 - Colunas que possivelmente necessitam de tabela de referência

Dentre as colunas apontadas anteriormente, aquelas que podem ter uma tabela de referência reaproveitada considerando zero por cento de registros não encontrados durante a conferência de possíveis vínculos existentes, porém não formalizados de forma relacional são as apresentadas no Quadro 15:

Coluna	**Possíveis tabelas de referência**
evt_custom_filter.cst_reporter	evt_user
evt_email_draft.emd_iss_id	evt_issue
evt_email_draft.emd_usr_id	evt_user
evt_faq.faq_usr_id	evt_user
evt_issue.prj_id	evt_project
evt_issue.prc_id	evt_project_category
evt_issue.iss_duplicated_iss_id	evt_issue

Quadro 15 - Lista de colunas e suas possíveis tabelas de referência já existentes

Durante este experimento foram filtradas aquelas tabelas de acordo com a nomenclatura das colunas conforme sugerido no processo. Se não fosse levada em consideração a proximidade dos nomes das tabelas e as supostas chaves estrangeiras a lista de possíveis tabelas de referência seria bem maior, consequentemente aumentando a quantidade de verificações necessárias para a tomada de decisão.

5.2.1 Avaliação dos Resultados

Nas quinze tabelas analisadas foram encontradas treze colunas que representam possíveis oportunidades de adicionar tabelas de referência ou reaproveitar alguma existente conforme o Quadro 15. Dentre as colunas apontadas identificou-se que a *evt_columns_to_display.ctd_rank* dispensa a necessidade de uma tabela de referência para aprimorar sua representação. Além disso, o propósito de existência da coluna *evt_custom_field.fld_report_form* não foi identificado para saber se a mesma necessitaria ou não de uma tabela de referência. Todas as tabelas identificadas para reaproveitamento no Quadro 15 realmente procedem. Ao executar o processo de acordo com as regras da seção 3.2.1, as seguintes colunas, que também apresentam a possibilidade deste tipo de refatoração, não foram capturadas pela heurística:

a) *evt_custom_field.fld_type*. Como a tabela que contém tal coluna possui apenas três registros e com valores diferentes nesta coluna, foi refutada devido a regra "e)". Identificou-se que uma tabela de referência é adequada para a representação porque esta coluna representa os tipos que os campos customizados da ferramenta podem ter como, por exemplo, *text*, *integer*, *date*, etc.;
b) *evt_email_account.ema_type*. Essa coluna também não foi considerada devido a regra "e)"
c) *evt_issue.iss_usr_id*. Essa coluna pode reaproveitar a referência para a tabela *Evt_issue* e não foi capturada porque os agrupamentos gerados por esta coluna excedem o limiar estabelecido.

Desse modo, é possível concluir que, para a amostra em questão, a heurística teve uma precisão de 84,61%. Se for identificado que o propósito de existência da coluna *evt_custom_field.fld_report_form* e descobrir que a mesma é passível de uma tabela de referência para representar seus dados, o percentual de precisão é de aproximadamente 92,30%. Embora tenham ocorrências não capturadas devido a regra "e)", isso não significa que a mesma, possivelmente, deva ser retirada da heurística, porque ela serve para desconsiderar aquelas colunas que não geram agrupamentos. A geração de agrupamentos é um dos principais indícios que uma coluna pode ter para determinar que se trata de um conjunto de valores, sob um determinado contexto e que, portanto, se documentados adequadamente na base de dados, terão uma representatividade melhor.

5.3. Colunas com Tipos ou Tamanhos Inadequados

Para executar a heurística proposta na seção 3.3, foram adotados os seguintes critérios sobre os parâmetros de entrada nas 106 colunas das dez primeiras tabelas listadas no Quadro 8, 1% de tolerância para conversões falhas e 10% de tolerância para novos registros que possam exceder o tamanho da maior ocorrência atual. Ao executar o processo sugerido sob essas condições, o Quadro 16 foi gerado como resultado.

(continua)

Coluna	**Tipo atual**	**Tipo proposto**
evt_columns_to_display.ctd_prj_id	Int(10)	Smallint
evt_columns_to_display.ctd_page	Varchar(20)	Varchar(12)
evt_columns_to_display.ctd_min_role	Tinyint(3)	Tinyint(1)
evt_columns_to_display.ctd_rank	Tinyint(3)	Tinyint(1)
evt_custom_field.fld_id	Int(10)	Smallint
evt_custom_field.fld_report_form	In(1)	Bool
evt_custom_field.fld_report_form_required	In(1)	Bool
evt_custom_field.fld_anonymous_form	In(1)	Bool
evt_custom_field.fld_anonymous_form_required	In(1)	Bool
evt_custom_field.fld_close_form	Tinyint(1)	Bool
evt_custom_field.fld_close_form_required	Tinyint(1)	Bool
evt_custom_field.fld_list_display	Tinyint(1)	Bool
evt_custom_field.fld_min_role	Tinyint(1)	Bool
evt_custom_filter.cst_id	Int(10)	Tinyint(3)
evt_custom_filter.cst_hide_closed	Int(1)	Bool
evt_custom_filter.cst_is_global	Int(1)	Bool
evt_custom_filter.cst_search_type	Varchar(15)	Varchar(9)
evt_customer_account_manager.cam_id	Int(11)	Tinyint(1)
evt_customer_account_manager.cam_prj_id	Int(11)	Tinyint(1)
evt_customer_account_manager.cam_customer_id	Int(11)	Tinyint(1)
evt_customer_account_manager.cam_usr_id	Int(11)	Tinyint(1)
evt_customer_note.cno_id	Int(11)	Tinyint(1)
evt_customer_note.cno_prj_id	Int(11)	Tinyint(1)
evt_customer_note.cno_customer_id	Int(11)	Tinyint(1)
evt_email_account.ema_id	Int(11)	Tinyint(2)
evt_email_account.ema_prj_id	Int(10)	Tinyint(2)
evt_email_account.ema_hostname	Varchar(255)	Varchar(100)
evt_email_account.ema_password	Varchar(64)	Varchar(32)
evt_email_account.ema_get_only_new	Int(1)	Bool
evt_email_account.ema_leave_copy	Int(1)	Bool
evt_email_account.ema_use_routing	Int(1)	Bool
evt_email_account.ema_issue_auto_creation_options	Text	Varchar(10)
evt_email_draft.emd_id	Int(11)	Tinyint(3)
evt_email_draft.emd_usr_id	Int(11)	Tinyint(3)

(conclusão)

evt_email_draft.emd_iss_id	Int(11)	Int(6)
evt_email_draft.emd_sup_id	Int(11)	Tinyint(3)
evt_email_draft.emd_subject	Varchar(255)	Varchar(200)
evt_email_response.ere_id	Int(11)	Smallint
evt_faq.faq_id	Int(11)	Smallint
evt_faq.faq_prj_id	Int(11)	Smallint
evt_faq.faq_usr_id	Int(11)	Int(6)
evt_faq.faq_title	Varchar(255)	Varchar(100)

Quadro 16 - Oportunidades identificadas para tipos e tamanhos inadequados

5.3.1. Avaliação dos Resultados

Das 106 colunas verificadas, 42 (39,62%) foram marcadas para adequação de tipo ou tamanho. Dentre as colunas marcadas, as ocorrências da tabela *evt_customer_account_manager* podem desencadear algum problema se aplicadas por possuir, apenas, um registro e se as sugestões forem muito acentuadas por conta disso. Em contrapartida, se em cinco anos de existência da base de dados essa tabela não tiver expandido, é pouco provável que ocorram problemas de dimensionamento de tais colunas.

Durante os experimentos, também foi identificada a possibilidade de desconsiderar chaves estrangeiras nas regras de verificação para evitar falha de incompatibilidade de tipos com suas respectivas referências. Essa situação não foi considerada durante o desenvolvimento das heurísticas, mas pode ser contemplada no formato de regra para aumentar confiabilidade na aplicação das oportunidades de refatoração identificadas.

Os resultados do experimento foram revisados e não foi encontrado algum falso positivo ou ocorrência que a heurística deixou de capturar. Desse modo, foi apresentada uma precisão de 100% para a amostra em questão.

Embora, em alguns casos, a determinação de um campo com tamanho mais justo possa contribuir também para melhorar a representatividade, quando se trata do tipo *Varchar* para o banco de dados MySQL, não são apresentadas vantagens sobre a performance. Isso porque o armazenamento dos dados é alocado conforme a demanda, diferentemente do tipo *Char*, que sempre aloca todo o tamanho do campo preenchendo com espaço as posições não utilizadas na *string* armazenada.

5.4. Valores Padrões Faltantes

Para executar o experimento de identificar oportunidades de refatoração de colunas com falta de valor padrão, foi utilizada a implementação do Algoritmo 4. O limiar de predominância mínima, adotado para considerar um valor padrão, é de 60% sobre as colunas das quinze primeiras tabelas do Quadro 8. O resultado deste experimento está representado no Quadro 17.

(Continua)

Coluna	**Possível valor padrão**
evt_columns_to_display.ctd_page	list_issues
evt_columns_to_display.ctd_min_role	1
evt_custom_field.fld_anonymous_form	0
evt_custom_field.fld_anonymous_form_required	0
evt_custom_field.fld_close_form	0
evt_custom_field.fld_close_form_required	0
evt_custom_field.fld_list_display	0
evt_custom_filter.cst_prj_id	1
evt_custom_filter.cst_iss_pri_id	0
evt_custom_filter.cst_iss_sta_id	0
evt_custom_filter.cst_iss_pre_id	0
evt_custom_filter.cst_created_date	Null
evt_custom_filter.cst_created_date_filter_type	Null
evt_custom_filter.cst_created_date_time_period	0
evt_custom_filter.cst_created_date_end	Null
evt_custom_filter.cst_updated_date	Null
evt_custom_filter.cst_updated_date_time_period	0
evt_custom_filter.cst_updated_date_end	Null
evt_custom_filter.cst_last_response_date	Null
evt_custom_filter.cst_last_response_date_end	Null
evt_custom_filter.cst_first_response_date	Null
evt_custom_filter.cst_first_response_date_filter_type	Null
evt_custom_filter.cst_first_response_date_end	Null
evt_custom_filter.cst_closed_date	Null
evt_custom_filter.cst_closed_date_filter_type	Null
evt_custom_filter.cst_closed_date_end	Null
evt_custom_filter.cst_sort_by	iss_pri_id
evt_custom_filter.cst_sort_order	asc
evt_email_account.ema_hostname	outlook.office365.com
evt_email_account.ema_port	995
evt_email_account.ema_get_only_new	0
evt_email_account.ema_issue_auto_creation	disabled
evt_email_account.ema_use_routing	0

(conclusão)

evt_email_draft.emd_unknown_user	Null
evt_history_type.htt_role	0
evt_issue.iss_customer_id	0
evt_issue.iss_customer_contact_id	0
evt_issue.iss_customer_contract_id	0
eventum.issue.iss_prj_id	1
evt_issue. iss_sta_id	5
evt_issue.iss_prc_id	142
evt_issue.iss_res_id	2
evt_issue.iss_percent_complete	0
evt_issue.iss_duplicated_iss_id	Null
evt_issue.iss_last_customer_action_date	Null
evt_issue.iss_expected_resolution_date	Null

Quadro 17 - Oportunidades identificadas para valores padrões faltantes

Durante a execução deste experimento, observou-se a necessidade de adicionar ao conjunto de regras da heurística uma verificação sobre a quantidade de registros mínimos que as tabelas que contém as colunas alvo deste processo devem ter. Isso evita, por exemplo, a indicação de valores padrões que possam não ser adequados, futuramente, por uma tabela que possui poucos registros atualmente, mas que, se utilizada em larga escala, pode apresentar uma diversidade de valores bem diferente para suas colunas. Um exemplo disso é o caso da tabela *evt_email_account*, utilizada para armazenar as contas de e-mail para envio de mensagens via SMTP que a aplicação possa necessitar. Certamente, poderia modificar, ou até mesmo anular, a indicação de valor padrão para a coluna *ema_hostname*.

5.4.1. Avaliação dos Resultados

Das 159 colunas analisadas, 46 (28,93% sobre o total analisado) apresentaram uma indicação de valor padrão. Ao verificar a viabilidade de aplicar as indicações do Quadro 17, constataram-se os seguintes falsos positivos:

a) *evt_email_account.ema_hostname*: embora o valor indicado seja pertinente para o contexto atual, tem grande probabilidade de tal valor não ser apropriado se a tabela *evt_email_account* for populada com uma diversidade maior de valores;
b) *evt_issue.iss_sta_id*: esta coluna representa os status que um ticket (chamado) pode

assumir ao longo do seu ciclo de vida. O valor “5” indica que o chamado foi fechado. Normalmente, esse valor é determinado quando o atendimento é considerado encerrado. Ele é predominante perante os demais valores, porque um chamado passa por outros status até assumir tal valor. Evidentemente, ao longo dos anos, um sistema como esse vai apresentar mais chamados fechados do que em processo de atendimento. Por esse motivo, um chamado (registro da tabela *evt_issue*) não pode nascer com valor “5”, pois normalmente, de acordo com as regras de negócio do sistema, ele nasce como aberto (valor “1”);

c) *evt_issue.iss_res_id*: esta coluna indica o tipo de resolução do chamado ou motivo para finalização do chamado. O valor “2” significa que o chamado foi corrigido (*fixed*). Embora seja um valor que reflita a predominância, a colocação desse valor como padrão também não é indicada, porque esta coluna deve ser preenchida apenas quando um chamado for fechado.

Por fim, analisando os resultados, não foi identificada alguma falta de indicação de valor padrão e, como foram encontrados apenas três falsos positivos diante das oportunidades indicadas pela heurística, conclui-se uma precisão de 93,48% da proposta perante a amostra. Além disso, se observou que, se fossem aplicadas as oportunidades de refatoração apontadas nos experimentos anteriores, algumas colunas não seriam apontadas no Quadro 17.e, consequentemente, o resultado seria mais objetivo por não conter, por exemplo, colunas obsoletas. Também seria possível indicações de valor padrão diferentes dependendo das ações tomadas como, por exemplo, para colunas convertidas em *flags*.

5.5. Padronização de Formatos

O experimento para detectar formatos de dados que podem ser padronizados inicialmente foi realizado através da execução de métodos da classe *StandardizeFormatDetector* e, posteriormente, através do processo 3.5.3 nas seguintes colunas:

a) *evt_mail_queue.maq_recipient*: coluna destinada a armazenar o nome completo de uma pessoa e seu respectivo nome sobre um determinado formato. Um exemplo de valor válido que essa coluna pode assumir é: “Luiz Fogliato Jr” <fogliato.jr@gmail.com>. Tal formato pode ser validado pela seguinte expressão

regular: ^\"[a-zA-Z].*\"\ \<[a-zA-Z0-9][a-zA-Z0-9\._-]+@([a-zA-Z0-9\._-]+\.)br\>$;

b) *evt_mail_queue.maq_sender_ip_address*: coluna destinada a armazenar endereços de IP. Endereços de IP podem ser validados pela seguinte expressão regular: ^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$;

c) *evt_support_email.sup_from*: destinada a armazenar endereços de e-mail. Um endereço de e-mail pode ser validado pela seguinte expressão regular: ^[a-zA-Z0-9][a-zA-Z0-9\._-]+@([a-zA-Z0-9\._-]+\.)br$

Ao aplicar a heurística em tais colunas, o Quadro 18 foi gerado como resultado. As ocorrências fora do padrão da coluna *evt_mail_queue.maq_recipient* apresentavam valores com falhas na codificação de caracteres especiais. Por exemplo, o dado que deveria ser armazenado como “Cláudia Razza” <claudia.razza@xyz.com.br> estava como =?UTF-8?Q?Cl=C3=A1udia_Razza?= <claudia.razza@xyz.com.br>. Na coluna *maq_sender_ip_address*, duas ocorrências fora do padrão possuíam letras. Em outra, faltava o ponto como dígito separador e, nas duas últimas, havia números de IP com quatro dígitos antes de um dos separadores.

A coluna *evt_support_email.sup_from* apresentou 30 registros com o “.br” em que faltava o endereço de e-mail e o restante eram valores com simples junções de caracteres alfanuméricos, apenas para satisfazer a obrigatoriedade de preenchimento do campo.

Coluna	**Registros fora do padrão**	**Percentual em relação ao total**	**Registros não capturados pela Regex**
evt_mail_queue.maq_recipient	11180	8,84%	0
evt_mail_queue.maq_sender_ip_address	5	0,004%	0
evt_support_email.sup_from	122	0,365%	0

Quadro 18 - Relação de oportunidades identificadas para padronizar formatos

Não foi possível executar o experimento para a primeira coluna usando a implementação da classe *StandardizeFormatDetector* porque o método *ConvertSamplesToRegex* não gerou uma expressão regular capaz de identificar o padrão desejado a partir de vários exemplos verídicos informados.

5.5.1. Avaliação dos Resultados

Embora no experimento realizado não houvesse ocorrências de registros não capturados pela expressão regular (regex), o que afere total precisão da heurística nos casos avaliados, obviamente, podem acontecer casos de baixíssima precisão. Por exemplo, se a *regex* validadora não for suficientemente capaz de cobrir todas as possibilidades de valores válidos. Por motivos desconhecidos sobre o comportamento da aplicação, os nomes que ficavam entre aspas duplas do formato definido para a coluna *evt_mail_queue.maq_recipient* não possuíam acentos. Caso contrário a execução do processo teria severamente falhado. Isso porque a expressão regular utilizada não considera caracteres acentuados no início dos nomes.

Ademais, o fato de a implementação realizada não ter efetividade sobre o primeiro caso, aponta que o método *ConvertSamplesToRegex* deve ser aperfeiçoado para cobrir uma gama maior de formatos que podem ser exemplificados na entrada do processo.

Por fim, embora a avaliação dos resultados indique estes contrapontos, a heurística proposta se demonstrou adequada para atender à demanda que foi destinada de alguma forma e deverá ser aprimorada para aumentar sua precisão.

5.6. Transformação de Colunas Nulas em Não Nulas e Vice-versa

Para identificar oportunidades de transformar colunas nulas em não nulas e vice-versa, foram consideradas as colunas das dez tabelas que apresentam a maior quantidade de registros. Sendo assim, as tabelas são as seguintes: *evt_mail_queue_log, evt_note, evt_issue, evt_issue_history, evt_subscription_type, evt_mail_queue, evt_support_email_body, evt_subscription, evt_support_email e evt_issue_user*.

Aplicando o processo sugerido na seção 3.6.2 nas colunas das tabelas citadas, considerando um tempo mínimo de existência de dois anos, o Quadro 19 é gerado como resultado da busca de tais oportunidades de refatoração:

(continua)

Coluna	**Transformação**
evt_issue.iss_percent_complete	Nula para não nula

(conclusão)

evt_note.not_message_id	Nula para não nula
evt_mail_queue.maq_sender_ip_address	Não nula para nula
evt_support_email.sup_parent_id	Não nula para nula
evt_support_email.sup_to	Não nula para nula
evt_issue_user.isu_assigned_date	Nula para não nula

Quadro 19 - Resultado da busca de oportunidades de transformação de colunas

Durante a análise do conteúdo de algumas colunas de tipo inteiro que estavam definidas como nulas, mas preenchidas em todos os registros e que, portanto, poderiam ser marcadas para serem transformadas em não nulas, foram ignoradas porque possuíam valores zerados e conforme as regras devem ser nulas. Algumas das colunas que se enquadram nesta situação apresentavam, inclusive, grandes chances de adição de tabelas de referência. Isso faz com que seja conveniente transformar os valores zerados em nulos, não apenas por terem uma representatividade equivocada, mas, principalmente, porque as chaves primárias das possíveis tabelas de referência para reaproveitamento são auto incremento (que normalmente iniciam em 1).

5.6.1. Avaliação dos Resultados

A execução do processo de forma manual se demonstrou de forma mais eficiente que o Algoritmo 6 porque algumas colunas não nulas do tipo *tinyint* que possuíam apenas valores “0” e “1”, acabavam sendo marcadas para transformação em não nulas. Se essas colunas fossem convertidas, anteriormente, para um tipo booleano, esse problema não aconteceria com o algoritmo proposto. Como o algoritmo não possui uma conferência sobre a pertinência dos valores, algo nesse sentido deve ser feito. Uma alternativa para contornar o problema é, justamente, adicionar a verificação da possibilidade de converter a coluna em *flag* antes de marcar tais colunas para a transformação.

Todas as colunas apontadas para transformação são de pertinente aplicação, principalmente, a marcação da coluna *evt_mail_queue.maq_sender_ip_address* para permitir valores nulos. Como esta coluna serve para armazenar o IP dos usuários que executam ações no sistema que resultam em envio de e-mails e, nem sempre, o sistema consegue obter esta informação, logo, é melhor permitir valor nulo para explicitar esta possibilidade ao invés de

preencher o campo com espaço em branco só para satisfazer a obrigatoriedade. Dessa forma, melhora significativamente a representatividade da coluna.

Por fim, analisando as colunas alvo do processo, não foram encontradas ocorrências que a heurística deixou de capturar. Desse modo, para o experimento, afere-se total precisão para a detecção de oportunidades de refatoração para transformar colunas nulas em não nulas e vice-versa em tal amostra.

5.7. Normalização da Nomenclatura

Os elementos alvo deste experimento são todas as tabelas e colunas do esquema da base de dados. As tabelas possuem como regras de nomeação o prefixo “evt”, o dígito separador é “_” e as expressões que compõem os nomes devem estar no singular. Aplicando a implementação do algoritmo 7, presente na classe *NormalizeNamesDetector*, foi possível identificar apenas uma ocorrência de tabela que precisa ter a nomenclatura normalizada de acordo com o escopo especificado que foi a *evt_columns_to_display*.

Como as colunas possuem as mesmas regras das tabelas, exceto a prefixação, onde cada coluna deve ter as iniciais do nome da tabela a qual está contida, desconsiderando a letra “e” de “evt” e utilizando três caracteres. Devido a essa peculiaridade, a implementação deve ser executada de acordo com o conjunto de colunas pertencentes a cada tabela, o que, nesse caso torna a busca de oportunidades de refatoração um pouco morosa de realizar. Como resultado do processo, foram identificadas as seguintes colunas:

a) ocorrências por apresentar prefixos incoerentes: as quatorze colunas da tabela *evt_custom_field*, as três colunas da tabela *evt_email_response*, as três colunas da tabela *evt_project_phone_category*, e as três colunas da tabela *evt_reminder_action_type;*
b) ocorrências por não estarem no singular: as colunas *evt_user.usr_preferences* e *evt_issue.iss_trigger_reminders*.

5.7.1. Avaliação dos Resultados

Avaliando os resultados e verificando os elementos alvos do processo manualmente não foi possível identificar alguma coluna ou tabela que a heurística deixou de capturar. Além disso, todos os elementos apontados são realmente passíveis de normalização de nomenclatura. A tabela *evt_columns_to_display*, por apresentar a palavra "columns" que está no plural quando o padrão estabelecido para a base de dados são nomes no singular, o mesmo ocorre para algumas colunas citadas. Os casos de colunas marcadas para normalização da nomenclatura por prefixos incoerentes ocorrem por não condizerem com as iniciais do nome da tabela a qual pertencem. Por exemplo, a coluna *evt_project_phone_category.phc_title*, quando poderia ser nomeada como *evt_project_phone_category.ppc_title*.

Quanto à determinação dos prefixos, às vezes, pode depender da preferência do desenvolvedor como, por exemplo, a coluna *evt_email_response.ere_id*, que só foi considerada no processo porque as colunas das outras tabelas que começam com *evt_email_X*, normalmente, são prefixadas por *emX*, onde *X* é a inicial do próximo termo. Ou seja, seguindo essa lógica das outras ocorrências, essa coluna deveria ser nomeada como *evt_email_response.emr_id*. Isso leva a concluir que a forma que foi nomeada tal coluna, inicialmente, não necessariamente represente equívoco, mas não mantém conformidade com as demais nomeações.

Por fim, como a prefixação das colunas deveria ser analisada tabela à tabela os experimentos foram executados de forma pouco produtiva. Isto pode representar uma oportunidade de continuar com mais pesquisas e experimentos para aperfeiçoar o processo para capturar colunas com falhas na nomeação em todo esquema de uma só vez, ao invés de um processo fragmentado.

5.8. Considerações Finais

Os experimentos realizados foram em MySQL mas poderia ser qualquer outro SGBD através dos processos propostos. Ademais, se executados em outras bases de dados, os resultados possivelmente serão diferentes, pois a detecção de oportunidades de refatoração, assim como a refatoração em si, é uma atividade situacional.

Algumas heurísticas podem apresentar resultados mais drásticos do que os experimentos aqui realizados como, por exemplo, a detecção de tipos e tamanhos adequados que não encontrou ocorrências de converter um campo textual para numérico. A maior parte das ocorrências aponta para ajustes sobre a mesma categoria de tipo. De modo geral, os experimentos apresentaram resultados satisfatórios, pois em nenhum deles houve ocorrências de falsos positivos que comprometessem a eficiência sobre as amostras. O Quadro 20 resume a precisão das heurísticas nos experimentos realizados neste trabalho.

Heurística	**Precisão em percentual**
Elementos em desuso	91,23
Tabelas de referência	84,61
Tipos e tamanhos inadequados	100
Valores padrão	93,48
Padronizar formatos	100
Transformar colunas nulas em não nulas e vice-versa	100
Normalização da nomenclatura	100

Quadro 20 - Relação de precisão das heurísticas nos experimentos

Tal relação de precisão tem, certamente, grandes chances de ser diferente se as heurísticas forem aplicadas em outra base de dados. Os percentuais de precisão poderiam ser menores ou maiores até mesmo na própria base de dados do Eventum, se todos os elementos do esquema fossem considerados naqueles experimentos que trabalharam apenas com uma amostra.

6. CONCLUSÕES

Existe uma diversidade de trabalhos relacionados à aplicação de refatoração em bases dados. Porém, com foco especificamente na detecção de oportunidades de refatoração em bases de dados, não foram encontrados durante a pesquisa e isso é um diferencial desta dissertação, pois representa uma vasta linha de pesquisa a ser explorada. As aplicações das heurísticas propostas podem ser no meio acadêmico, para que, por exemplo, professores verifiquem a qualidade de modelagem que os alunos empregam nas bases de dados de projetos que possam desenvolver ou gerar dinamicamente. Sobretudo, pode ter diferentes aplicações no meio corporativo para melhorar a qualidade dos dados armazenados e tornar não apenas a base de dados, mas os sistemas, de um modo geral, mais evolutivos.

A detecção de oportunidades de refatoração em bases de dados através das heurísticas, principalmente com implementações que automatizem os processos sugeridos, além de propiciar a execução facilitada de auditoria sobre projetos de bases de dados existentes também torna uma atividade atraente para o ciclo de vida de aplicações. Isto porque tais implementações podem ser aplicadas até mesmo durante o desenvolvimento para, por exemplo, identificar a nomenclatura indevida de elementos decorrentes de falta de atenção durante a modelagem, também para identificar tipos ou tamanhos inadequados durante a fase de testes.

As propostas (regras, algoritmos e processos) se mostraram ser de completa viabilidade de aplicação, pois as regras normalmente foram incisivas para capturar as ocorrências de cada tipo de imperfeição, os algoritmos implementados foram de fácil transposição para uma linguagem de programação real e os processos também foram úteis para aquelas detecções que os algoritmos não cobriram ou que ainda não tinham implementações correspondentes. Estes fatores contribuíram diretamente para os satisfatórios resultados apresentados durante os experimentos. Contudo, conforme o exposto, algumas implementações ainda devem ser aperfeiçoadas e a ferramenta *Database Smell Detector* deve ser aprimorada em sua totalidade para que a prática da detecção de oportunidades de refatoração seja mais facilitada possível.

Por fim, durante os experimentos e a avaliação dos resultados ficou evidente a influência que a resolução de um conjunto de oportunidades de refatoração pode ter sobre outro, mesmo que sejam para propósitos diferentes. Por exemplo, conforme visto na sessão 5.6.1, a resolução dos casos de colunas com tipos e tamanho indevidos poderia resultar em

uma avaliação mais objetiva por parte da heurística para detectar colunas não nulas para nulas, gerando assim um ciclo de benefícios para a execução das atividades envolvidas e para a própria base de dados.

6.1. Trabalhos Futuros

Como a detecção de oportunidade de refatoração em bases de dados relacionais demonstrou uma disciplina promissora, a lista de pesquisas e trabalhos que podem ser desenvolvidos é vasta. Entretanto, os mais relevantes identificados são os seguintes:

- **estender este trabalho para adição de outras detecções que não foram contempladas no escopo deste trabalho.** Embora esse trabalho tenha uma cobertura e contribuições significativas em termos de imperfeições que podem ser detectadas, existem algumas que devem ser incorporadas em trabalhos futuros, como por exemplo, elementos redundantes, que foi cogitada inicialmente em ser contemplada por este trabalho e foi descartada porque são necessários esforços e pesquisas focadas neste nicho que, por si só, podem render uma dissertação à parte. Além disso, é conveniente investir mais pesquisas em algumas imperfeições que foram citadas na introdução, mas que, até o momento, não foram atendidas por este trabalho porque não foram encontradas formas de detectar através de processos, materiais relevantes para os seguintes casos: tabelas com muitas linhas, tabelas com muitas colunas, colunas e tabelas com vários propósitos de uso. Nesta linha de pesquisa, podem existir imperfeições que não foram citadas, mas que se forem passíveis de alguma heurística podem ser integradas a este trabalho. Por fim, pode-se desenvolver pesquisas para produção de heurísticas complementares sobre a detecção de imperfeições que devem ser corrigidas para que uma determinada base de dados seja normalizada na primeira forma normal, segunda ou até mesmo terceira conforme as abordagens de CORONEL E MORRIS (2014);
- **integrar algumas heurísticas com ferramentas de modelagem de bases de dados.** Muitas ferramentas *computer-aided software engineering* (CASE) estão disponíveis no mercado para modelar bases de dados, e que, por sua vez, geram bases de dados a partir dos modelos criados. Exemplos de ferramentas como essas são: Mysql WorkBench, Enterprise Architect, DBWrench, ERWin, entre outras. É comum

desenvolvedores utilizarem estas ferramentas no ciclo de desenvolvimento de software para gerar bases de dados a partir de Diagramas ER (entidade relacionamento) e/ou até mesmo a partir de diagramas de classes através de IDEs como o Visual Studio. Portanto, para minimizar as chances de bases de dados serem geradas com imperfeições como, por exemplo, elementos com nomenclaturas errôneas, é pertinente o investimento em pesquisas e trabalhos que integrem heurísticas de detecção de oportunidades de refatoração com tais ferramentas para que seja possível realizar detecções em tempo de modelagem, evitando, assim, retrabalhos que futuramente podem ser desencadeados;

- **realizar comparativos entre as heurísticas e ferramentas de auxílio para a solução de problemas pontuais.** Conforme citado, durante as pesquisas deste trabalho não foram encontrados trabalhos que tratem especificamente da detecção de oportunidades de refatoração em bases de dados. Contudo, se for investido mais tempo em pesquisas de ferramentas que contribuam para alguma heurística, é pertinente que comparações sejam feitas com as propostas realizadas para melhorar, complementar ou até mesmo integrar às soluções aqui fornecidas. Por exemplo, se fosse encontrada alguma ferramenta capaz de "varrer" um determinado esquema de base de dados em busca de colunas com tipos ou tamanhos inadequadas, obviamente sua forma de atuação deve ser comparada com a heurística que propõe resolver o mesmo tipo de problema e integrar tal ferramenta ou parte de seus processos à solução se apresentar melhores resultados;
- **produção de processos automáticos para a execução das heurísticas que compõem o trabalho.** Os processos aqui sugeridos, mesmo que executados por meio de alguma implementação, consistem de uma única execução a partir de determinados parâmetros de entrada. Entretanto, as propostas podem ser evoluídas para um formato que seja executado de forma automática. Para exemplificar isso, ao invés de ter uma interface gráfica que o usuário informe os parâmetros do Algoritmo 3 para a execução da respectiva implementação, é possível definir um ponto centralizador para o armazenamento de tais parâmetros, como por exemplo, um arquivo de configuração que seja lido por um processo no formato de serviço do sistema operacional ou uma aplicação do tipo console que seja executada sob uma determinada frequência e notifique os interessados quais campos que podem ser modificados para ficarem mais condizentes com o estado atual das informações contidas na base de dados. Os sistemas operacionais atuais oferecem nativamente ferramentas para realizar a

execução de programas de forma agendada e recorrente que podem ser utilizadas como parte desta estratégia. Desse modo, é possível manter a base de dados constantemente monitorada para identificar imperfeições.

REFERÊNCIAS BIBLIOGRÁFICAS

ALTERVISTA. **Thesauros Web service**. Disponível em: <http://thesaurus.altervista.org> Acesso em 13 de jun. 2015.

AMBLER, S. W. **Agile Modeling**: Best Practices for the Unified Process and Extreme Programming. Addison Wesley Professional. 2002.

AMBLER, S. W. **Agile Database Techniques**: Effective Strategies for the Agile Software Developer. Programming. Addison Wesley Professional. 2003.

AMBLER, S. W.; SADALAGE, P. J. **Refactoring Databases**: Evolutionary Database Design. Addison Wesley Professional. 2006.

ANATEL. **REGULAMENTO DO SERVIÇO MÓVEL PESSOAL – SMP.** 2003. Disponível em: <http://legislacao.anatel.gov.br/resolucoes/2007/9-resolucao-477> Acesso em 17 jul. de 2014.

ANDRITSOS, P.; FUXMAN, A.; MILLER, R. J. **Clean Answers over Dirty Databases**: A Probabilistic Approach. Publicado na conferência internacional de engenharia de dados pela IEEE, 2006.

BARONI, A.; ABREU, F.; CALERO, C. **Finding Where To Apply Object-Relational Database Schema Refactorings**: An Ontology-Guided Approach. 2005. Disponível em: < http://ctp.di.fct.unl.pt/~mgoul/papers/2005/2005_JISBD.pdf>. Acesso em 14 de abr. 2014.

BILENKO, M.; MOONEY, R. J. **Adaptive Duplicate Detection Using Learnable String Similarity Measures**. In: ACM SIGKDD International Conference On Knowledge Discovery And Data Mining, KDD, 9., 2003. Proceedings. . . New York: ACM, 2003. p.39–48.

BORATE, S. **How to Check When a MySQL Table Was Last Updated.** 2011. Disponível em <http://www.codediesel.com/mysql/how-to-check-when-a-mysql-table-was-last-updated/> Acesso em 16 de jul. 2014.

BROCKE, J. V.; ROSERMAN M. **Manual de BPM**: gestão de processos de negócio. Bookman. 2010.

CORONEL, C.; MORRIS, S. **Database Systems:** Design, Implementation and Management. Cengage Learning. 2014.

FAROULT, S.; HERMITE, P. **Refactoring SQL Applications**. O Reilly. 2008.

FELLEGI, I. P.; SUNTER, A. B. **A Theory for Record Linkage**. Journal of the American Statistical Association, [S.l.], v.64, n.328, p.1183–1210, 1969.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. Addison Wesley Professional.1999.

FOGLIATO, L. J. **Emprego de Ontologias para Descrever a Evolução em Bases de Dados Relacionais**. Artigo da disciplina de projeto e especificação de ontologias. UFSM. 2013. Disponível em: < https://www.dropbox.com/s/j0miojkad2vneiu/PEO3_LuizFogliatoJuniorSBC.pdf> Acesso em 17 de abr. 2014.

FOGLIATO, L. J. **Técnicas Evolucionárias para Refatoração em Banco de Dados.** Relatório da disciplina de trabalho individual I. UFSM. 2013. Disponível em: < https://www.dropbox.com/s/dl5gcw7yrmif8xe/TI_LuizFogliatoJunior.pdf > Acesso em 27 de jul. 2014.

GAMMA, E.; Vlissides, J.; Johnson, R.; Helm, R. et al. **Design Patterns**: elements of reusable object-oriented software. [S.1.: s.n.]: Addison-Wesley, 1999.

GOMES, I. **Dicionário Aberto**. Universidade do Minho. Disponível em: < http://dicionario-aberto.net/ > Acesso em 13 de jun. 2015.

GUTH, G. J. A. **Surname Spellings and Computerized Record Linkage**. Historical Methods Newsletter, [S.l.], v.10, n.1, p.10–19, December 1976.

JUNIOR, J. L. **Descoberta de Equivalência Semântica entre Atributos em Bancos de Dados Usando Redes Neurais**. Dissertação de mestrado. UFRGS 2004

KIMBAL, R.; CASERTA, J.; **The Data Warehouse ETL Toolkit**. Wiley Publishing. 2004

LIQUIBASE; **Liquibase Reference Manual**. Disponível em: < http://www.liquibase.org/documentation/index.html>. Acesso em 13 de abr. 2014.

LIMA, A. E. N. **Pesquisa de Similaridade em XML**. Monografia (Graduação em Ciência da Computação) — Instituto de Informática, Porto Alegre, UFRGS, 2002.

MAMONE, M. **Migrating to iPhone and Ipad for .Net Developers.** Apress. 2011

MELLO, F. **Database Refactoring**. 2013 Disponível em: <http://fabriziomello.blogspot.com.br/2013/06/database-refactoring.html>. Acesso em: 15 de junho de 2013.

MICROSOFT. **Entity Framework Documentation**. 2014 Disponível em: <http://msdn.microsoft.com/en-us/data/ee712907.aspx#getstarted> Acesso em 17 mai. 2014.

MICROSOFT; **Expressões regulares do .NET Framework**. Disponível em: http://msdn.microsoft.com/pt-br/library/hs600312(v=vs.110).aspx. Acesso em 31 de dez. 2013.

MYSQL; **MySQL 5.0 Reference Manual**. Disponível em: <http://dev.mysql.com/doc/refman/5.0/en/table-size-limit.html>. Acesso em 28 de dez. 2013.

OLIVEIRA, P. J; RODRIGUES, F.; HENRIQUES, P. R. **Limpeza de Dados** – Uma Visão Geral. Universidade do Minho. 2004. Disponível em:

<http://wiki.di.uminho.pt/twiki/pub/Research/Doutoramentos/SDDI2004/ArtigoOliveira.pdf>. Acesso em 09 de jul. 2014.

OPDYKE, W. F. **Refactoring Object-Oriented** Frameworks. 1992. PhD Thesis – University of Illinois at Urbana Champaign, USA.

ORACLE. **Eventum:** Issue tracking system. 2009. Disponível em: <http://download.softagency.net/mysql/Downloads/eventum/> Acesso em 28 de jul. 2015.

PIPINO, L. L.; LEE, Y. W.; WANG, R. Y. **Data Quality Assessment**. Assocition for Computing Machinery. 2002

PIVETA, E. K. **Improving the search for refactoring Opportunities on Object-Oriented And Aspect-Oriented Software**. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul. 2009.

RAHM, E.; DO, H. H.. **Data Cleaning: Problems and Current Approaches**. IEEE Bulletin of the Technical Committee on DataEngineering, 24(4). 2000.

SAATY, T. **AHPFREE**: A free, web-based, collaborative implementation of the Analytic Hierarchy Process. 2007. Disponível em: <https://code.google.com/p/ahpfree> Acesso em 25 de abr. 2015.

STOREY, V. C.; DEY, D.; ULLRICH, H.; SHANKAR, S.; **An ontology-based expert system for database design**. Data & Knowledge Engineering n. 28, p. 31-46, 1998.

TÂN, N.; BÍNCH, T. **Application Of Database Refactoring For Improving Software Quality** Department Press H C and Technology. 2011 Disponível em:< http://ud.udn.vn/bankhcnmt/zipfiles/So45-quyen1/Microsoft%20Word%20-%209-nguyenphuongtam.pdf >. Acesso em 14 de abr. 2014.

RAMAN, V.; HELLERSTEIN, J. M. **Potter's Wheel:** An Interactive Data Cleaning System. Universidade de Berkeley, Califórnia –USA. 2002 . Disponível em: < http://control.cs.berkeley.edu/pwheel-vldb.pdf >. Acesso em 18 de set. 2014.

RED HAT. HIBERNATE – **Relational Persistence for Idiomatic Java**. 2004 Disponível em <http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ > Acesso em 17 de mai. 2014.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence**: A Modern Approach. 2nd ed. [S.l.]: Prentice Hall, 2002.

SAATY, T. L. L. **How to Make a Decision: the analytic hierarchy process**. European Journal of Operational Research, [S.l.], v.48, n.1, p.9 – 26, 1990.

SAATY, T. L. **Decision-Making With the AHP**: why is the principal eigenvector necessary? European Journal of Operational Research, [S.l.], v.145, n.1, p.85 – 91, 2003.

SKINNER, S. **RegExr v2.0**. 2008. Disponível em: <http://regexr.com>. Acesso em 20 de mar. 2015.

WALEK, B.; KLIMES, C. **A Tool for Database Testing and Optimization**. International Journal of Computer and Communication Engineering, Vol. 1, n. 3. 2012.

WORDNET; **Wordnet Documentation**. Disponível em: < http://wordnet.princeton.edu/wordnet/documentation/>. Acesso em 07 de jun. 2014.

VASARHELYI, M. A.; ISSA, H. Duplicate Records Detection Techniques: issues and illustration. 2010. Disponível em: <http://www.systemsthinking.nl/Paper%20Vasarhelyi%20Issa.pdf>. Acesso em 09 de out. 2013.