

UNIVERSIDADE FEDERAL DO RIO GRANDE  
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE  
SISTEMAS

LEANDRO SOUZA MARQUES

**Uma ferramenta para Refatoração em  
Bancos de Dados PostgreSQL**

Trabalho de Conclusão apresentado como  
requisito parcial para a obtenção do grau de  
Tecnólogo em Análise e Desenvolvimento de  
Sistemas

Prof. Msc. Igor Avila Pereira  
Orientador

Rio Grande, julho de 2016

## CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Marques, Leandro Souza

Uma ferramenta para Refatoração em Bancos de Dados PostgreSQL / Leandro Souza Marques. – Rio Grande: TADS/FURG, 2016.

78 f.: il.

Trabalho de Conclusão de Curso (tecnólogo) – Universidade Federal do Rio Grande. Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Rio Grande, BR-RS, 2016. Orientador: Igor Avila Pereira.

1. Refatoração. 2. Banco de dados. 3. Refatoração em banco de dados. I. Pereira, Igor Avila. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE

Reitora: Prof<sup>a</sup>. Cleuza Maria Sobral Dias

Pró-Reitora de Graduação: Prof. Denise Maria Varella Martinez

Coordenador do Curso: Prof. Rafael Betito

*O mundo não é um mar de rosas. É um lugar ruim e asqueroso... e não importa quão durão você é... ele te deixará de joelhos e te manterá assim se permitir. Nem você, nem eu, nem ninguém vai bater tão duro como a própria vida, mas não se trata de bater duro, se trata de quanto você aguenta apanhar e seguir em frente, o quanto você é capaz de aguentar e continuar tentando. É assim que a vitória é conquistada!*

— ROCKY BALBOA

## AGRADECIMENTOS

Aos meus pais, Gilberto e Denilda, pelo amor e pelo carinho durante a minha caminhada. A minha mãe, pela dedicação e paciência empenhados na minha educação, sem isso eu não teria chegado aqui. Ao meu pai, pelas batalhas travadas diariamente para manter a nossa família e por me inspirar por ser um exemplo de caráter, força e resistência.

A minha irmã, Liége, pelo amor, pelo carinho e por estar sempre disposta a me ouvir.

Aos meus amigos de "milianos", Duda, Abib, Manoel, William, Arthur, Pardo, Maurício e Henrique, pela amizade, pelas festas, pelos porres, pelas noites de Diabolo, pelos churrascos, pelos futebóis, pelos vôleis, pelos "rootismos" e por estarem juntos comigo a tanto tempo.

A meu grande amigo, Duda, por estar sempre disposto a me ajudar, a me ouvir e principalmente pela amizade de quase toda a minha vida.

Aos meus colegas de FURG, principalmente, aos diretores, Samuel, Azamba e Diogo, pelo apoio e pela paciência para com as minhas falhas no período do curso.

A minha namorada, Daniela, pelo carinho, pela paciência e por acreditar em mim.

A todos os professores do curso, pelo conhecimento compartilhado e pelo comprometimento para com os alunos. E principalmente ao meu amigo e orientador, Igor. Grande parceiro, que com muita boa vontade, conhecimento e profissionalismo me ajudou a desenvolver este trabalho.

Aos meus colegas de profissão, que se tornaram amigos, Fredo, Diego, Gabriel, Márcio, Bruno, Lucas, pelas indiadas, pelas festas, pelas madrugadas de trabalho e principalmente pelo conhecimento compartilhado em tantos desafios que enfrentamos juntos, que teriam sido muito mais penosos se não fossem eles.

A minha amigona do coração, Vanessa, pelo carinho, pela amizade e pelos conselhos que iluminam o meu caminho.

Aos amigos, que eu tive o prazer de conhecer neste curso, Peres, Andrei e Bruno, pela grande parceria nos desafios enfrentados em nessa caminhada, pelos lanches depois das aulas, pelo incentivo em momentos que fraquejei e pela amizade.

# SUMÁRIO

<b>LISTA DE ABREVIATURAS E SIGLAS</b>	7
<b>LISTA DE FIGURAS</b>	8
<b>LISTA DE TABELAS</b>	10
<b>RESUMO</b>	11
<b>ABSTRACT</b>	12
<b>1 INTRODUÇÃO</b>	13
1.1 Objetivos	14
1.2 Organização do Texto	14
<b>2 REVISÃO BIBLIOGRÁFICA</b>	16
2.1 Refatoração	16
2.1.1 Quando usar a refatoração	17
2.1.2 Quando não usar a refatoração	17
2.1.3 Refatoração e Projeto	18
2.1.4 <i>Code smells</i>	18
2.2 Refatoração em banco de dados	19
2.2.1 Refatoração de Banco Dados x Refatoração de Código	22
2.2.2 Importância da Refatoração em Banco de Dados	22
2.2.3 Database Smells	23
2.3 Categorias de Refatoração em Banco de Dados	25
2.3.1 Refatorações Estruturais	27
2.3.2 Refatorações de Qualidade de Dados	30
2.3.3 Refatorações de Integridade Referencial	33
2.3.4 Refatorações Arquiteturais	34
<b>3 ESTUDO DOS SISTEMAS EXISTENTES</b>	35
3.1 dbForge Studio for MySQL	35
3.2 SchemaCrawler	38
3.3 Liquibase	41
3.4 Resumo dos Sistemas Existentes	43

<b>4</b>	<b>ANÁLISE DE PROJETO</b>	<b>45</b>
4.1	Diagrama de Caso de Uso	45
4.2	Diagrama de Classes	46
4.3	Diagrama Relacional do Banco de Dados da Ferramenta	52
<b>5</b>	<b>IMPLEMENTAÇÃO</b>	<b>53</b>
5.1	Estrutura do PostgreSQL	53
5.2	Implementação das Classes	54
5.2.1	Banco Analisado	54
5.2.2	Maus Cheiros	55
5.2.3	Estrutura Interna da Ferramenta	60
<b>6</b>	<b>ESTUDO DE CASO</b>	<b>64</b>
<b>7</b>	<b>CONCLUSÃO</b>	<b>73</b>
<b>8</b>	<b>TRABALHOS FUTUROS</b>	<b>75</b>
	<b>REFERÊNCIAS</b>	<b>76</b>
	<b>FOLHA DE APROVAÇÃO</b>	<b>78</b>

## **LISTA DE ABREVIATURAS E SIGLAS**

TI	Tecnologia da Informação
TADS	Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas
JDBC	Java Database Connectivity
SGBD	Sistema de Gerenciamento de Banco de Dados
CSV	Comma-separated values
HTML5	Hypertext Markup Language, versão 5
JSON	JavaScript Object Notation
ANSI	American National Standards Institute
SQL	Structured Query Language
XML	eXtensible Markup Language

## LISTA DE FIGURAS

Figura 2.1:	Os dois cenários de acoplamento do banco de dados(AMBLER; SADALAGE, 2006). . . . .	20
Figura 2.2:	O esquema inicial do banco de dados antes da refatoração(AMBLER; SADALAGE, 2006). . . . .	20
Figura 2.3:	O esquema final do banco de dados após a refatoração(AMBLER; SADALAGE, 2006). . . . .	21
Figura 2.4:	O esquema de banco de dados durante o período de transição(AMBLER; SADALAGE, 2006). . . . .	22
Figura 3.1:	Opção de refatoração: Renomear Coluna . . . . .	35
Figura 3.2:	renomenando a coluna <i>id_estado</i> para <i>id_uf</i> e resolvendo as dependências(chave estrangeira na tabela <i>cms_cidade</i> e campo utilizado no <i>join</i> da <i>view cidde_estado</i> ). Exibindo o comando <i>SQL</i> para criar a tabela após o ajuste . . . . .	36
Figura 3.3:	renomenando a coluna <i>id_estado</i> para <i>id_uf</i> e resolvendo as dependências(chave estrangeira na tabela <i>cms_cidade</i> e campo utilizado no <i>join</i> da <i>view cidde_estado</i> ). Exibindo o <i>SELECT</i> da <i>view cidade_estado</i> após a mudança . . . . .	37
Figura 3.4:	renomenando a coluna <i>id_estado</i> para <i>id_uf</i> e resolvendo as dependências(chave estrangeira na tabela <i>cms_cidade</i> e campo utilizado no <i>join</i> da <i>view cidde_estado</i> ). Exibindo o comando <i>SQL</i> para criar a tabela <i>cms_cidade</i> , após a mudança, com a referência ao campo renomeado. . . . .	37
Figura 3.5:	Arquivo com as configurações dos <i>maus cheiros</i> que serão pesquisados pela ferramenta. . . . .	40
Figura 3.6:	Linha de comando para executar a ferramenta. . . . .	40
Figura 3.7:	Relatório obtido pela análise feita pela ferramenta em busca de <i>maus cheiros</i> . . . . .	41
Figura 3.8:	<i>changelog</i> com um <i>changeSet</i> para criação da tabela <i>cms_estado</i> . . .	42
Figura 3.9:	<i>ChangeSet</i> para fazer a alteração/refatoração <i>Rename Column</i> . . . .	43
Figura 3.10:	Comando <i>SQL</i> gerado pelo <i>ChangeSet</i> supracitado. . . . .	43
Figura 4.1:	Caso de uso . . . . .	45
Figura 4.2:	Diagrama de classes da estrutura interna da ferramenta . . . . .	47
Figura 4.3:	Diagrama de classes dos <i>maus cheiros</i> . . . . .	49
Figura 4.4:	Diagrama de classes da estrutura do banco que será analisado pela ferramenta . . . . .	50
Figura 4.5:	Diagrama relacional de banco de dados . . . . .	52



Figura 6.1:	banco que será analisado pela ferramenta . . . . .	64
Figura 6.2:	Página de cadastro do usuário . . . . .	66
Figura 6.3:	Página de login . . . . .	66
Figura 6.4:	Página de cadastro de um banco de dados . . . . .	67
Figura 6.5:	Página com a lista de bancos de dados do usuário . . . . .	68
Figura 6.6:	Página com a lista de maus cheiros que a ferramenta é capaz de analisar	68
Figura 6.7:	Resultado da procura pelo primeiro mau cheiro da lista . . . . .	69
Figura 6.8:	Resultado obtido quando a SQL é executada com sucesso . . . . .	69
Figura 6.9:	Resultado obtido quando a SQL falha . . . . .	69
Figura 6.10:	Lista de log . . . . .	70
Figura 6.11:	Detalhe de um log . . . . .	70
Figura 6.12:	Lista dos <i>maus cheiros</i> cadastrados pelo usuário . . . . .	71
Figura 6.13:	Formulário para cadastro de um novo <i>mau cheiro</i> . . . . .	72

## LISTA DE TABELAS

Tabela 2.1:	Categorias de refatoração de banco de dados . . . . .	26
Tabela 3.1:	Tabela de comparação entre os sistemas existentes . . . . .	44
Tabela 7.1:	Tabela de comparação entre os sistemas existentes e a ferramenta desenvolvida no presente trabalho . . . . .	74

## RESUMO

O mercado atual está em constante mudança, sendo assim as empresas necessitam adaptar-se a essa condição, necessitando responder as novas oportunidades e ao surgimento de novos produtos e serviços concorrentes de forma mais rápida e eficaz (SOMMERVILLE, 2011). As metodologias ágeis são soluções mais adequadas ao desenvolvimento de aplicativos nos quais os requisitos do sistema mudam rapidamente durante o processo de desenvolvimento. Uma das técnicas utilizadas pelas metodologias ágeis é a refatoração que consiste em alterar a estrutura interna do software porém sem alterar seu comportamento observável, com objetivo de torná-lo mais legível e facilitar futuras alterações (BECK; FOWLER, 1999). Embora desenvolvedores tenham adotado as metodologias ágeis, a comunidade de dados perdeu toda a revolução do desenvolvimento ágil de software, inclusive as técnicas de refatoração. No entanto, assim como o código, a base de dados deve também acompanhar as mudanças de requisitos que acontecem no decorrer da construção do software (AMBLER; SADALAGE, 2006). A escassez e falta de maturidade das ferramentas que dão o devido suporte as técnicas de refatoração é um dos impedimentos à adoção de metodologias ágeis pelos administradores de banco de dados. Levando isso em consideração, bem como a relevância destas técnicas e a falta de trabalhos acadêmicos neste nicho. O presente trabalho faz um *overview* da refatoração em banco de dados com o objetivo de gerar um material acadêmico com foco nessa área da engenharia de software pouco explorada. Como resultado final do trabalho foi desenvolvida uma ferramenta capaz de auxiliar os administradores de banco de dados no processo de refatoração. Essa aplicação realiza uma análise em uma base de dados à procura de pontos que indiquem uma refatoração em potencial. Caso a ferramenta encontre tais pontos, exibirá as refatorações necessárias, bem como as alterações exigidas para resolver automaticamente as dependências relacionadas às refatorações. O usuário poderá então, optar por executar as alterações sugeridas pela ferramenta.

**Palavras-chave:** Refatoração, banco de dados, refatoração em banco de dados.

## **A tool for Refactoring PostgreSQL Database**

### **ABSTRACT**

Nowadays the market keeps constantly changing, following this trend enterprises must adapt to this condition, finding answers to new situations in an effective and agile way(SOMMERVILLE, 2011). Agile Methods appear to be the most fitting solution when developing applications that face constant system requirement revisions. One technique that Agile Methods advocate is refactoring, this process reinforces that the developer must rearrange the internal structure of the software, while avoiding interfering with its observable behavior. Aiming to create a more readable code that is developer friendly in case future alterations are needed(BECK; FOWLER, 1999). Even though developers have adopted the use agile methods database community lost the agile method revolution, even the application of refactoring is not a common practice nowadays. Just like the code, database must follow the new trends in requirements that happen during software construction(AMBLER; SADALAGE, 2006). Lack of maturity and scarce tools that support refactoring techniques are an impediment to a more general adoption of agile methods by database administrators. Taking this in account, as well as the noted relevance of these techniques and a lack of academic attention to this niche. This an overview on database refactoring that intends to create academic material that focuses on this field that lacks material. As a final result a tool will be developed to help database administrators to refactor their codes. This application analyzes the database code searching for segments that potentially need refactoring. In case such problems are found it will warn the developer of it and, if possible, automatically correct the problems found. The user can then choose if he agrees with the changes prompted.

**Keywords:** database, refactoring, refactoring database.

# 1 INTRODUÇÃO

O mercado atual está em constante mudança, sendo assim, as empresas necessitam adaptar-se a essa condição, necessitando responder as novas oportunidades e ao surgimento de produtos e serviços concorrentes de forma rápida e eficaz.

Com esta mudança constante, os processos de desenvolvimento de software tradicionais não são uma boa metodologia em razão de serem demorados devido a seu caráter preditivo, prescritivo, sequencial, burocrático, rigoroso, orientado a processos e dados, formais e controlado, que tem o sucesso alcançado desde que esteja em conformidade com o que foi projetado inicialmente. Porém, nesse cenário atual torna-se praticamente impossível de obter um conjunto completo de requisitos de software, os requisitos iniciais inevitavelmente serão alterados para que o sistema atenda as novas necessidades. Nesse caso as metodologias adotadas devem minimizar o impacto das mudanças (SOMMERVILLE, 2011).

As metodologias ágeis são soluções mais adequadas ao desenvolvimento de aplicativos nos quais os requisitos do sistema mudam rapidamente durante o processo de desenvolvimento. O software não é produzido como uma única unidade, mas como uma série de incrementos, onde cada incremento contém um conjunto de funcionalidades do sistema. Assim, os usuários e outros *stakeholders*<sup>1</sup> participam de cada incremento, podendo propor alterações e novos requisitos que devem ser implementados para a próxima versão do sistema, o que minimiza o impacto das mudanças (SOMMERVILLE, 2011).

Uma das técnicas utilizadas pelas metodologias ágeis é a **refatoração**, que consiste em alterar a estrutura interna do software porém sem alterar seu comportamento observável, com objetivo de torná-lo mais legível e facilitar futuras alterações (BECK; FOWLER, 1999). A refatoração é essencial no processo evolutivo, uma vez que ela possibilita a reestruturação constante do código de forma segura, afim de adaptar o projeto existente para atender as mudanças de requisitos sugeridas ou percebidas pelos *stakeholders* (BECK; FOWLER, 1999). Embora desenvolvedores adotem as metodologias ágeis, a comunidade de dados perdeu toda a revolução desenvolvimento ágil de software, inclusive as técnicas de refatoração.

Segundo Sachin Rekhi, o mundo de desenvolvimento está dividido em desenvolvedores orientados a objetos, adeptos das metodologias ágeis e profissionais que trabalham com banco de dados, apreciadores das metodologias clássicas, baseadas em processos burocráticos, rigorosos, formais e controlados. Porém, assim como o código, a base de dados deve acompanhar as mudanças de requisitos que acontecem no decorrer da construção do software (AMBLER; SADALAGE, 2006). No entanto, há três grandes dificuldades na

---

<sup>1</sup>Definição encontrada em <http://pmstudycircle.com/2012/03/stakeholders-in-project-management-definition-and-types/>

implementação dos processos ágeis quando o assunto é banco de dados: os impedimentos culturais, a curva de aprendizado do processo e a escassez de ferramentas que auxiliem os desenvolvedores. Embora exista essa barreira inicial a ser superada pelas equipes de desenvolvimento, as vantagens compensam essa dificuldade. Como por exemplo, minimizar o desperdício de código, evitar retrabalho, manter a familiaridade com o banco de dados, trabalhar de maneira compatível com os desenvolvedores de código e reduzir o esforço global.

Segundo Amber (2006) a refatoração de banco de dados é uma técnica primária para desenvolvedores ágeis. Uma vez que ela apresenta vantagens quando comparada com as técnicas clássicas, onde há um investimento de requisitos detalhados no início do projeto, um trabalho pesado na arquitetura e na concepção de artefatos de design. Essas vantagens são inerentes ao processo evolucionário de desenvolvimento de bancos de dados.

Além das vantagens inerentes ao desenvolvimento evolucionário, a refatoração apresenta outras qualidades. Estas refletem no código do banco de dados, fazendo com que se mantenha a integridade referencial, um design mais legível e de manutenção simplificada, uma performance otimizada, uma estrutura menos acoplada e mais consistente. Todas essas qualidades são implementáveis de maneira segura e confiável por conta do uso de técnicas pré-definidas pelo processo de refatoração (JÚNIOR, 2013).

É possível destacar que no processo de refatoração os ganhos são significativos quando aplicados em bancos de dados legados, uma vez que as mudanças são executadas em pequenos passos, de forma segura e controlada. Todavia, as técnicas utilizadas no processo são aplicadas segundo as estruturas definidas pelos modelos propostos pela refatoração (PRITCHARD, 2013).

## **1.1 Objetivos**

Levando em consideração a atual necessidade do mercado, a falta de material acadêmico com foco em refatoração em banco de dados, bem como a falta de ferramentas que auxiliem os desenvolvedores nesse processo. O presente trabalho explora essas técnicas com intuito de gerar um material acadêmico com foco nesse segmento pouco explorado.

Abrindo caminho para discussões sobre a implementação dos processos de refatoração em bancos de dados no âmbito acadêmico e empresarial, visto que há uma resistência histórica cultural à adoção desses processos.

Como resultado final desse estudo foi desenvolvida uma ferramenta capaz de auxiliar os administradores de banco de dados no processo de refatoração. Essa aplicação é capaz de analisar uma base de dados a procura de pontos que indiquem uma refatoração em potencial, caso ela encontre esses pontos, ela exibirá as refatorações necessárias, bem como as alterações para resolver automaticamente as dependências dessas refatorações.

## **1.2 Organização do Texto**

Este Trabalho de conclusão de curso, está dividido em oito capítulos. O capítulo 2, trata dos conceitos teóricos para o entendimento do trabalho, com intuito de expor uma visão geral da refatoração de código e de banco de dados, bem como expor seus benefícios, seus contrapontos e suas técnicas.

No capítulo 3, são analisados os poucos softwares existentes no mercado com algum suporte a refatoração. Realizando uma avaliação de suas funcionalidades, afim de entender os seus propósitos e diagnosticar as suas carências, para que a ferramenta de-

envolvida no presente trabalho explore essas carências, com intuito de preencher uma lacuna existente.

O capítulo 4 contém a análise feita para o desenvolvimento da ferramenta proposta no presente trabalho, por meio da documentação de engenharia do projeto. Essa documentação consiste em diagrama de caso de uso, diagrama de classes e diagrama relacional de banco de dados.

No capítulo 5, é descrita a implementação da ferramenta, desde a implementação das classes que refletem os metadados do banco que será analisado, até as classes dos maus cheiros contemplados pela ferramenta. Além disso, esse capítulo contém informações do desenvolvimento da estrutura interna da aplicação, bem como das tecnologias utilizadas para tal desenvolvimento.

O capítulo 6 contém um estudo de caso, onde são exemplificadas todas as funcionalidades da ferramenta, desde o cadastro do usuário do sistema até a refatoração de um mau cheiro.

O capítulo 7 contém as conclusões obtidas no desenvolvimento do presente trabalho, levando em consideração a carência de material teórico neste nicho e a falta de maturidade das ferramentas voltadas para a refatoração em banco de dados.

O capítulo 8 aborda os trabalhos futuros que podem ser desenvolvidos com base no presente trabalho, como por exemplo, o desenvolvimento de classes capaz de encontrar outros maus cheiros. Outro trabalho futuro, seria com relação a portabilidade da ferramenta, uma vez que ela é exclusiva para *PostgreSQL*

## 2 REVISÃO BIBLIOGRÁFICA

Este capítulo trata dos conceitos teóricos para o entendimento do trabalho, com intuito de expor uma visão geral da refatoração de código e de banco de dados, bem como expor seus benefícios, seus contrapontos e suas técnicas.

### 2.1 Refatoração

Refatorar um software é alterar a sua estrutura interna porém sem alterar seu comportamento observável com objetivo de torná-lo mais legível e facilitar futuras alterações. A refatoração não é apenas uma limpeza de código, mas uma técnica para limpar o código de maneira mais eficiente e controlada, uma vez que o desenvolvedor domine as técnicas de refatoração saberá quais e como usá-las minimizando assim as falhas (KATAOKA et al., 2001).

Como o objetivo da refatoração é tornar o código mais fácil de entender, mudanças que visam melhorar o desempenho do software na maioria das vezes não são consideradas refatorações, embora assim como as refatorações, as mudanças de código para esta finalidade também não alteram o comportamento observável (BECK; FOWLER, 1999). A refatoração ajuda o desenvolvedor a entender um código que não está familiarizado, uma vez que, para refatorar o profissional trabalha na compreensão do código existente.

O primeiro passo da refatoração é entender o que o código faz, para isso, o programador faz uma análise buscando entender o seu propósito. Caso o entendimento do código seja prejudicado pela complexidade do fonte analisado, o programador então deve refatorá-lo para que este reflita melhor o seu comportamento (BECK; FOWLER, 1999). Além disso, ao refatorar o desenvolvedor torna o código mais legível, o que muitas vezes ajuda a achar falhas, já que o entendimento do código fica mais claro (TORRES, 2016).

Tornar o código mais fácil de ser entendido é essencial para futuras manutenções no software. É muito importante que o código comunique o seu propósito de forma clara, para que o programador não gaste, por exemplo, uma semana para fazer uma alteração que poderia ser feita em uma hora, caso o código estivesse mais legível (CASTRO, 2007).

A refatoração sistemática ajuda a preservar a estrutura do software, visto que a desestruturação do código é cumulativa, ou seja, quanto mais difícil é entender o projeto por meio do código, mais difícil será preservá-lo e mais rápido o código se desestruturará (TORRES, 2016).

A refatoração pode não parecer uma boa ideia quando o assunto é tempo de desenvolvimento de software, já que com esta técnica os profissionais acabam despendendo determinado tempo para fazer alterações em códigos já homologados. Todavia, um código bem fatorado melhora a sua qualidade, a sua legibilidade e diminui a presença de falhas. Isso faz com que se tenha um bom projeto, o que facilita as mudanças futuras e



consequentemente diminui o tempo gasto nessas modificações. O ganho quanto ao tempo de desenvolvimento é notável, porém é mais perceptível a longo prazo (SAAB, 2011).

### **2.1.1 Quando usar a refatoração**

Segundo Fowler, a refatoração deve ser uma constante no projeto, ou seja, não se deve estabelecer um período de refatoração pré-definido em cronograma. A equipe deve ter liberdade de utilizar essa técnica quando achar necessário. No entanto, existem dois momentos mais apropriados para a aplicação dessa técnica: a adição de uma nova funcionalidade e a revisão de código (BECK; FOWLER, 1999).

O momento mais apropriado para refatorar é quando desenvolvedor irá adicionar alguma funcionalidade, pois como já foi dito, o processo de refatoração ajuda a entender o código legado, o que é muito importante na adição de novas funcionalidades.

O desenvolvedor deve refatorar quando analisar o código e verificar se existem modificações que podem facilitar a adição dessa e de outras novas funcionalidades. Muitas vezes dar um passo atrás para refatorar algo já existente antes de adicionar a nova funcionalidade pode parecer mais perda de tempo, mas a longo prazo, isso fará o desenvolvedor trabalhar mais rapidamente com o código (BECK; FOWLER, 1999).

A revisão de fonte é uma prática adotada em muitas organizações, visto que ela ajuda a difundir o conhecimento através da equipe, além de ser muito importante na escrita do código claro e para o entendimento de mais aspectos de um sistema grande de software.

As revisões também possibilitam que outras pessoas contribuam com ideias. Ao invés de apenas sugerir tais modificações, o revisor pode imediatamente implementá-las, ou seja, o revisor não precisa imaginar como ficaria com tais implementações. Isso faz com que as revisões de código gerem um resultado mais concreto (SILVA, 2015).

Kent Beck (2006) cita quatro aspectos que tornam os programas difíceis de trabalhar:

- Programas que são difíceis de ler, são difíceis de modificar.
- Programas que tem lógica duplicada, são difíceis de modificar.
- Programas que, para inclusão de novas funcionalidades, requerem a alteração de código existente, são difíceis de modificar.
- Programas com lógicas condicionais complexas são difíceis de modificar..

Segundo Beck, esses problemas com relação a estrutura dos códigos acontecem na maioria das vezes porque o desenvolvedor que está adicionando ou alterando uma funcionalidade, está focando no que ele quer que o programa faça hoje, sem se preocupar com o amanhã. No entanto, é difícil prever o que poderá ser feito futuramente. A refatoração é uma maneira de sair dessa situação, quando se descobre que a decisão de ontem não faz mais sentido hoje, o desenvolvedor muda essa decisão e faz o trabalho de hoje (BECK; FOWLER, 1999).

### **2.1.2 Quando não usar a refatoração**

Há momentos em que não se deve refatorar, como por exemplo, quando o código está muito confuso e se torna mais fácil reescrevê-lo do zero do que refatorar. Um outro caso é quando o código não funciona ou quando ele está tão cheio de erros que o desenvolvedor não consegue estabilizá-lo.

Considerando que na maioria das vezes o ganho da refatoração é a longo prazo, já que ao refatorar ganhasse produtividade nas futuras alterações de código, um momento onde não se deve refatorar é próximo de um prazo final (BECK; FOWLER, 1999).

### 2.1.3 Refatoração e Projeto

Nas metodologias clássicas, as orientadas a planejamento, na fase de projeto espera-se que seja encontrada a versão final do projeto, enquanto a programação seria apenas mecânica, onde o desenvolvedor seguiria o que foi determinado na fase de projeto. Fazendo uma analogia, o projeto seria uma planta de engenharia, enquanto o código seria o trabalho de construção. No entanto, um software é muito mais flexível que uma construção física, pois mudar constantemente a estrutura de códigos é viável, já mudar constantemente a estrutura de prédios é inviável. Desta forma a pressão na fase de projeto é muito grande, pois espera-se que seja encontrada a versão final do software nesta etapa. Gasta-se muito mais tempo e trabalho nesta fase, afim de evitar ao máximo as alterações. Nesse caso, alterações podem ser muito custosas, uma vez que o projeto já está finalizado e as alterações podem afetar muito essa estrutura.

Com a refatoração a ênfase do projeto muda, pois agora o objetivo é conseguir uma solução razoável, para que seja possível começar a construir a aplicação. À medida que isso acontece, o desenvolvedor conhece melhor o problema e acaba percebendo que a solução inicial não é a melhor solução. Agora alterações não são tão custosas, visto que o desenvolvedor tem mais liberdade, uma vez que não serão feitas alterações no projeto final estruturado.

Com a refatoração, o projeto caminha na direção da simplicidade, uma vez que, o desenvolvedor tem consciência que irá refatorar, não necessita fazer na primeira vez a solução final.

Afim de evitar maiores transtornos quando há necessidade de mudanças no projeto, usando as metodologias orientadas a planejamento se desenvolve uma solução flexível que possa atender à necessidade atual e futuras alterações dessas necessidades. No entanto soluções flexíveis costumam ser mais complexas, além disso, essa flexibilidade pode ser inútil, pois nada garante que as mudanças acontecerão de fato.

Com a refatoração, a abordagem de alterações é diferente, o desenvolvedor tem que analisar a dificuldade de alterar a solução simples e a flexível, na maioria das vezes é muito mais fácil alterar a solução simples. Nesse caso, o desenvolvedor implementa essa solução e isso leva a um projeto menos complexo e sem sacrificar a sua abrangência, o que torna o processo de projetar menos estressante (BECK; FOWLER, 1999).

### 2.1.4 Code smells

Em 1999, Fowler introduziu o conceito de *code smells*, que é uma indicação superficial que geralmente corresponde a problemas mais profundos no sistema. A tradução literal seria *cheiros do código*, porém como são pontos que indicam possíveis problemas, o correto é utilizar a expressão *maus cheiros do código* (JÚNIOR, 2013).

Um *mau cheiro*, por definição, é algo que é fácil de identificar, como por exemplo, um método muito longo. No entanto, um *mau cheiro* nem sempre representa um problema como, por exemplo, alguns métodos longos são simplesmente bons. Por isso, é uma indicação superficial de que pode haver um problema, nesse caso, é necessário olhar mais profundamente a fim de que se descubra um possível problema oculto (BECK; FOWLER, 1999).

*Maus cheiros* geralmente não são *bugs*, eles não são tecnicamente incorretos e não

impedem que o programa funcione. Em vez disso, eles indicam deficiências na concepção, que podem estar prejudicando o desenvolvimento ou aumentando o risco de erros e falhas no futuro.

Do ponto de vista de princípios e qualidade, *maus cheiros* são certas estruturas no código que indicam violações nos princípios fundamentais de design e impactam negativamente na sua qualidade. Alguns exemplos de *maus cheiros* são:

- **Códigos duplicados:** Códigos com a mesma funcionalidade (ou funcionalidades muito parecidas) e que podem ser encontrados em mais de um local.
- **Métodos longos:** Métodos, funções ou procedimentos que adquiriram tamanhos maiores do que o necessário, realizando outros comportamentos além do qual ele inicialmente deveria realizar.

## 2.2 Refatoração em banco de dados

A refatoração de banco de dados consiste em mudanças disciplinadas no esquema da base de dados, com o intuito de melhorar a sua concepção, mantendo o seu comportamento e suas informações (AMBLER; SADALAGE, 2006).

Ao refatorar, o profissional não pode adicionar uma nova funcionalidade ou excluir uma funcionalidade existente, nem pode adicionar novos dados ou alterar o significado dos dados existentes. As refatorações atuam tanto em aspectos estruturais como em aspectos funcionais da base de dados.

Refatorações de banco de dados são conceitualmente mais difíceis do que refatorações de códigos, uma vez que a refatoração de código só precisa manter a semântica comportamental, enquanto refatorações banco de dados também devem manter a semântica da informação (SOARES, 2010). Além disso, existe um acoplamento muito maior com essas estruturas, conforme pode ser visto na Figura 2.1.

### Ambiente de uma aplicação

### Ambiente Multi-Aplicação

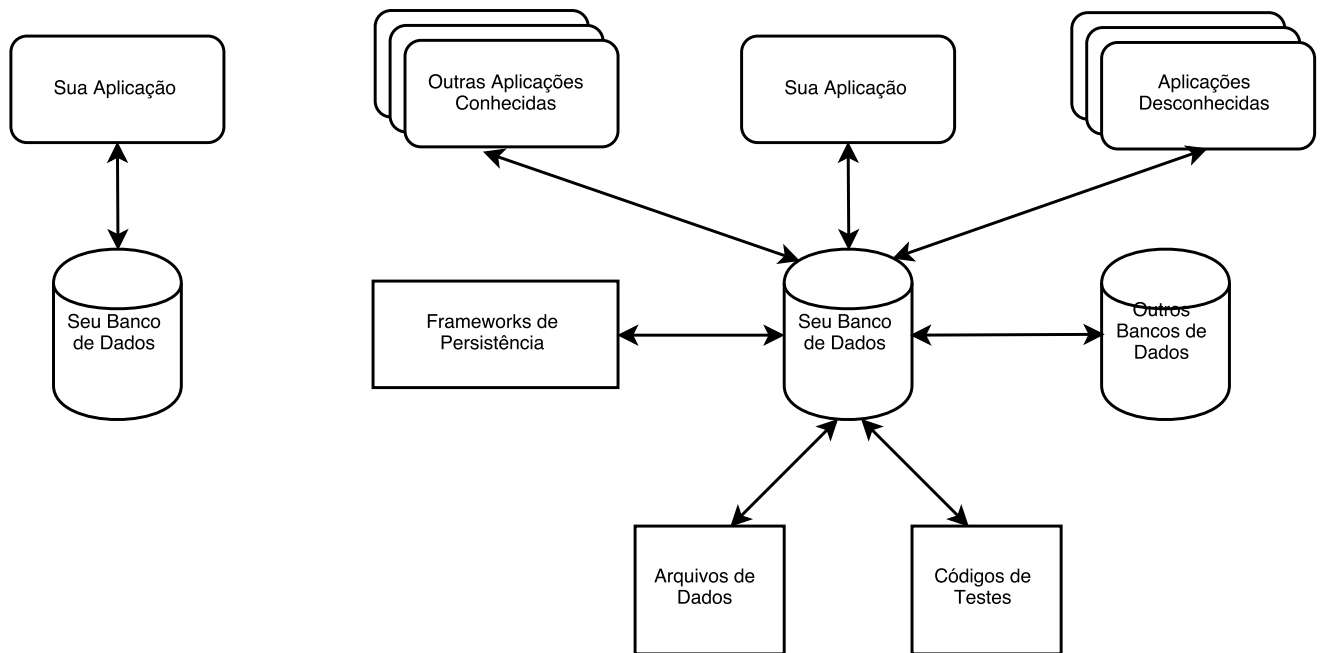


Figura 2.1: Os dois cenários de acoplamento do banco de dados(AMBLER; SADALAGE, 2006).

Com o intuito de elucidar os dois cenários supracitados, será usado como exemplo um banco de dados onde há uma tabela *pessoa* e uma tabela *aluno*, como pode ser visto na Figura 2.2, porém a tabela *pessoa* contém o campo *coeficiente* equivocadamente, pois este campo deveria fazer parte da tabela *aluno*, neste caso deve-se aplicar a refatoração *Mover Coluna*.

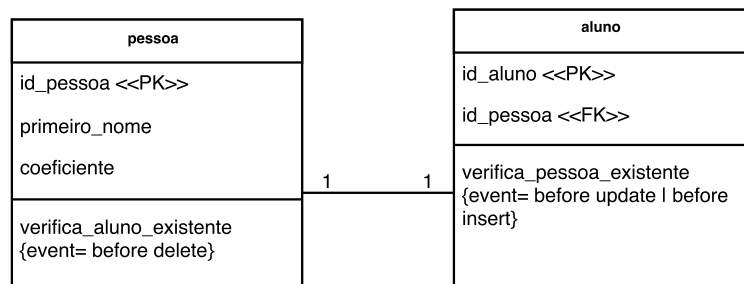


Figura 2.2: O esquema inicial do banco de dados antes da refatoração(AMBLER; SADALAGE, 2006).

O ambiente de uma aplicação, é o cenário mais simples que um administrador de banco de dados pode encontrar, uma vez que existe apenas uma aplicação vinculada a estrutura de dados a ser modificada. Neste caso os profissionais envolvidos têm acesso tanto ao banco de dados como ao código da aplicação que irá acessá-lo.

Considerando que apenas uma aplicação acessa os dados, serão poucos pontos a serem modificados para que a aplicação se adapte a nova estrutura, logo, nesse caso não há necessidade de estrutura e período de transição, visto que é mais simples sincronizar esta única aplicação com a base de dados (JÚNIOR, 2013).

Neste caso, o ideal é que duas pessoas trabalhem juntas em pares e que ambas tenham tanto habilidades de programação como de gerenciamento de banco de dados, ou então que cada uma domine ao menos um dos segmentos.

A equipe deve entrar em um consenso se há necessidade de refatorar o *schema* do banco de dados, em caso positivo a refatoração deve ser desenvolvida e testada dentro do *sandbox* do desenvolvedor. Feito isso, as mudanças (ilustrado na Figura 2.3) são publicadas e o sistema é testado e corrigido se necessário (AMBLER; SADALAGE, 2006).

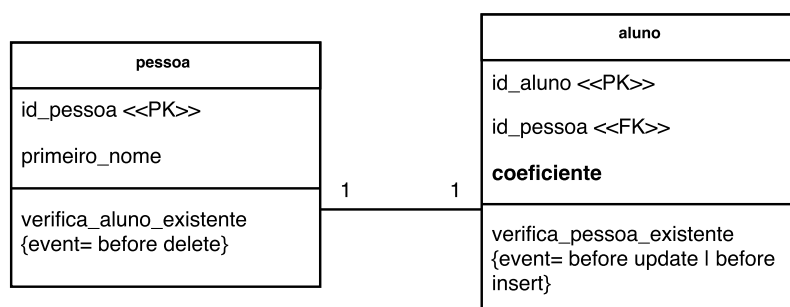


Figura 2.3: O esquema final do banco de dados após a refatoração(AMBLER; SADALAGE, 2006).

O ambiente multi-aplicação é um pouco mais complexo porque envolve a publicação de *releases* de diferentes aplicações em uma frequência de tempo variada. Nesse caso a principal diferença entre este ambiente e o ambiente com apenas uma aplicação é o fato de que haverá um período de depreciação. Nesse período, a coluna *coeficiente* da tabela *pessoa* deve ser mantida para que as aplicações dependentes desta estrutura não quebrem, até que as equipes de desenvolvimento atualizem e implantem as mudanças necessárias nestas aplicações (AMBLER; SADALAGE, 2006). Todavia, os dados dessas duas colunas devem estar sincronizados e para isso cria-se duas *triggers* *sincroniza\_coeficiente\_aluno* e *sincroniza\_coeficiente\_pessoa*, como pode ser visto na Figura 2.4. Logo após o período de transição deve-se remover a coluna da tabela *pessoa* e as *triggers*, obviamente é importante executar testes, afim de verificar se as aplicações não são mais dependentes da coluna removida, o resultado disso é representado na Figura 2.3.

Sendo assim, a ferramenta proposta no presente trabalho será desenvolvida para refatorar um banco de dados que se encontra em um ambiente de uma aplicação, já que ela não conterá a funcionalidade de depreciar a estrutura original.

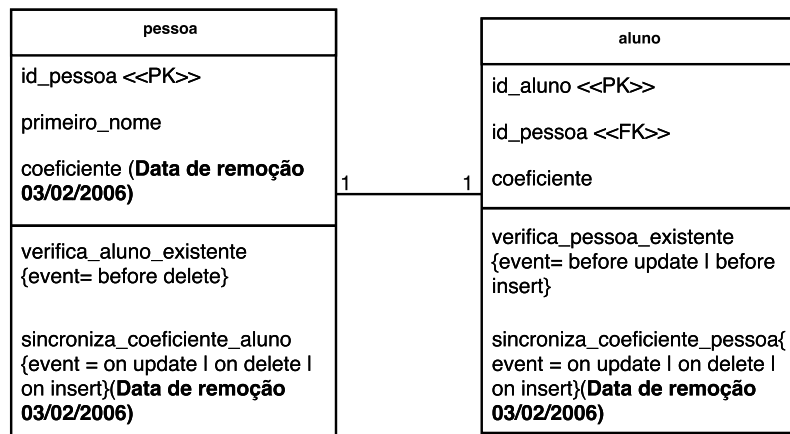


Figura 2.4: O esquema de banco de dados durante o período de transição (AMBLER; SADALAGE, 2006).

### 2.2.1 Refatoração de Banco Dados x Refatoração de Código

Conforme já foi citado, a diferença entre a refatoração em código e a em banco de dados é que essa segunda além da semântica comportamental deve manter a semântica das informações existentes na estrutura. Isso significa que após a refatoração os registros do banco de dados devem continuar fornecendo as mesmas informações que forneciam antes (DOMINGUES; ALMEIDA JR.; PALETTA, 2015).

Um exemplo de situação onde deve-se ter atenção com a semântica da informação seria um caso onde deve-se aplicar uma refatoração para mudar a formatação de um campo utilizado para guardar um número telefônico. Essa coluna originalmente armazena as informações assim (416) 555-1234 e 905.555.1212, mas deve armazenar apenas os números, como 4165551234 e 9055551212, nota-se que embora a informação tenha sido alterada ainda é possível mostrar os telefones no formato (XXX) XXX-XXXX.

### 2.2.2 Importância da Refatoração em Banco de Dados

É muito importante uma abordagem evolutiva durante desenvolvimento de banco de dados, uma vez que ela apresenta as seguintes vantagens:

- **Minimizar o desperdício:** Os requisitos são descobertos durante a construção do software, isso minimiza a chance de um requisito já desenvolvido ser considerado como não mais necessário.
- **Evitar retrabalho:** Os requisitos do projeto vão sendo construídos com a participação de todos *stakeholders*, inclusive dos clientes, a cada versão. Isso faz com que a equipe tenha um *feedback* dos clientes constantemente, o que reduz consideravelmente o retrabalho.
- **Garantir a funcionalidade do sistema:** As entregas são feitas em um curto período de tempo, fazendo com que o profissional produza regularmente o banco de dados, o que reduz drasticamente o risco do projeto.

- **Trabalhar de maneira compatível com os desenvolvedores de código:** Como os desenvolvedores estão adotando as metodologias evolucionárias de desenvolvimento, é importante que os administradores de banco de dados estejam em sincronia com esse processo.
- **Reduzir o esforço global:** Quando uma equipe trabalha de forma evolutiva ela necessita resolver o *Sprint*<sup>1</sup> atual, ou seja, as soluções desenvolvidas não necessitam ser flexíveis visando atender futuras alterações dos requisitos (AMBLER; SADALAGE, 2006).

Além das vantagens inerentes ao desenvolvimento evolucionário, a refatoração apresenta outras qualidades. Estas refletem no código do banco de dados, tais como: manter a integridade referencial, design mais legível e de manutenção simplificada, melhorar a performance, reduzir o acoplamento, aumentar a consistência, ajustar a normalização e manter a padronização do banco de dados. Todas essas qualidades são implementáveis de maneira segura e confiável por conta do uso de técnicas pré-definidas pelo processo de refatoração (JÚNIOR, 2013).

Considerando as vantagens citadas acima, é evidente que a refatoração pode ser vantajosa de forma significativa quando aplicada em bancos de dados legado, uma vez que a refatoração torna o banco mais compreensível. O próprio processo de refatoração faz com que o desenvolvedor tenha um conhecimento maior da estrutura do banco, desse modo é possível encontrar e corrigir problemas que até então passaram despercebidos, já que a estrutura não era clara. Consequentemente, com a melhora da legibilidade torna-se cada vez mais fácil fazer novas refatorações (DEVMEDIA, 2011).

### 2.2.3 Database Smells

Assim como já vimos, Fowler introduziu o conceito de um "code smell", uma categoria comum de possíveis problemas em seu código que indicam a necessidade de refatoração. Da mesma forma, existem *database smells*, que são indicações superficiais que geralmente correspondem a problemas mais profundos na estrutura da base de dados. Eles indicam a necessidade em potencial de aplicação de técnicas de refatoração em um banco de dados (AMBLER; SADALAGE, 2006).

Nessa seção são descritos alguns *database smells* citados por Amber, porém no capítulo de trabalhos relacionados, mais especificamente na ferramenta SchemaCrawler serão listados mais alguns maus cheiros encontrados pela ferramenta e descritos por Sualeh Fatehi (FATEHI, 2015).

Além dos maus cheiros descritos por Ambler e Fatehi, o autor do presente trabalho, introduziu, baseado nos conceitos estudados, o mau cheiro de colunas que são chaves estrangeiras mas que não possuem a devida restrição. Essas colunas seguem padrões de nomenclatura de chave estrangeira do mesmo tipo da potencial chave primária e armazenam dados correspondentes aos dados da potencial chave primária.

#### 2.2.3.1 Coluna Multiuso

Algumas vezes uma coluna é utilizada de forma equivocada para armazenar dados com propósitos diferentes. Para que seja possível trabalhar com essas colunas é necessário que sejam analisados outros dados pertencentes ao registro em questão, para que seja identificado o significado do dado armazenado nesta coluna.

<sup>1</sup> Definição encontrada em <http://searchsoftwarequality.techtarget.com/definition/Scrum-sprint>

Um exemplo seria um banco de dados ter uma tabela para armazenar as informações de uma pessoa, esta tabela teria um campo utilizado para armazenar a data de nascimento caso essa pessoa fosse um aluno e caso ela fosse um funcionário armazenar a data de contratação deste empregado. Isso geraria frustração quando for necessário armazenar ou consultar a data de aniversário do funcionário (AMBLER; SADALAGE, 2006).

#### 2.2.3.2 *Tabela Multiuso*

Em alguns casos uma tabela é utilizada para armazenar vários tipos de entidades, o que também torna necessário a verificação do tipo de entidade armazenada em cada registro, por meio de um ou mais campos da tabela. Além de que nesses casos há grandes chances de alguns campos serem nulos. Como essa tabela armazena mais de uma entidade ela será horizontalmente maior, o que pode comprometer o desempenho do banco de dados, nesses casos é provável que exista uma falha de projeto.

Um exemplo seria uma tabela utilizada para armazenar dados de pessoas físicas e pessoas jurídicas, onde existem campos que são pertencentes a pessoas e outros a empresas, como exemplo, o campo CPF e CNPJ, assim quando um registro for de uma empresa o campo CPF será nulo (AMBLER; SADALAGE, 2006).

#### 2.2.3.3 *Dados Redundantes*

A redundância de dados é um problema grave em estruturas de dados operacionais, uma vez que pode significar uma oportunidade para gerar inconsistências. Pois dados que deveriam ser armazenados em um campo só são guardados em mais de um campo e em tabelas diferentes (AMBLER; SADALAGE, 2006).

Um exemplo seria um banco de dados que armazena o endereço de alunos em lugares diferentes, já que durante um determinado tempo esses endereços eram armazenados na tabela aluno, no entanto foi feita uma alteração nessa estrutura e foi criada uma tabela para armazenar apenas os endereços. Por exemplo, em uma tabela consta que o Igor da Silva mora na Avenida Atlântica, Nº 123 e na outra tabela consta que ele mora na Rua 24 de Maio, Nº 456. Dessa forma as aplicações que consomem estas informações podem acabar utilizando o endereço errado.

#### 2.2.3.4 *Tabelas Com Muitas Colunas*

Tabelas com muitas colunas indicam que estão sendo armazenados dados de várias entidades. Por exemplo, se a tabela de alunos armazenar endereço residencial, comercial e de correspondência ou então armazenar vários números de telefones. Nesse caso será necessário normalizar a estrutura, adicionando uma tabela para endereços e uma para telefones (AMBLER; SADALAGE, 2006).

#### 2.2.3.5 *Tabelas Com Muitas Linhas*

Grandes tabelas são indicativos de problemas de desempenho. Por exemplo, o consumo de tempo gasto para fazer uma pesquisa em uma tabela com milhões de registros. Para melhorar isso pode-se mover algumas colunas para outra tabela ou então mover alguns registros para outra tabela. Ambas as estratégias de redução de tabela melhoram o desempenho (AMBLER; SADALAGE, 2006).



#### 2.2.3.6 *Colunas Inteligentes*

Uma coluna inteligente é aquela composta pela união de mais de uma informação, onde a posição dos dados determina onde começa e termina cada informação. Por exemplo, a coluna identificadora da tabela *aluno* onde os quatro primeiros dígitos representam o código da cidade natal dele seguido por um número serial, assim formando o ID do aluno (AMBLER; SADALAGE, 2006).

#### 2.2.3.7 *O Medo da Mudança*

O medo de mudança da estrutura do banco de dados é uma boa indicação de que existe um sério risco técnico em mãos, que só vai piorar com o tempo. Um sinal de que é necessário refatorar é quando a possível necessidade de realizar uma mudança na estrutura despertar medo no administrador de banco. Já que esse sentimento é proveniente da falta de clareza de tal estrutura e como o principal intuito da refatoração é tornar a estrutura mais clara deve-se então aplicá-la (AMBLER; SADALAGE, 2006).

### 2.3 **Categorias de Refatoração em Banco de Dados**

Com intuito de organizar os estudos nesta área, Ambler dividiu as técnicas de refatoração em seis categorias, tal como descrito na Tabela 2.1, adaptada de (AMBLER; SADALAGE, 2006). O presente trabalho pretende abordar apenas as refatorações que serão contempladas na aplicação desenvolvida pelo autor. Assim sendo, são listadas todas as refatorações, porém são descritas apenas algumas refatorações das categorias utilizadas por esta aplicação.

Tabela 2.1: Categorias de refatoração de banco de dados

Categoria de Refatoração	Descrição	Lista de Refatorações
Estrutural	Alteração nas definições de uma ou mais tabelas ou views.	Remover Coluna, Remover Tabela, Remover View, Introduzir Coluna Calculada, Introduzir Chave Primária Substituta, Mesclar Colunas, Mesclar Tabelas, Mover Coluna, Renomear Coluna, Renomear Tabela, Renomear View, Substituir LOB por Tabela, Substituição de Coluna, Substituir Relacionamentos de Um para Muitos por Tabelas Associativas, Substituir Uma Chave Auxiliar por Uma Chave Natural, Fragmentar Colunas e Fragmentar Tabelas
Qualidade de Dados	Alteração que aperfeiçoa a qualidade das informações contidas na base de dados.	Adicionar tabela de referência, Aplicar Padronização de Códigos, Aplicar Padronização de Tipos, Consolidar Estratégia de Chaves, Remover Chaves Estrangeiras e Restrições, Remover Valor Padrão, Modificar Colunas Definidas Como Não Nulas Para Nulas, Introduzir Restrições e Chaves Estrangeiras, Padronizar Formatos, Introduzir Valores Padrões, Tornar uma Coluna Não-nula, Mover Dados e Substituir Tipos de Códigos por Flags.
Integridade Referencial	Alteração que garante a consistência das referências entre as tabelas.	Introduzir restrições e chaves estrangeiras, Adicionar Trigger Para Coluna Calculada, Remover Chaves Estrangeiras e Restrições, Introduzir Exclusão em Cascata, Introduzir Exclusão Lógica, Introduzir Exclusão Física e Introduzir de Trigger Para Históricos.
Arquitetural	Alteração que melhora de maneira global a interação entre programas externos e o banco de dados.	Adicionar métodos CRUD, Adicionar Tabela Espelho, Adicionar Método de Leitura, Encapsular Tabela com View, Introduzir Método de Cálculo, Introduzir Índice, Introduzir Tabela Read-Only, Migrate Method From Database, Migrate Method To Database, Substituir Método por View, Substituir View por Método e Usar a Fonte de Dados Oficial.
Rotinas ou Métodos	Alteração em uma stored procedure, function ou trigger para aperfeiçoar sua qualidade.	Adicionar Parâmetros, Parametrizar Métodos, Remover Parâmetros, Reordenar Parâmetros, Substituir Parâmetro por Método Explícito, Consolidar Expressão Condicional, Decompor Condicional, Extrair Método, Introduzir Variável, Remover Flag de Controle, Remove Middle Man, Renomear Parâmetro, Substituir Literal por Tabela de Referência, Replace Nested Conditional with Guard Clauses, Dividir Variável Temporária e Substituir Algoritmo.
Transformações (Não é uma refatoração)	Alteração na estrutura da base de dados que influencia em sua semântica.	Inserir Dados, Introduzir Nova Coluna, Introduzir Nova Tabela, Introduzir Nova View e Atualizar Dados.

### 2.3.1 Refatorações Estruturais

As refatorações estruturais têm como objetivo melhorar o modelo do banco de dados. Mudanças de requisitos, má concepção do projeto de banco e desuso de determinadas estruturas são as principais causas para se alterar o design do modelo da base de dados (DEV MEDIA, 2012). Quando se realiza refatorações estruturais é necessário considerar vários problemas comuns em modificar a estrutura de dados, esses pontos de atenção são (AMBLER; SADALAGE, 2006):

- **Evitar ciclos gerados por *triggers*:** Quando for necessário criar *triggers* para manter a estrutura original e a nova sincronizadas no período de transição, deve-se tomar cuidado para que uma alteração não desencadeie um ciclo resultante em um *deadlock*. Para evitar isso, (SOARES, 2010) sugere que sejam feitas verificações antes de executar a ação no alvo em questão.
- **Corrigir *views* quebradas:** Como *views* são fortemente acopladas deve-se ter um cuidado com estas estruturas após uma mudança no design da base de dados, uma vez que elas podem ser dependentes da estrutura alterada, o que possivelmente resultará em quebra.
- **Corrigir *triggers* e *stored procedures* quebradas:** Assim como as *views* as *triggers* também são fortemente acopladas as definições das tabelas, por isso, também deve-se ter cuidado com essas estruturas quando for aplicada refatorações como renomear, mover ou retirar algum elemento da base de dados (coluna, tabela, *view*, etc.).
- **Corrigir tabelas quebradas:** Como as tabelas são indiretamente acopladas a colunas de outras tabelas, através de convenções de nomenclatura, deve-se ter cuidado com renomeação delas, para que seja mantido o padrão existente.
- **Definir o período de transição:** Como as refatorações estruturais na maioria das vezes terão um impacto nas aplicações que acessam a estrutura alterada, deve-se definir um período de transição para que essas aplicações sejam atualizadas para trabalhar com a nova estrutura.

#### 2.3.1.1 Remover Coluna

A principal razão para aplicar esta refatoração é a melhoria do design do banco de dados ou então quando a coluna deixa de ser utilizada por algum motivo. Ela também é utilizada como um passo de outras refatorações, como a *Mover Coluna*, uma vez que após criar a coluna na tabela correta deve-se removê-la da tabela original.

Com a remoção de uma coluna os seus dados serão perdidos, nesse caso, pode ser necessário armazená-los de alguma forma em outra coluna/tabela (JÚNIOR, 2013).

#### 2.3.1.2 Remover Tabela

Muitas vezes uma tabela deixa de ser necessária, já que ela pode ter sido substituída por outra tabela, por uma *view* ou então ela deixou de ser utilizada por uma fonte consumidora. Nesse caso, aplica-se essa refatoração, para liberar recurso do banco e para que essa tabela não seja usada equivocadamente.

Assim como na remoção de coluna, na remoção de uma tabela também deve-se levar em consideração que os seus dados serão perdidos. Nesse caso, pode ser necessário armazenar alguns dos dados em outra tabela, para que em caso de necessidade esses dados possam ser consumidos por uma *view*, por exemplo(AMBLER; SADALAGE, 2006).

#### 2.3.1.3 *Remover View*

Muitas vezes uma *view* deixa de ser necessária, já que ela pode ter sido substituída por outra *view*, por uma tabela ou então ela deixou de ser utilizada por uma fonte consumidora, nesse caso aplica-se essa refatoração.

A remoção de *views* não implica na exclusão de dados, entretanto, esta ação deve ser executada quando a mesma não é mais utilizada por algum programa.

#### 2.3.1.4 *Introduzir Coluna Calculada*

Muitas vezes as aplicações necessitam obter dados que são resultados de um cálculo envolvendo uma ou mais tabelas, no entanto, isso pode acabar comprometendo o desempenho desses programas. Essa refatoração tem como objetivo melhorar o desempenho dessas aplicações, uma vez que ela consiste na criação de uma coluna que armazenará o resultado do cálculo previamente calculado.

Deve-se manter essa coluna atualizada, por meio da sincronização entre ela e os dados envolvidos no cálculo. Para isso deve-se criar *triggers* que serão responsáveis por manter a coluna atualizada toda vez que houver mudanças nos dados utilizados pelo cálculo(AMBLER; SADALAGE, 2006).

#### 2.3.1.5 *Mesclar Colunas*

Há vários motivos pelos quais pode ser interessante mesclar duas ou mais colunas. Esses motivos podem ser:

- **Uma coluna idêntica a outra:** Uma coluna que apresenta a mesma semântica de outra existente, criada equivocadamente por falta de comunicação da equipe ou por falta de clareza da estrutura existente, nesse caso, o desenvolvedor pode acabar criando uma coluna com a mesma funcionalidade de uma existente.
- **Colunas que estão desnecessariamente fragmentadas:** Um exemplo, seria uma tabela onde o DDI, DDD e o telefone são armazenados separadamente, porém foi constatado que não há necessidade dessa separação e deve-se juntar todas essas informações em uma única coluna.

Esta refatoração pode resultar em uma perda de precisão dos dados, quando forem mescladas colunas ricas em detalhes, por isso deve haver muito cuidado no tratamento dessas informações. Em casos onde as colunas são usadas para finalidades diferentes pode ser que seja necessário manter alguma coluna em seu estado original(Júnior, 2013).

#### 2.3.1.6 *Mesclar Tabelas*

Há vários motivos pelos quais pode ser interessante mesclar duas ou mais tabelas. Esses motivos podem ser:

- **Tabelas que caem em desuso ou tornam-se desnecessárias em razão de novos projetos:** Um exemplo seria uma tabela aluno que contém, todos os dados dos alunos, inclusive dados de identificação dos mesmos. Porém com o passar do tempo foi

criada a tabela *dados\_identificacao\_aluno*, nesse caso pode ser interessante mesclar ambas tabelas.

- **Tabelas que acabam tendo o mesmo propósito de uso:** Com o passar do tempo, algumas tabelas podem passar a ser utilizadas para a mesma finalidade. Outro caso seriam tabelas que tem relação um-para-um sem necessidade, nesse caso é aconselhado que seja feita uma refatoração para mesclá-las.
- **Tabelas repetidas por engano:** Assim como acontece com as colunas, as tabelas podem ser criadas repetidamente por engano. Isso pode acontecer quando os desenvolvedores estão em equipes diferentes, ou quando os metadados que descrevem o esquema da tabela não estão disponíveis.

Esta refatoração pode resultar em uma perda de precisão dos dados quando forem mescladas tabelas ricas em detalhes, por isso deve haver muito cuidado no tratamento dessas informações. Em casos onde as tabelas são usadas para finalidades diferentes pode ser que seja necessário manter alguma tabela em seu estado original (AMBLER; SADALAGE, 2006).

#### 2.3.1.7 Mover Coluna

Essa ação consiste em migrar uma coluna de uma tabela, com todos os seus dados, para outra tabela existente. Há várias razões para aplicar esta refatoração. As duas primeiras razões podem parecer contraditórios, porém como já foi mencionado as refatorações são circunstanciais. A seguir algumas razões para se aplicar essa refatoração:

- **Normalizar:** Para aumentar a normalização da tabela de origem pode ser necessário a moção de uma coluna para outra tabela, assim, reduzindo a redundância de dados dentro do banco.
- **Reduzir normalização:** Embora este ponto aparentemente seja oposto do anterior, é comum descobrir que uma tabela é incluída em um *join* simplesmente para ter acesso a uma determinada coluna, o que pode acabar prejudicando o desempenho das consultas, com esta refatoração pode-se eliminar a necessidade desse *join*.
- **Reorganizar uma divisão de tabelas:** No projeto inicial uma tabela pode ter sido dividida em duas, ou então essa divisão pode ter ocorrido por meio da refatoração de divisão de tabelas. Em ambos os casos o desenvolvedor pode perceber que faltou mover alguma coluna e nesse caso aplicar esta refatoração.

Mover a coluna para aumentar a normalização pode reduzir o desempenho, uma vez que poderá fazer com que a quantidade de *joins* aumente, no entanto, reduzir a normalização irá fazer com que aumente a redundância dos dados (JÚNIOR, 2013).

#### 2.3.1.8 Renomear Coluna

Assim como o código, o banco de dados deve estar em conformidade com algumas convenções de nomenclatura adotadas pelas empresas. Nesse caso, essa refatoração é utilizada para aumentar a legibilidade do esquema de banco de dados, em conformidade com essas convenções.

Deve-se analisar o custo dessa refatoração, já que com essa alteração, *views*, *triggers*, *stored procedures* e as aplicações que referenciam essa coluna também devem ser alteradas para utilizar o novo nome (AMBLER; SADALAGE, 2006).

### 2.3.1.9 Renomear Tabela e Views

A principal razão para se aplicar essa refatoração é esclarecer o significado e a intenção da tabela/view no esquema de banco de dados ou para fazer com que estas estruturas estejam em conformidade com as convenções de nomenclatura de banco de dados aceitos pelas empresas.

Outro motivo para aplicar esta ação é permitir a portabilidade de banco de dados, uma vez que em um processo de migração de um banco para outro pode se ter problemas relacionados a nomes de tabelas que usam palavras-chaves reservadas.

Assim como a renomeação de coluna, deve-se avaliar o custo benefício dessa refatoração, já que poderá ser necessário alterar *views*, *triggers*, *stored procedures* e aplicações que fazem referência a tabela/view renomeada(JúNIOR, 2013).

### 2.3.1.10 Dividir Tabela

Essa refatoração consiste em dividir uma tabela, tanto horizontalmente, quanto verticalmente. Ou seja, transferir algumas colunas da tabela para outra tabela ou transferir os registros da tabela para outra.

É muito comum algumas tabelas conterem muitas colunas, entre essas colunas estão algumas que são o conjunto principal de dados e outras colunas que armazenam dados supérfluos.

Por exemplo, a tabela *pessoa* possui as colunas centrais *primeiro\_nome*, *data\_nascimento* e *CPF*, além desses campos, ela possui campos não essenciais como a coluna *imagem* entre outros. Como a coluna *imagem* armazena informações muito grandes e é utilizada por poucas aplicações, deve se considerar dividir a tabela. Isso ajudaria a melhorar o tempo de acesso as informações pelas aplicações que selecionam todas as colunas da tabela de *pessoa*(AMBLER; SADALAGE, 2006).

Algumas tabelas contém muitos registros, o que torna o acesso as informações contidas nela muito custosos. Sendo assim, é necessário particionar uma tabela verticalmente, o que consiste em dividir um grande volume de dados que seria armazenado em uma única tabela em partições (tabelas) menores, onde cada uma delas atende a um conjunto específico de dados definido a partir de critérios/regras bem definidas para divisão(TELLES, 2013).

## 2.3.2 Refatorações de Qualidade de Dados

Assim como seu nome já diz, as técnicas de refatoração contidas nessa categoria são responsáveis por mudanças que melhoram e asseguram a coerência das informações contidas dentro de um banco de dados.

Porém ao realizar refatorações que melhoram a qualidade dos dados é necessário considerar vários problemas comuns em fazer este tipo de modificação, esses pontos de atenção são:

- **Corrigir restrições quebradas:** Em casos de alteração de informação em colunas que contém restrições, pode ser que seja necessário remover essas restrições para aplicar a refatoração nos dados. Nesse caso, logo depois deve-se adicionar novamente as restrições.
- **Corrigir views quebradas:** Muitas vezes *views* usam dados fixos em suas cláusulas *where*, em caso de alteração desses dados a *view* pode quebrar ou passar a não retornar as informações desejadas.

- **Bloquear a atualização dos dados:** No processo de refatoração pode ser que seja necessário bloquear a atualização dos dados por meio das aplicações, ou seja, o banco ficará bloqueado para inserção, remoção e atualização das informações contidas nele.

### 2.3.2.1 Aplicar Padronização de Tipos

É muito importante para a legibilidade do banco que os tipos de dados das colunas estejam em conformidade com os padrões da empresa, de forma que sejam consistentes com os tipos de dados de outras colunas semelhantes dentro do mesmo banco de dados. Com base nisso, temos quatro principais motivos para se aplicar esta refatoração:

- **Garantir a integridade referencial:** Caso seja necessário adicionar uma chave estrangeira em todas as tabelas que armazenam determinada informação será necessário padronizar os tipos de dados das colunas individuais, ou seja, a chave primária e as chaves estrangeiras devem ter o tipo de dado padronizado.
- **Estar em conformidade com os padrões da empresa:** Muitas vezes ao explorar a base de dados descobre-se que algumas colunas não estão em conformidade com os padrões de modelagem de dados adotados pelas equipes de desenvolvimento.
- **Reduzir a complexidade do banco e do código:** Muitas vezes falta de padronização dos tipos de dados do banco reflete na padronização do código das aplicações, já que embora os dados sejam de natureza semelhantes ou iguais, estão armazenados com tipos distintos, por consequência, as aplicações poderão ter que tratar essas informações de forma distinta.

Muitas vezes as colunas a serem alteradas são utilizadas em muitos lugares, como em *views*, *triggers*, *stored procedures* e aplicações. Nesse caso, a alteração no tipo de dado dessas colunas pode não compensar.

Outro problema que o desenvolvedor pode se deparar quando for aplicar esta refatoração é quando a coluna a ser refatorada contém dados que não podem ser convertidos para tipo de dados que o campo deve assumir (AMBLER; SADALAGE, 2006).

### 2.3.2.2 Tornar Uma Coluna Não-nula

Há três principais motivos para se alterar uma coluna que pode assumir o valor nulo para que ela não possa mais assumir este valor:

- **Manter a consistência:** A primeira razão para se aplicar esta refatoração é manter a consistência do banco de dados, já que em algumas bases de dados, mal projetadas, colunas que desempenham um papel importante apresentam equivocadamente a possibilidade de assumir o valor nulo. Como é o caso de colunas que fazem parte do índice de uma tabela, ou então, tabelas onde todas as colunas podem assumir o valor nulo.
- **Garantir a inserção:** Obrigar a alimentação dos valores de determinados campos perante as aplicações.
- **Validação:** Outro motivo para se aplicar esta ação é fazer a checagem, em nível de banco de dados, para verificar se o campo é nulo.

Uma vez que essa ação for aplicada em uma coluna, caso o banco de dados esteja em um ambiente multi-aplicação deve-se atribuir um valor padrão para este campo, caso contrário, as aplicações que trabalham com as informações desta tabela poderão quebrar (AMBLER; SADALAGE, 2006).

#### 2.3.2.3 Adicionar Tabela de Referência

Algumas tabelas podem conter colunas que armazenam dados sistêmicos (enumeradores). Esses dados normalmente são códigos, que categorizam e/ou criam conjunto de registros sobre um determinado contexto. Por exemplo, uma coluna chamada *uf*, que contém o código do estado (RS, SP, RJ, etc.).

Essa modificação consiste na criação de uma tabela que armazena esses valores e consequentemente faz as devidas referências. Os motivos que levam o desenvolvedor a fazer esta refatoração são:

- **Introduzir a integridade referencial:** Introduzir a integridade referencial para garantir a qualidade dos dados, de modo que o campo aceite apenas um conjunto específico de dados.
- **Exibir as possibilidades:** Em casos que seja necessário listar os possíveis valores que o campo pode assumir.
- **Simplificar as restrições:** Quando há restrições de possíveis valores que podem ser utilizados. A adição de novos valores a essas restrições é mais complexa e intrusiva do que adicionar um registro em uma tabela de referência.
- **Possibilitar o detalhamento da informação:** Possibilitar o armazenamento de informações descritivas, mais detalhadas sobre os códigos.

Existem dois principais pontos de atenção. Um deles, é que a coluna deve apresentar dados válidos para preencher a tabela de pesquisa. Nesse caso, é importante que os dados estejam padronizados, ou então, que seja possível aplicar uma padronização neles. A outra questão é referente ao desempenho, uma vez que a adição de uma chave estrangeira aumenta o número de *joins* (JÚNIOR, 2013).

#### 2.3.2.4 Consolidar Estratégia de Chaves

Esta refatoração consiste em escolher uma estratégia para as chaves do banco de dados. Muitas entidades do banco de dados podem ter mais de um campo que representam potenciais chaves primárias, sendo assim é interessante definir um padrão para os campos que serão chaves primárias.

Com chaves primárias bem definidas o banco de dados terá um desempenho melhor, uma vez que, para cada chave primária há um índice correspondente. Além disso, muitas organizações seguem convenções próprias para o desenvolvimento do banco de dados, nesse caso é essencial que todas as tabelas sigam essas diretrizes.

Aplicar essa refatoração requer uma análise da base de dados, afim de verificar o custo dessa refatoração, uma vez que, para modificar a chave primária de uma tabela é necessário alterar também as chaves estrangeiras que acessam as chaves primárias que serão modificadas (AMBLER; SADALAGE, 2006).



### 2.3.3 Refatorações de Integridade Referencial

As refatorações de integridade referencial consistem na adição e remoção de restrições e regras de relacionamento entre as tabelas e seus campos, com o objetivo de garantir a consistência das referências entre as tabelas.

#### 2.3.3.1 Adicionar Foreign Key

Chave estrangeira (*foreign key*) é uma restrição adicionada a um campo, que estabelece o relacionamento entre duas tabelas. Nesse caso, essa refatoração consiste em adicionar essa chave estrangeira a uma tabela existente para impor uma relação com outra tabela. Os motivos que levam o desenvolvedor a fazer está refatoração são:

- **Garantir a integridade referencial:** Assim, as ações como a exclusão de registros na tabela que contém a chave primária serão controladas pelo banco de dados, de modo que impedirá a exclusão ou excluirá todos os registros dependentes da mesma.
- **Manter a consistência:** Garantir que apenas dados válidos sejam inseridos no campo que contém a restrição, pois a adição dessa restrição impede que dados inválidos sejam persistidos.

Conforme já foi citado, essas restrições prejudicam o desempenho do banco de dados.

Outra questão que deve ser verificada é com relação aos dados contidos na coluna onde será aplicada essa refatoração, uma vez que ela não pode conter dados diferentes dos contidos na coluna referenciada.

As aplicações podem acabar quebrando em alguns pontos, uma vez que com adição dessa restrição deverá ser considerada a ordem de operações como inserções, atualizações e exclusões(JÚNIOR, 2013).

#### 2.3.3.2 Introduzir Excluir em Cascata

Essa refatoração consiste em indicar para o banco que ao excluir um registro de uma tabela, ele deve deletar automaticamente os registros que fazem referência ao registro excluído. Os motivos que levam o desenvolvedor a fazer está refatoração são:

- **Manter a consistência:** Manter a integridade referencial, garantindo que os registros filhos não fiquem órfãos.
- **Utilizada por outra refatoração:** Essa refatoração muitas vezes é utilizada como parte da refatoração citada logo acima (Adicionar *Foreign Key*).

Ao utilizar essa refatoração deve-se evitar dependências cíclicas, onde dois ou mais processos ficam impedidos de continuar suas execuções, ou seja, ficam bloqueados, esperando uns pelos outros.

Outra questão a ser considerada é referente as aplicações. O desenvolvedor deverá verificar as aplicações que fazem exclusão de registros da tabela pai, afim de garantir que não há nem um código equivocado, já que com está refatoração uma exclusão dessas, repercutirá em todos os registros filhos(AMBLER; SADALAGE, 2006).

### 2.3.4 Refatorações Arquiteturais

Refatorações arquiteturais são mudanças que melhoram de maneira global a interação entre programas externos e o banco de dados. Os objetivos secundários são organizar, padronizar e melhorar o desempenho do banco de dados.

#### 2.3.4.1 Introduzir Índice

Essa refatoração consiste em Introduzir um novo índice em uma tabela. A principal razão para se aplicar esta refatoração é para fins de otimização, permitindo uma localização mais rápida de um registro.

Os índices devem ser atualizados toda vez que há uma alteração na tabela, desse modo, fazer essa refatoração pode prejudicar o desempenho de ações como atualização, inserção e exclusão de registros nessa tabela.

Outro problema é quando há uma necessidade de inserir índice exclusivo, porém a coluna contém valores duplicados.

#### 2.3.4.2 Encapsular Tabela Com View

Essa refatoração consiste em encapsular as colunas de uma tabela em uma *view*. Os motivos que levam o desenvolvedor a fazer esta refatoração são:

- **Implementar uma fachada para uma tabela:** Diminuir o acoplamento de uma tabela que sofrerá alguma refatoração estrutural, como a Renomeação de Coluna, por exemplo. Com esta ação as aplicações irão acessar a *view* criada, que servirá de fachada para acesso aos dados da tabela. Dessa forma, a estrutura da tabela poderá ser alterada com mais segurança.
- **Controle de acesso:** Essa refatoração também é utilizada por questões de segurança, em casos onde há a necessidade de liberar acesso a tabelas que contenham dados confidenciais. Nesse caso, a *view* irá conter apenas as colunas que não contêm dados confidenciais.

Dependendo do tipo de acesso necessário pelas aplicações, essa refatoração poderá não contemplar todas as necessidades, já que banco de dados pode não suportar o mesmo nível de acesso em *views* como em tabelas. Por exemplo, em caso da necessidade de inserir algum registro, o banco deve suportar *views* atualizáveis.

## 3 ESTUDO DOS SISTEMAS EXISTENTES

Neste capítulo são analisados os poucos softwares existentes no mercado com algum suporte a refatoração em base de dados. Realizando uma avaliação de suas funcionalidades, afim de entender seus propósitos e diagnosticar as suas carências, a fim de que a ferramenta desenvolvida no presente trabalho explore essas carências e preencha uma lacuna existente.

### 3.1 dbForge Studio for MySQL

O dbForge Studio é um cliente MySQL para Windows, que disponibiliza aos desenvolvedores e administradores de MySQL as opções para criar/executar consultas, desenvolver/depurar rotinas e automatizar o gerenciamento de objetos de banco de dados. Esta ferramenta fornece utilitários para comparar e sincronizar bancos de dados.

Além das funcionalidades citadas acima, o dbForge Studio conta com uma ferramenta de refatoração, como pode ser visto na Figura 3.1.

A vantagem dessa ferramenta quando comparada com outros clientes de banco de dados é que ela é capaz de ajustar as dependências das refatorações disponíveis. No entanto, ela conta apenas com duas refatorações estruturais, a renomeação de tabela e de coluna (DBFORGE STUDIO FOR MYSQL, 2015).

Ao renomear uma coluna, o administrador sempre precisa se preocupar com as referências feitas a esse campo. As referências podem ser feitas por outras colunas que usam chaves estrangeiras que apontam para este campo, por *views* que listam esta coluna em

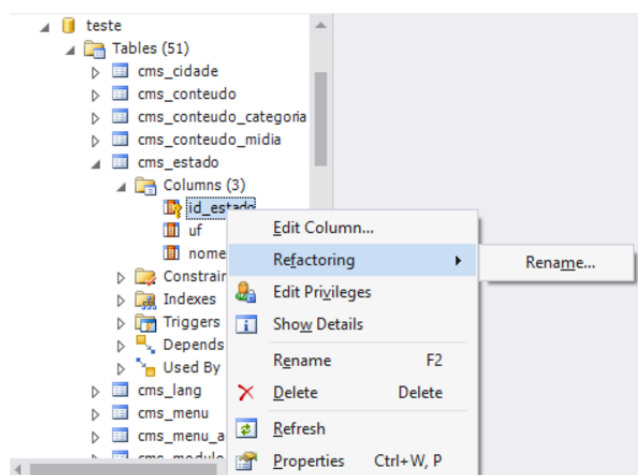


Figura 3.1: Opção de refatoração: Renomear Coluna

suas cláusulas *select* ou então por *stored procedures* que a utilizam em sua estrutura.

No entanto, clientes de banco de dados tradicionais não tratam essas dependências. Dessa forma, os desenvolvedores terão que encontrar e editar os campos, as *views* e as *stored procedures* que fazem referência a coluna renomeado, afim de manter o padrão entre os campos e de evitar que essas estruturas resultem em erro.

A ferramenta de refatoração de banco de dados do dbForge Studio, para o MySQL, poupa esse trabalho. A aplicação vai encontrar todas as chaves estrangeiras, as *views* e as *stored procedures* que fazem referência a esse campo e vai aplicar o novo nome da coluna, como pode ser visto nas Figuras 3.2, 3.3 e 3.4.

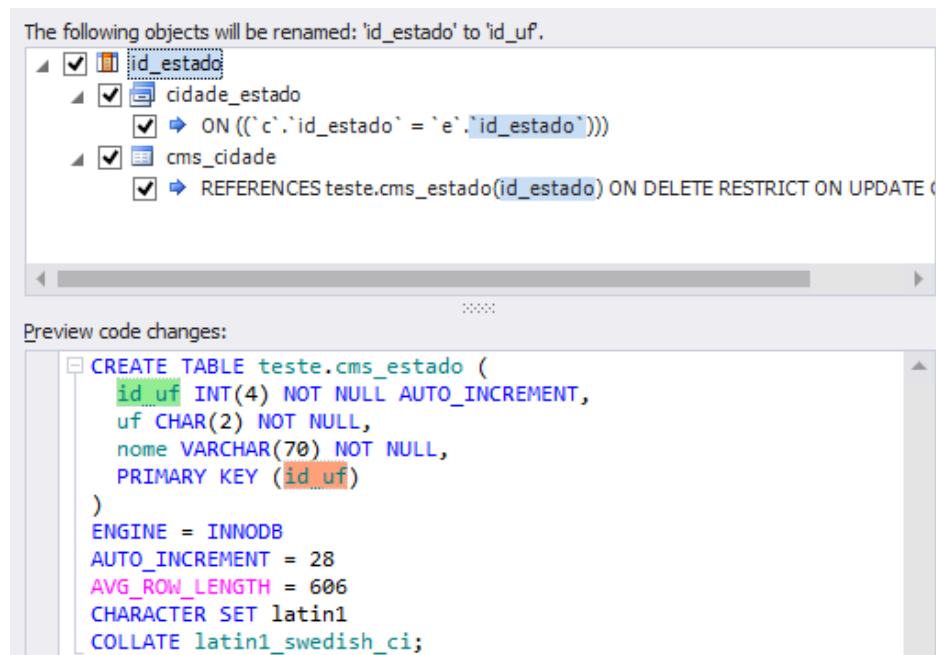


Figura 3.2: renomeando a coluna *id\_estado* para *id\_uf* e resolvendo as dependências(chave estrangeira na tabela *cms\_cidade* e campo utilizado no *join* da *view cidade\_estado*). Exibindo o comando *SQL* para criar a tabela após o ajuste

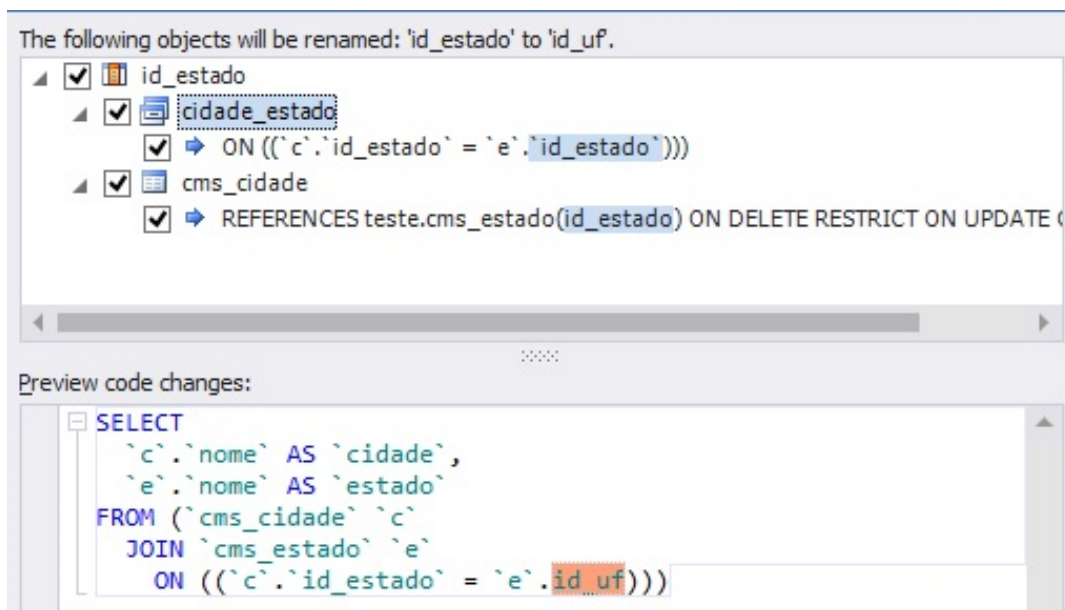


Figura 3.3: renomeando a coluna *id\_estado* para *id\_uf* e resolvendo as dependências(chave estrangeira na tabela *cms\_cidade* e campo utilizado no *join* da view *cidade\_estado*). Exibindo o *SELECT* da view *cidade\_estado* após a mudança

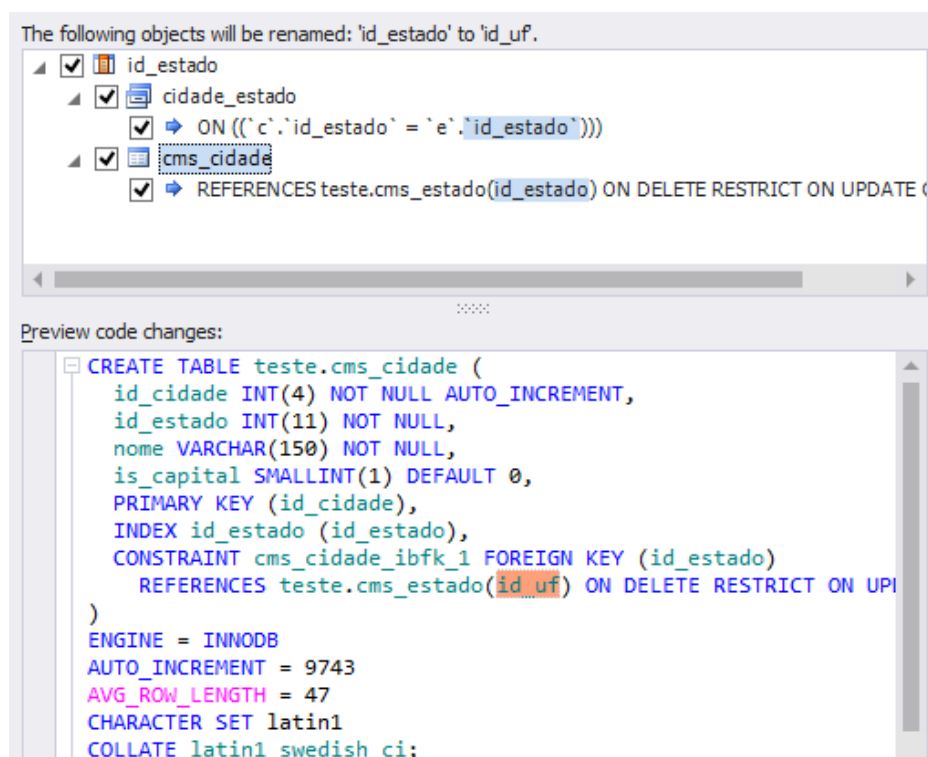


Figura 3.4: renomeando a coluna *id\_estado* para *id\_uf* e resolvendo as dependências(chave estrangeira na tabela *cms\_cidade* e campo utilizado no *join* da view *cidade\_estado*). Exibindo o comando *SQL* para criar a tabela *cms\_cidade*, após a mudança, com a referência ao campo renomeado.

A renomeação de tabela funciona como a renomeação de coluna. No caso do exemplo acima, se alterar o nome da tabela *cms\_estado* para *cms\_uf*, o trecho *JOIN cms\_estado* será mudado para *JOIN cms\_uf*.

## 3.2 SchemaCrawler

SchemaCrawler é uma ferramenta de descoberta e extração de informação do *esquema* do banco de dados, independente de plataforma, já que é desenvolvida em Java e funciona em qualquer sistema operacional que suporte Java 8 ou superior. SchemaCrawler suporta quase qualquer banco de dados que tem um driver JDBC, mas por conveniência, ele já vem com drivers dos SGBDs mais usados.

Essa ferramenta tem uma boa mistura de características úteis para gerência da estrutura de dados. Ela permite que os usuários procurem por objetos nos *esquemas* de banco de dados usando expressões regulares. SchemaCrawler vem com um conjunto de ferramentas de linha de comando que permitem que os metadados do banco de dados sejam apresentados como texto simples, texto separado por vírgulas (CSV), HTML5, HTML5 com diagramas incorporados ou JSON.

A ferramenta permite que os usuários explorem a base de dados e busquem por objetos específicos por meio da funcionalidade *grep*, através de filtros de tabelas, colunas e *stored procedures*. Isso pode ser útil, por exemplo, quando desenvolvedor quer pesquisar através de seu *esquema* para encontrar todas as tabelas que possuem a coluna *id\_estado*.

É possível também, configurar o formato de saída da informação e também quais dados devem ser mostrados, como por exemplo, suprimir os nomes de *esquemas* e nomes de *chaves estrangeiras*, ou então mostrar colunas em ordem alfabética (FATEHI, 2015).

SchemaCrawler é muito útil para fazer análise do banco, para gerar documentação e diagramas da base de dados. No entanto a funcionalidade deste software mais relevante para o presente trabalho é a *SchemaCrawler Lint*, utilizada para encontrar *maus cheiros* no banco de dados. Por exemplo, nomes de tabelas que são palavras reservadas do SQL, tais como uma tabela chamada *user*, ou então duas colunas onde uma é *chave primária* e a outra *estrangeira*, mas que têm tipos de dados diferentes.

SchemaCrawler Lint é uma ferramenta para analisar o banco de dados e encontrar potenciais falhas de projeto. Abaixo pode ser visto alguns *maus cheiros* de projeto, introduzidos por Sualeh Fatehi, que podem ser encontrados com esta ferramenta:

- `schemacrawler.tools.linter.LinterColumnTypes`: Procura colunas em tabelas diferentes, que têm o mesmo nome, mas têm diferentes tipos de dados.
- `schemacrawler.tools.linter.LinterTableWithNoPrimaryKey`: Verifica a existência de tabelas sem chave primária.
- `schemacrawler.tools.linter.LinterTableWithNoRemarks`: Verifica a existência de tabelas e colunas sem observações.
- `schemacrawler.tools.linter.LinterNullColumnsInIndex`: Verifica a existência de tabelas que possuem colunas anuláveis em um índice único.
- `schemacrawler.tools.linter.LinterTableWithNoIndexes`: Verifica a existência de tabelas sem índice.

- `schemacrawler.tools.linter.LinterTableWithQuotedNames`: Verifica a existência de tabelas que possuem espaços ou palavras reservadas no padrão ANSI SQL em seus nomes ou em nomes de suas colunas.
- `schemacrawler.tools.linter.LinterTableAllNullableColumns`: Verifica a existência de tabelas que possuem todas as colunas além da chave primária anuláveis. Elas podem acabar não contendo dados úteis, o que pode indicar um *mau cheiro*.
- `schemacrawler.tools.linter.LinterForeignKeyMismatch`: Verifica tabelas onde a chave primária tem um tipo de dado e a coluna que é chave estrangeira tem outro tipo de dado.
- `schemacrawler.tools.linter.LinterNullIntendedColumns`: Verifica a existência de tabelas onde o valor padrão é 'NULL' em vez de NULL, uma vez que isso pode indicar um erro do momento de criação da tabela.
- `schemacrawler.tools.linter.LinterTableEmpty`: Verifica se há tabelas vazias. Provavelmente seja uma tabela criada, mas que não tenha utilizada.
- `schemacrawler.tools.linter.LinterTableWithPrimaryKeyNotFirst`: Verifica a existência de tabelas onde as colunas de chave primária não estão em primeiro lugar, uma vez que esta é a convenção.
- `schemacrawler.tools.linter.LinterForeignKeyWithNoIndexes`: Verifica se há tabelas onde as chaves estrangeiras não têm índices. Isto pode causar pesquisas ineficientes.

Para utilizar essa funcionalidade é necessário que o desenvolvedor crie um arquivo XML com as configurações que devem ser executadas pela ferramenta conforme pode ser visto na Figura 3.5.

```

<schemacrawler-linter-configs>
  <linter id="schemacrawler.tools.linter.LinterColumnTypes">
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterTableWithNoPrimaryKey">
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterTableWithNoRemarks">
    <column-exclusion-pattern><![CDATA[.*EXTRA_PK\..PUBLICATIONID]]></column-exclusion-pattern>
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterNullColumnsInIndex">
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterTableWithNoIndexes">
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterTableWithQuotedNames">
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterForeignKeyMismatch">
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterNullIntendedColumns">
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterTableEmpty">
    <table-exclusion-pattern><![CDATA[.*EXTRA_PK]]></table-exclusion-pattern>
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterTableWithPrimaryKeyNotFirst">
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterTableAllNullableColumns">
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterForeignKeyWithNoIndexes">
    <run>true</run>
  </linter>
  <linter id="schemacrawler.tools.linter.LinterRedundantIndexes">
    <run>true</run>
  </linter>

```

Figura 3.5: Arquivo com as configurações dos *maus cheiros* que serão pesquisados pela ferramenta.

A interface com usuário deixa um pouco a desejar, uma vez que, para executar a ferramenta o usuário deve utilizar a linha de comando (Figura 3.6), o que não é nada amigável, já que é necessário que o desenvolvedor conheça os parâmetros que devem ser utilizados.

```

C:\Users\Leandro\Desktop\schemacrawler-14.03.03-main\_schemacrawler>schemacrawler.cmd -server=postgresql -database=teste -user=postgres -password=123456 -infolevel=standard -command=lint -linterconfigs=schemacrawler-mauscheiros-configs.xml > teste_postgres.txt
C:\Users\Leandro\Desktop\schemacrawler-14.03.03-main\_schemacrawler>

```

Figura 3.6: Linha de comando para executar a ferramenta.

Após analisar a base de dados em busca dos *maus cheiros* configurados no arquivo XML, a ferramenta pode produzir relatórios em texto(Figura 3.7), em HTML5 ou JSON.



```

Lints
=====

Database [database]
-----

column with same name but different data types id_estado [lint, medium]
[lint, medium]

public.cms_cidade [table]
-----

foreign key data type different from primary key estado_cidade_fk [lint, medium]
foreign key with no index estado_cidade_fk
empty table
no primary key
should have remarks
should have remarks id_cidade, id_estado, nome, is_capital

public.cms_estado [table]
-----

all data columns are nullable [lint, medium]
empty table
should have remarks
should have remarks nome, id_estado, uf
primary key not first

```

Figura 3.7: Relatório obtido pela análise feita pela ferramenta em busca de *maus cheiros*.

Além das funcionalidade nativas descritas acima, o SchemaCrawler é também uma API, que mapeia o banco de dados e fornece os metadados em forma de objetos Java, que torna o trabalho com metadados de banco de dados tão fácil quanto trabalhar com simples objetos java. Sendo assim, os usuários podem expandir a ferramenta implementando novas funcionalidade apartir desta API.

### 3.3 Liquibase

A utilização de ferramentas de controle de versão é uma prática muito utilizada por desenvolvedores de código, uma vez que, as alterações de código precisam ser passíveis de serem revertidas e integradas, de forma harmônica e garantindo a integridade do código. Fazer tudo isso de forma manual é um processo árduo, que normalmente gera erros.

Com o banco de dados não é diferente, principalmente quando desenvolvido por equipes que utilizam metodologias ágeis, uma vez que a base de dados está em constante mudança por meio das refatorações(PAPOTTI, 2012).

Liquibase é uma biblioteca independente de banco de dados, de código aberto, utili-

zada para rastrear, gerenciar e aplicar alterações em banco de dados. Por ser implementada em Java, o Liquibase é independente de plataforma.

Com essa ferramenta, o desenvolvedor escreve um arquivo XML, que o Liquibase chama de *changelog*. Dentro desse arquivo serão escritas as alterações/refatorações (pode ser um *create table*, um *add column*, um *insert*, um *update*, etc) do banco de dados por meio da tag *<changeSet>*, como pode ser visto na Figura 3.8.

Cada tag *<changeSet>*, deve conter apenas uma alteração/refatoração, uma vez que, o Liquibase interpreta o *changeset* como uma transformação atômica (tudo ou nada) e caso haja necessidade de um *rollback* ele será feito em todo *changeSet* (LâMPADA, 2012).

```
<?xml version="1.0" encoding="UTF-8"?>

<databaseChangeLog
  xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-2.0.xsd">
  <changeSet id="1" author="leandro">
    <createTable tableName="cms_estados">
      <column name="id_uf" type="bigint" autoIncrement="true">
        <constraints primaryKey="true" nullable="false"/>
      </column>
      <column name="uf" type="char(2)">
        <constraints nullable="false"/>
      </column>
      <column name="nome" type="varchar(70)">
        <constraints nullable="false"/>
      </column>
    </createTable>
  </changeSet>
</databaseChangeLog>
```

Figura 3.8: *changelog* com um *changeSet* para criação da tabela *cms\_estado*.

*ChangeSets* são escritos em XML, unicamente identificados pela combinação do atributo *id* da alteração/refatoração realizada, do atributo *author* e do *path* do arquivo de *ChangeLog*. Eles são escritos em uma notação específica do Liquibase, que sabe traduzir esse XML para comandos SQL de vários SGDBs.

O Liquibase armazena em uma tabela do banco a chave (*path* + *author* + *id*) dos *changeSets* que ele já rodou, assim, ele só aplica os *changesets* que ainda não rodaram. Em caso de problema em algum *release*, a ferramenta pode aplicar o *rollback* dos *changesets*.

O Liquibase conta com um conjunto de alterações/refatorações como: *Add Column*, *Rename Table*, *Rename Column* (Figura 3.9), *Drop Column*, *Alter Sequence*, *Modify Data Type* e etc. No entanto ele não é capaz de resolver as dependências das refatorações, como pode ser visto na Figura 3.10, ele executa apenas o comando especificado no XML. A importância dele no processo de refatoração está no seu controle de publicação de alteração.

```
<changeSet id="2" author="leandro">
  <renameColumn
    newColumnName="id_estado"
    oldColumnName="id_uf"
    schemaName="public"
    tableName="cms_estado"/>
</changeSet>
```

Figura 3.9: *ChangeSet* para fazer a alteração/refatoração *Rename Column*.

```
ALTER TABLE public.cms_estado RENAME COLUMN id_uf to id_estado;
```

Figura 3.10: Comando SQL gerado pelo *ChangeSet* supracitado.

### 3.4 Resumo dos Sistemas Existentes

Conforme já foi mencionado, existem poucas ferramentas que dão suporte ao desenvolvimento evolucionário de banco de dados. Como pode ser visto na Tabela 3.1, as três ferramentas estudadas foram desenvolvidas para propósitos diferentes, porém, todas apresentam funcionalidades importantes no processo de refatoração.

O Liquibase é uma ferramenta desenvolvida para controle de versão, funcionalidade fundamental para a evolução do banco de dados. Porém, esta ferramenta não é capaz de encontrar *database smells*, assim como a dbForge Studio. No entanto, o dbForge Studio conta com uma funcionalidade de refatoração, que pode ser facilmente utilizada por meio de sua interface amigável. Ele é capaz de aplicar uma refatoração e resolver automaticamente as suas dependências.

O Liquibase não resolve automaticamente as dependências de uma refatoração, para que isso aconteça, o desenvolvedor deve escrever um arquivo de configuração (*XML*) com a refatoração desejada e com as alterações necessárias para resolver as suas dependências.

O SchemaCrawler é uma ferramenta totalmente diferente das citadas acima, uma vez que, ele não altera nada da estrutura de dados. É uma ferramenta apenas para descoberta e extração de informação da estrutura do banco. No entanto, ele tem um papel fundamental no processo de refatoração, já que uma das suas funcionalidades é encontrar *database smells*. Porém, assim como o Liquibase, a sua utilização não é tão amigável como a do dbForge, já que é necessário escrever arquivos de configuração e executar a ferramenta por meio da linha de comando.

Outra característica importante do SchemaCrawler é que ele disponibiliza uma API, o que possibilita a sua expansão, já as outras duas ferramentas não contam com nem uma possibilidade de expansão.

Com relação aos SGBDs suportados, tanto o Liquibase quanto o SchemaCrawler são bem abrangentes, já que eles são compatíveis com todos os bancos que tenham um driver JDBC.

Entretanto, embora dbForge Studio seja um software direcionado apenas para o MySQL, ele possui diversas vantagens (Refatoração e depurador de stored procedures) em comparação à outros clientes de banco de dados. Além disso, conforme pode ser visto na Tabela 3.1, o dbForge tem apenas uma versão para windows, enquanto o SchemaCrawler e o

Liquibase, por serem desenvolvidos em java, podem ser utilizados em qualquer sistema operacional que suporte o java.

Tabela 3.1: Tabela de comparação entre os sistemas existentes

	SchemaCrawler	dbForge Studio for MySQL	Liquibase
Plataforma	Todas que suportam Java 8 ou superior	Windows	Todas que suportam Java 1.5 ou superior
SGBD	Todos que tem um driver JDBC	MYSQL	Todos que tem um driver JDBC
Encontra <i>data-base smells</i>	Sim	Não	Não
Refatora	Não	Sim	Sim
Resolve dependências automaticamente	Não	Sim	Não, o desenvolvedor tem que escrever o arquivo de configuração para resolve-las
Usabilidade	Ruim, configuração em XML e execução em terminal	Boa, interface amigável	Ruim, configuração em XML e execução em terminal
Expansível	Sim	Não	Não
Suporte ao processo de refatoração(Rollback e versionamento)	Não	Não, mas oferece algumas opções de suporte(Comparação e sincronização de banco)	Sim, por ser uma ferramenta de versionamento, faz o controle das alterações/refatorações

## 4 ANÁLISE DE PROJETO

Esse capítulo contém a análise feita para o desenvolvimento da ferramenta proposta no presente trabalho, por meio da documentação do projeto. Essa documentação consiste em diagrama de caso de uso, diagrama de classes e diagrama relacional de banco de dados.

### 4.1 Diagrama de Caso de Uso

O diagrama de caso descreve o cenário que mostra as funcionalidades do sistema do ponto de vista do usuário. Nele são descritos os atores, as funcionalidades e os seus relacionamentos.

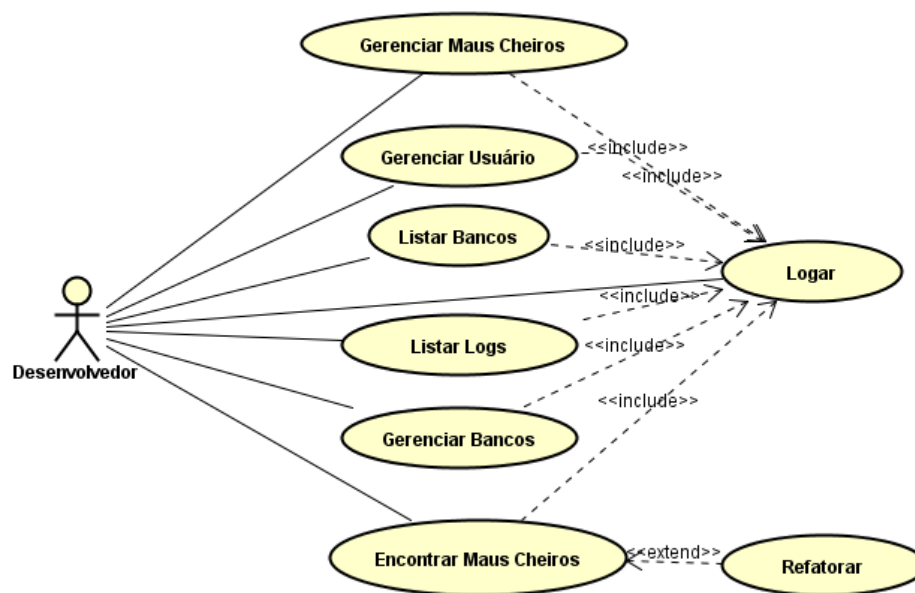


Figura 4.1: Caso de uso

- **Logar:** Operação utilizada para autenticar o desenvolvedor que irá utilizar o sistema.
- **Gerenciar Usuário:** Caso de uso que representa as operações de cadastro e de edição de usuário do sistema. Cada usuário poderá, após autenticar-se no sistema, cadastrar e gerenciar os bancos de dados que deseja analisar.

- **Listar Bancos:** Após logar no sistema, ele irá exibir a lista de bancos, previamente cadastrados, que podem ser analisados e refatorados.
- **Listar Logs:** Na tela de listagem de bancos, estando logado, o desenvolvedor poderá escolher a opção de listar os *logs* de tentativas de refatorações. Serão listadas as refatorações executadas com sucesso e as refatorações que falharam. O usuário pode selecionar um *log* para visualizá-lo de forma detalhada.
- **Gerenciar Bancos:** Caso de uso que representa as operações de cadastro, edição e exclusão dos bancos de dados dos usuários. Quando o usuário está logado no sistema, ele pode cadastrar os bancos que serão analisados pela ferramenta. Além disso, ele pode também, excluir esses bancos e editar as informações deles.
- **Encontrar Maus Cheiros:** Essa operação é a mais importante do sistema. Na listagem de bancos, o desenvolvedor poderá escolher opção *maus cheiros*, que direcionará o usuário para a página onde terá a lista de *maus cheiros* disponíveis na ferramenta e mais a lista de *maus cheiros* cadastrados pelo usuário.

Ao escolher a opção **encontrar** de um mau cheiro disponível, o sistema então varrerá toda estrutura da base de dados e exibirá ao usuário os pontos onde a ferramenta encontrou esse *mau cheiro*. Além disso, exibirá os comandos *SQL* para refatorá-los, bem como os comandos para resolver as dependências dessas refatorações. Para executar essa operação é necessário que o desenvolvedor esteja logado.

- **Refatorar:** Após executar a operação de **encontrar** o *mau cheiro*, o desenvolvedor poderá editar as sugestões de refatorações da ferramenta e acionar a execução das refatorações.

Ao executar esta operação, o sistema irá executar os comandos *SQLs* sugeridos pela ferramenta, afim de ajustar a base de dados. Caso isso não seja possível por algum motivo, a ferramenta irá exibir uma mensagem com o erro de *SQL*. Independente do resultado da execução destes comandos a ferramenta guardará um *log* das operações.

- **Gerenciar Maus Cheiros:** Caso de uso que representa as operações de cadastro, edição e exclusão de *maus cheiros* cadastrados pelos usuários.

Além dos *maus cheiros* nativos da ferramenta, o usuário tem a opção de inserir novas opções de *maus cheiros*. Para isso, o usuário deve desenvolver a classe do *mau cheiro* que ele pretende analisar, essa classe deve seguir os padrões da ferramenta.

Além de inserir, o usuário pode editar os *maus cheiros* cadastrados por ele e excluí-los, caso seja necessário.

## 4.2 Diagrama de Classes

Como pode ser visto na Figura 4.2, na Figura 4.3 e na Figura 4.4 os diagramas de classes definem todas as classes que a aplicação necessita possuir, bem como a forma como estas classes interagem entre si e qual a responsabilidade de cada classe na realização das operações solicitadas pelo autor.

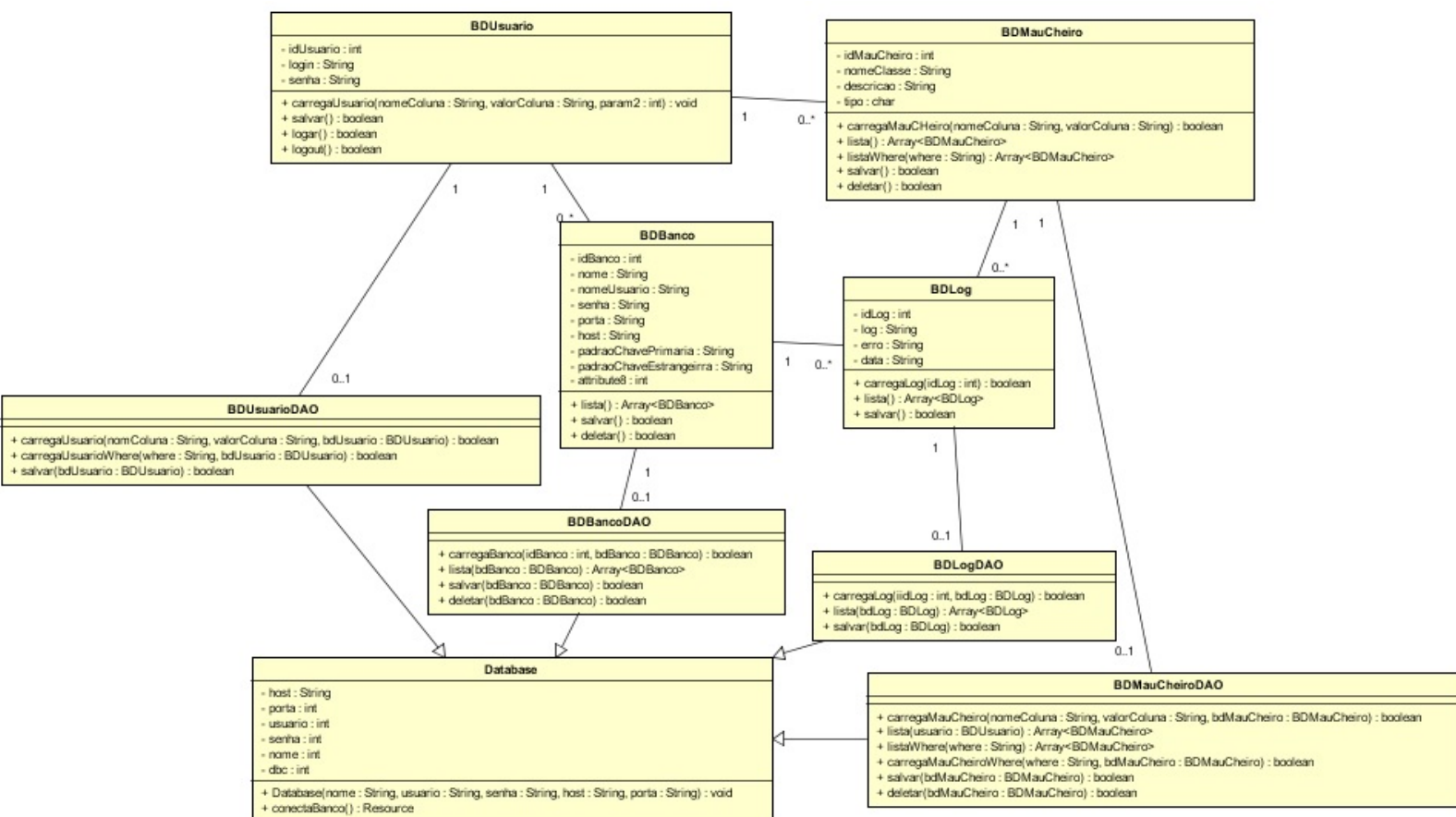


Figura 4.2: Diagrama de classes da estrutura interna da ferramenta

- **Database;** Classe que tem como finalidade fazer a conexão com o banco e disponibilizar essa conexão para as classes que herdam ela, ou seja, todas as classes DAO. Essas classes herdam *Database*, que em seu construtor chama o método *conecta-Banco*, criando assim uma conexão por meio do *pg\_connect*.
- **BDUsuario;** Classe utilizada para manipular os usuários do sistema. Através da classe é possível logar e autenticar estes usuários, bem como validar a sua navegação. A classe contém uma lista de bancos de dados e uma lista *maus cheiros* cadastrados pelo usuário.
- **BDUsuarioDAO:** Classe responsável por fazer a interação com o banco de dados, com o objetivo de manipular as informações dos usuários do sistema. Por meio da classe, é possível, por exemplo, inserir, atualizar os usuários e carregar as informações de um usuário.
- **BDBanco:** Essa classe representa a tabela *bancos*, utilizada para armazenar as informações dos bancos cadastrados pelos usuários. Ela contém os atributos utilizados pela classe *Banco*, para fazer a conexão com a base de dados que será analisada e carregar a sua estrutura.

Além disso, esta classe contém atributos da configuração do banco, utilizadas pelos *maus cheiros*, como por exemplo, o padrão da chave primária.

- **BDBancoDAO:** Classe utilizada para fazer a interação com o banco de dados, com o objetivo de manipular as informações dos bancos cadastrados pelos usuários. Por meio da classe, é possível, por exemplo, inserir, atualizar, deletar e carregar as informações de um ou de vários bancos.
- **BDLog:** Essa classe é a estrutura responsável por manipular os *logs* das refatorações dos bancos de dados analisados. Ela contém atributos como a data do *log* e o *script SQL* executado pela refatoração, bem como, o banco de dados e o *mau cheiro* analisado.
- **BDLogDAO:** Classe utilizada para fazer a interação com a base de dados, com o objetivo de manipular as informações dos *logs* gerados pela execução das refatorações. Por meio da classe, é possível, por exemplo, inserir, listar e carregar as informações de um ou de vários *logs* armazenados no banco de dados.
- **BDMauCheiro:** Essa classe representa a tabela *maus\_cheiros*, utilizada para armazenar as informações dos *maus cheiros* nativos do sistema e os *maus cheiros* cadastrados pelos usuários.

Ela contém os atributos como *nomeClasse*, utilizado pelo sistema para instanciar a classe de cada *mau cheiro*, também contém o atributo *descricao*, que representa a descrição do *mau cheiro* armazenada no banco de dados.

- **BDMauCheiroDAO:** Classe utilizada para fazer a interação com a base de dados, com o objetivo de manipular as informações dos *maus cheiros*. Por meio dela, é possível, por exemplo, inserir, listar e carregar as informações de um ou de vários *maus cheiros*.



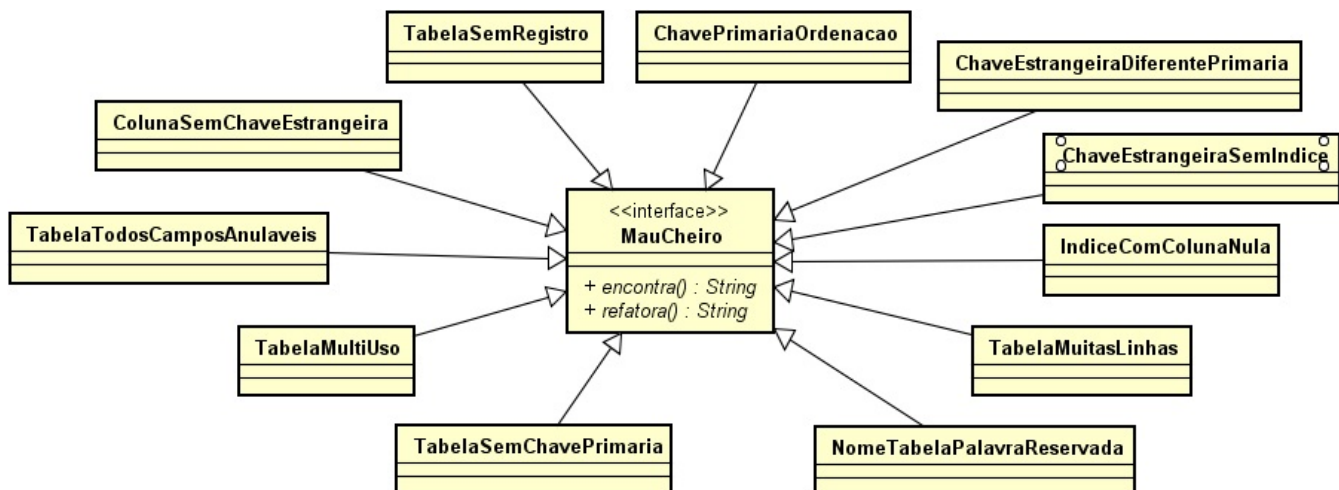


Figura 4.3: Diagrama de classes dos *maus cheiros*

- ***iMauCheiro***: Essa interface tem o objetivo de criar um contrato, que obriga as classes concretas que implementam a interface desenvolver os métodos *encontra* e *refatora*.

O método *encontra* das classes concretas que implementam *iMauCheiro* contém a lógica responsável por procurar o *mau cheiro* no banco de dados passado como parâmetro para ele.

O método *refatora* é responsável por percorrer os *maus cheiros* encontrados e montar a *SQL* com as sugestões de refatoração que serão exibidas para o usuário.

- ***TabelaSemChavePrimaria*, *TabelaMultiUso*, *TabelaTodosCamposAnulaveis*, *ColunaSemChaveEstrangeira*, *TabelaSemRegistro*, *ChavePrimariaOrdenacao*, *ChaveEstrangeiraDiferentePrimaria*, *ChaveEstrangeiraSemIndice*, *IndiceComColunaNula*, *TabelaMuitasLinhas* e *NomeTabelaPalavraReservada***: São as classes concretas que implementam a interface *iMauCheiro*. Sendo assim, nelas são desenvolvidos os métodos *encontra* e *refatora*, de acordo com a característica de cada *mau cheiro*, ou seja, o código para encontrar *mau cheiro* é particular a cada classe. Assim como o método para gerar o script de refatoração, onde cada classe representante de um *mau cheiro* tem o seu método *refatora*.

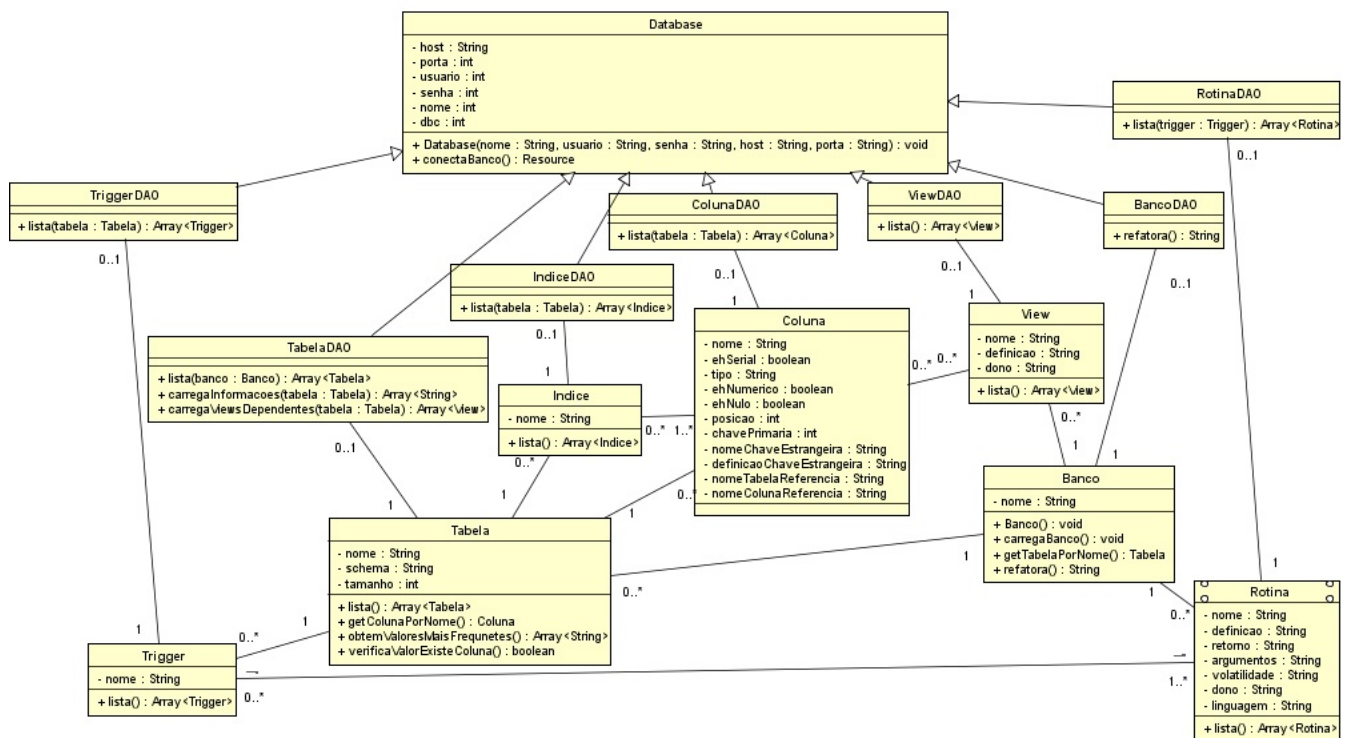


Figura 4.4: Diagrama de classes da estrutura do banco que será analisado pela ferramenta

- **Banco:** Essa classe é responsável por carregar toda a estrutura do banco de dados a ser analisado. O construtor dessa classe, chama o método *carregaBanco* que se encarregará de setar os atributos *tabelas*, *views* e *rotinas*.
- **BancoDAO:** A classe *BancoDAO* é utilizada para executar a refatoração, por meio do método *refatora*, que recebe como parâmetro a *String* com a *SQL* e um objeto *BDLog* que salvará o log da operação realizada.

- **Tabela:** Classe que representa as tabelas do banco de dados analisado, ela contém os atributos referentes aos metadados das tabelas do banco. Além disso, contém atributos como *colunas*, *indices* e *triggers*.

Essa classe conta também com métodos utilizados para obter informações da tabela, como o método *getColunaPorNome*, que recebe como parâmetro o nome de uma coluna e retorna um objeto da classe *Coluna* presente na tabela.

- **TabelaDAO:** Classe utilizada para fazer a interação com a base de dados, com o objetivo principal de carregar as tabelas do banco de dados por meio do método *lista*, que busca todas as tabelas do banco e consequentemente, carrega os atributos *colunas*, *indices* e *triggers* da classe *Tabela*.
- **Coluna:** Esta classe é uma representação da estrutura das colunas de uma tabela. Os atributos dela refletem os metadados referentes as colunas. Alguns desses atributos são *tipo*, *ehSerial*, *ehNumerico*, *chavePrimaria*, *nomeColunaReferencia*, entre outros.
- **ColunaDAO:** Classe responsável por fazer a conexão com o banco de dados e por meio do método *lista* realizar uma consulta *SQL* para obter as informações correspondentes aos atributos da classe *Coluna*. Dessa forma é possível instanciar e

carregar os objetos dessa classe, retornando um vetor com objetos, pertencentes a tabela passada como parâmetro.

- **View:** Classe que representa a estrutura das *views* do banco analisado. Ela contém atributos que correspondem aos metadados do banco, como por exemplo, o atributo *definicao*, que representa a *SQL* de criação da *view*. Ela também contém uma classe que faz a interação com o banco, chamada de *ViewDAO*.
- **ViewDAO:** Esta classe contém apenas o método *lista*, que faz a conexão com o banco de dados e por meio de uma consulta *SQL* obtém as informações correspondentes aos atributos da classe *View*. Sendo assim, ele instancia e carrega os objetos dessa classe, retornando um vetor com esses objetos.
- **Indice:** Esta classe é uma representação dos índices das tabelas, ela é formada pelos atributos que correspondem aos metadados dos índices das tabelas do banco de dados. Entre os atributos que ela contém, está uma lista das colunas que fazem parte do índice.
- **IndiceDAO:** Esta classe contém apenas o método *lista*, que faz a conexão com o banco de dados e por meio de uma consulta *SQL* obtém as informações correspondentes aos atributos da classe *Indice*. Sendo assim, ele instancia e carrega os objetos dessa classe pertencentes a tabela passada como parâmetro, retornando um vetor com esses objetos.
- **Rotina:** Classe que representa a estrutura das rotinas do banco de dados, sendo elas pertencentes a uma *trigger* ou não. Ela contém atributos que correspondem aos metadados de uma rotina. Um dos atributos desta classe é o *definicao*, que representa o script *SQL* para criação da rotina.
- **RotinaDAO:** Esta classe contém apenas o método *lista*, que faz a conexão com o banco de dados e por meio de uma consulta *SQL* obtém as informações correspondentes aos atributos da classe *Rotina*. Ela pode ou não receber como parâmetro uma *trigger*, caso ela receba, serão listadas apenas rotinas chamadas por essa *trigger*. Caso não receba, serão listadas todas rotinas do banco de dados.
- **Trigger:** Esta classe representa a estrutura de uma *trigger* pertencente a uma tabela. Ela é formada pelos atributos correspondentes aos metadados das *triggers* do banco de dados analisado. Um dos atributos da classe *Trigger* é a lista de rotinas, que corresponde a lista de rotinas que a *trigger* dispara.
- **TriggerDAO:** Esta classe contém apenas o método *lista*, que faz a conexão com o banco de dados e por meio de uma consulta *SQL* obtém as informações correspondentes aos atributos da classe *Trigger*. Ela recebe uma tabela como parâmetro, para que a função retorne um vetor de *triggers* dessa tabela.

### 4.3 Diagrama Relacional do Banco de Dados da Ferramenta

O diagrama relacional de banco de dados descreve a estrutura utilizada para armazenar as informações utilizadas pela aplicação. Nele são descritas as tabelas e seus relacionamentos de acordo com que ilustra a Figura 4.5.

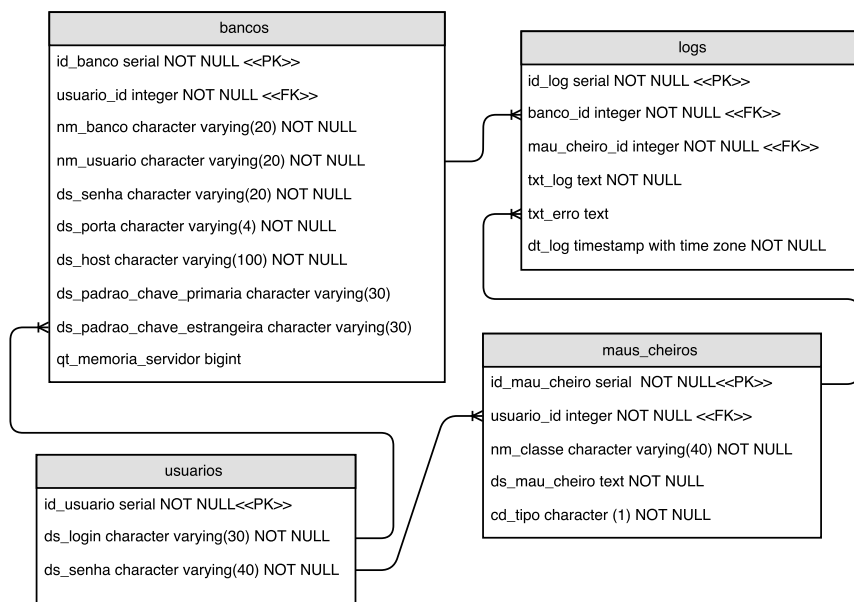


Figura 4.5: Diagrama relacional de banco de dados

**usuarios:** Essa tabela é responsável por armazenar os usuários do sistema. Para que seja possível a autenticação dos usuários. Essa estrutura contém os campos `ds_login` e `ds_senha`. São vinculados a essa tabela os bancos de dados cadastrados por cada usuário, para que seja possível identificar os responsáveis por cada banco. Também são vinculados a essa tabela os maus cheiros cadastrados pelo usuário.

**bancos:** Essa tabela contém os registros dos bancos cadastrados pelo usuário. Ela armazena dados para a conexão do banco como `host`, nome do banco, porta, usuário e senha para logar no banco de dados que será analisado pela ferramenta. Além dessas informações, essa tabela também armazena algumas informações do banco de dados que são utilizadas pelos *maus cheiros*, tais como, o padrão de chave primária, o padrão de chave estrangeira e o tamanho da memória do servidor.

**maus\_cheiros:** Essa tabela armazena os *maus cheiros* que serão analisados pela ferramenta, ela tem relação com a tabela *usuarios*, para que cada usuário do sistema possa cadastrar novos *maus cheiros*. Além disso, ela tem relação com a tabela *logs*, onde cada *log* gravado armazena o *id* do *mau cheiro* analisado. O principal campo dessa tabela é o campo `nm_classe`, nele é armazenado o nome da classe do *mau cheiro*, usado pela ferramenta para carregar a classe por *reflection*<sup>1</sup>.

**logs:** Tabela que contém o histórico das tentativas de refatoração dos *maus cheiros* em cada banco, bem como a data dessa tentativa. Os comandos *SQL* executados para refatorar um determinado *mau cheiro* serão armazenados no campo `txt_log`. Caso a execução destes comandos resultem em erro, esse erro será armazenado no campo `txt_erro`.

<sup>1</sup>Definição encontrada em <http://culttt.com/2014/07/02/reflection-php/>

## 5 IMPLEMENTAÇÃO

Nesse capítulo descreve como a ferramenta proposta foi construída, bem como quais tecnologias foram utilizados durante o fase de desenvolvimento.

### 5.1 Estrutura do PostgreSQL

Como o objetivo da ferramenta é analisar um banco de dados cadastrado pelo usuário, o primeiro passo para o desenvolvimento dessa ferramenta foi criar uma estrutura em classes, capaz de representar os metadados do banco de dados analisado. Para isso foi necessário criar classes que fizessem conexão com o banco de dados do usuário e por meio dos esquemas *pg\_catalog* e *information\_schema* obtivesse os metadados necessários para montar a estrutura do banco.

Para facilitar o entendimento da ferramenta será descrito algumas informações referentes aos dois esquemas supracitados.

O *pg\_catalog* é um esquema que contém os catálogos do sistema, que são os locais onde os *SGBDs* relacionais armazenam os metadados do esquema, tais como informações sobre tabelas, colunas, e informações de controle interno. No caso do *PostgreSQL*, os catálogos do sistema são tabelas comuns.

Normalmente, os catálogos do sistema não devem ser modificados manualmente, sempre existe um comando *SQL* para fazê-lo (Por exemplo, *CREATE DATABASE* insere uma linha no catálogo *pg\_database* e cria realmente o banco de dados no disco)(GROUP, 2016a).

Abaixo serão descritos alguns catálogos utilizadas pela ferramenta:

- ***pg\_tables*:** A *view pg\_tables* fornece acesso a informações úteis sobre todas as tabelas do banco de dados.
- ***pg\_views*:** A *view pg\_views* fornece acesso a informações úteis sobre todas as visões do banco de dados.
- ***pg\_attribute*:** O catálogo *pg\_attribute* armazena informações sobre as colunas das tabelas. Existe exatamente uma linha em *pg\_attribute* para cada coluna de cada tabela do banco de dados
- ***pg\_class*:** O catálogo *pg\_class* cataloga as tabelas e tudo mais que possui colunas, ou é de alguma forma semelhante a uma tabela. Ela é uma tabela de relacionamento entre as colunas e as demais estruturas.
- ***pg\_constraint*:** O catálogo *pg\_constraint* armazena restrições de verificação de chave primária, unicidade e chave estrangeira em tabelas.

- ***pg\_index***: O catálogo *pg\_index* contém parte das informações sobre índices. O restante se encontra, em sua maioria, em *pg\_class*.
- ***pg\_proc***: O catálogo *pg\_proc* armazena informações sobre as funções (ou procedimentos)
- ***pg\_type***: O catálogo *pg\_type* armazena informações sobre tipos de dado.

O *information\_schema* consiste em um conjunto de *views* contendo informações sobre os objetos definidos no banco de dados corrente. Ele é definido no padrão *SQL* e, portanto, pode-se esperar que seja portátil e permaneça estável. Isto é diferente dos catálogos do sistema, que são específicos do *PostgreSQL*. Entretanto, existem muitas informações que as *views* do *information\_schema* não contém, para obter este tipo de informação, é necessário consultar os catálogos do sistema (GROUP, 2016b).

- ***columns***: A *view columns* mostra informações sobre todas as colunas das tabelas e *views* do banco de dados. As colunas do sistema não são incluídas. Somente são mostradas as colunas que o usuário corrente tem acesso.
- ***parameters***: Contém informações sobre os parâmetros de todas as funções no banco de dados corrente.
- ***routines***: Contém todas as funções no banco de dados corrente. Somente são mostradas as funções que o usuário corrente pode acessar.
- ***tables***: A *view tables* contém todas as tabelas e *views* definidas no banco de dados corrente. Somente são mostradas as tabelas e *views* que o usuário corrente pode acessar.
- ***triggers***: A *view triggers* contém todos os gatilhos definidos no banco de dados corrente, que pertencem ao usuário corrente.

## 5.2 Implementação das Classes

Nesta sessão será descrita a implementação das classes que representam a estrutura do banco analisado e das classes de que representam cada mau cheiro.

### 5.2.1 Banco Analisado

Primeiramente foi desenvolvida a classe *Banco*, que em seu construtor chama a função *carregaBanco*, essa por sua vez define os atributos, *tabelas*, *views* e *rotinas*. Ela instancia um objeto da classe *Tabela* e logo em seguida chama o método *setTabelaDAO*, passando como parâmetro um objeto recém-criado da classe *TabelaDAO*, que será responsável por fazer a conexão com o banco de dados e obter os metadados das tabelas do banco analisado.

Feito isso, é feita uma atribuição ao atributo *tabelas*, da classe *Banco*, com o retorno do método *lista* da classe *Tabela* criada. Este método, simplesmente chama o método *lista* do seu objeto da classe *TabelaDAO*. Esse método da classe *DAO* faz uma conexão com o banco e por meio de uma *SQL* em *pg\_tables*, obtém informações como nome da tabela, nome do esquema e tamanho dela.

Depois de obter os registros das tabelas a ferramenta percorre todos registros e para cada registro é criado um objeto da classe *Tabela* e definidas as informações obtidas na

*SQL*. Logo após, é definido o atributo *colunas* por meio do método *lista* de um objeto *Coluna* recém-criado. Esse método *lista*, simplesmente chama o método *lista* do objeto *ColunaDAO*, presente dentro do objeto *Coluna*.

Esse método recebe como parâmetro a tabela a qual a coluna faz parte e através de uma *SQL* obtém informações como o tipo da coluna, se a coluna é *NOT NULL*, se é uma chave primária, ou se faz referência a outra coluna por meio de chave estrangeira. Essas informações são obtidas consultando tabelas e *views* como *pg\_class*, *pg\_constraint*, *pg\_attribute*, *columns* entre outras.

Após montar o vetor com as colunas é a vez de definir o atributo *indices* da tabela, por meio do método *lista* do objeto *Indice*. Esse método chama o método *lista* de seu atributo *indiceDAO*, que retorna um vetor de *índices*, onde os objetos desse vetor são preenchidos com as informações obtidas por meios de uma *SQL* nos catálogos *pg\_class*, *pg\_index* e *pg\_attribute*.

Com as colunas e os índices carregados, o próximo passo é definir a lista de *triggers* do objeto *Tabela*, da mesma forma que com as colunas e os índices, é chamado o método *lista*, que por sua vez chama o método *lista* do objeto *triggerDAO*, que faz a conexão com o banco e por meio de uma *SQL* obtém a lista de *triggers* da tabela corrente. Essa *SQL* consulta catálogos como *pg\_trigger*, *pg\_class* e *pg\_tables* para que seja possível obter as informações necessárias e retorna uma lista com as *triggers* da tabela.

Depois de definir a lista de tabelas do banco, o sistema faz o mesmo procedimento para definir o atributo *views* da classe *Banco*. Sendo assim, é chamado o método *lista* da classe *View* que por sua vez chama o método *lista* do seu atributo *viewDAO*, esse faz uma consulta *SQL* para obter as *views* do banco por meio da tabela *pg\_view* e retorna uma lista de *views*.

Para definir o objeto *rotinas* da classe banco, também é executado o método *lista* da classe *Rotina*, que retorna o resultado do método *lista* da classe *RotinaDAO*, que por sua vez faz a conexão com o banco de dados e obtém, por meio de uma *SQL* as informações das rotinas do banco. Com essas informações é montado um vetor com as rotinas e retornado para que seja atribuído ao atributo *rotinas* da classe *Banco*.

### 5.2.2 Maus Cheiros

Após mapear o banco de dados relacional nas classes da ferramenta, o próximo passo foi desenvolver as classes dos *maus cheiros*. Para isso, foi desenvolvida a interface *iMauCheiro*, que tem apenas a assinatura de dois métodos, o método *encontra* e o *refatora*.

Para cada *mau cheiro* analisado foi criada uma classe, que implementa a interface *iMauCheiro* e obrigatoriamente implementa o método *encontra* e *refatora*, com a lógica particular de cada uma dessas classes como pode ser visto logo abaixo:

- **ChaveEstrangeiraDiferentePrimaria:** O método *encontra* percorre as colunas de todas tabelas do banco de dados e por meio do método *getNomeColunaReferencia* verifica se a coluna faz parte de uma chave estrangeira. Caso esse método retorne algum valor então a coluna faz parte de uma chave estrangeira, nesse caso é verificado o tipo da coluna que faz referência e da coluna referenciada por meio do método *getTipo* dos dois objetos.

Na hipótese de os valores retornados pelo método serem diferentes os dois objetos são adicionados em um vetor de colunas com mau cheiro.

Depois de analisar cada coluna do banco e montar o vetor com todas as colunas que apresentam esse *mau cheiro*, a função faz uma chamada ao método *refatora*

passando como parâmetro o banco de dados e o vetor de *mau cheiros* que devem ser refatorados.

O método *refatora*, por sua vez percorre o vetor de *maus cheiros*, que tem em cada posição um vetor com dois elementos, a coluna chave estrangeira e a coluna chave primária. Para cada coluna chave estrangeira é montado *ALTER COLUMN* para alterar o tipo dela para o tipo da coluna chave primária.

Por fim, a função *refatora* retorna uma *String* com um *script* de refatoração para todas as colunas que apresentam esse *mau cheiro*.

- **ChaveEstrangeiraSemIndice:** O método *encontra* dessa classe percorre as colunas de todas tabelas do banco de dados, caso ele encontre uma coluna chave estrangeira ele percorre os índices da tabela e verifica se essa coluna está em algum desses índices.

Na hipótese dessa coluna não fazer parte de nem um índice ela é adicionada a um vetor de colunas com esse *mau cheiro*, que será passado como parâmetro para o método *refatora*.

O método *refatora* percorre o vetor de colunas com *mau cheiro* e gera um script com um *CREATE INDEX* para cada coluna chave estrangeira que não faz parte de um índice. Por fim esse método retorna uma *String* com os comandos *SQL* gerados.

- **ChavePrimariaOrdenacao:** O método *encontra* dessa classe percorre as colunas de todas tabelas do banco de dados e para cada tabela ele verifica se existe algum campo que não é chave primária antes dos campos que são chave primária. Essa verificação é possível pois as colunas da classe *Tabela* são ordenados com base no campo *ordinal\_position* da view *columns*.

Nesse caso, o método *refatora* não gera nem um script de sugestão de refatoração, apenas mostra para o usuário quais são as chaves primárias de cada tabela, que não são os primeiros campos.

- **ColunaSemChaveEstrangeira:** O método *encontra* dessa classe percorre as colunas de todas tabelas do banco de dados, e analisa o nome de cada coluna. Caso o nome da coluna siga o padrão de nome (cadastrado pelo usuário no cadastro do banco) de chaves estrangeiras, o sistema verifica se essa coluna é uma chave estrangeira por meio do método *getnomeColunaReferencia*.

Se a coluna não fizer referência a nem uma coluna chave primária é possível que ela seja uma chave estrangeira sem a devida restrição.

Após identificar a possível chave estrangeira, o sistema irá procurar uma possível chave primária baseada no padrão de nome de chave primária (cadastrado pelo usuário no cadastro do banco). Além disso, ele irá verificar se o tipo de dado da possível chave primária é igual ao tipo de dado da possível chave estrangeira.

Outra verificação que é feita, é com relação aos dados das duas colunas, uma vez que o sistema verifica se mais de 90% dos dados armazenados na possível chave estrangeira correspondem aos valores armazenados na possível chave primária.

A possível chave estrangeira é adicionada ao vetor de *maus cheiros*, junto com a chave primária correspondente.



O método *refatora* percorre o vetor com as possíveis chaves estrangeiras e caso exista uma coluna chave primária para essa coluna, ele monta um comando *ADD CONSTRAINT* para adicionar a restrição a essa possível chave estrangeira.

No caso do sistema não ter encontrado uma chave primária correspondente, ele apenas irá colocar na *String* de retorno o nome da possível chave estrangeira e informar que não encontrou a coluna que poderia ser uma possível chave primária.

- **IndiceComColunaNula:** O método *encontra* dessa classe percorre os índices de todas tabelas do banco de dados, e verifica se é um índice único, por meio do método *getEhUnico* que armazena a informação *indisunique* do catalogo *pg\_index*.

Após encontrar um índice único o sistema percorre as colunas desse índice e verifica se a coluna aceita o valor *null*, por meio do método *getEhNulo*, caso esse método retorne 1, ou seja, a coluna aceita o valor *null*. Sendo assim essa coluna é adicionada ao vetor de colunas que apresentam esse *mau cheiro*.

O método *refatora* dessa classe percorre todas as colunas do vetor de colunas gerado pelo método procura e adiciona na *String* de retorno um *ALTER COLUMN* para tornar a coluna não anulável. Caso o usuário execute o *SQL* sugerido pela ferramenta todas as colunas que fazem parte de um índice único e que aceitam valores nulos, passaram a não aceitar mais esses valores.

- **TabelaMuitasLinhas:** O método *encontra* dessa classe percorre todas as tabelas do banco de dados e verifica se o tamanho da tabela é menor que o tamanho da memória do servidor, uma vez que, esse é o critério adotado por alguns desenvolvedores, como o Telles cita em (TELLES, 2013).

O tamanho da memória é cadastrado no momento em que o usuário cadastra o banco de dados e o tamanho da tabela é obtido pela função *getTamanho* do objeto *Tabela*. Essa função retorna a informação obtida por meio da função do *PostgreSQL* *pg\_table\_size*.

A refatoração para esse *mau cheiro*, seria particionar a tabela, mas isso não é possível fazer automaticamente, uma vez que não é possível definir uma heurística capaz de analisar a tabela e definir qual é o melhor campo, por exemplo, que seria usado para definir as partições da tabela, bem como seria necessário repetir o processo de particionamento para todas as tabelas que fazem referência a essa tabela.

O método *refatora*, nesse caso, apenas retorna uma *String* com a lista de todas as tabelas com muitas linhas. Além do nome da tabela, o sistema retorna o tamanho e a quantidade de linhas dessa tabela.

- **TabelaMultiUso:** Segundo Ambler, a principal característica de uma tabela multiuso é a presença de campos nulos em determinados registros enquanto em outros registros esses campos são preenchidos e outros campos são nulos.

Um exemplo, seria uma tabela que armazena dados de pessoa física e pessoa jurídica, nesse caso, quando o registro for de uma pessoa física serão preenchidos campos como *CPF*, *nome*, *dt\_nascimento* e quando o registro for de uma empresa serão preenchidos campos como *CNPJ*, *nome\_fantasia* e *url\_site*.

O método *encontra* dessa classe percorre todas tabelas e para cada tabela ele percorre todos os registros dela, fazendo uma análise a procura de algum padrão que represente esse *mau cheiro*.

Primeiro, esse método analisa cada registro da tabela e monta grupos de campos nulos, baseados nos registros analisados. Por exemplo, no primeiro registro, os campos *CPF*, *nome* e *dt\_nascimento*, então é montado um grupo com esses campos. No segundo registro os campos *CNPJ*, *nome\_fantasia* e *url\_site* estão nulos, nesse caso é montado um outro grupo com esses campos.

Essa análise é feita em todos os registros e é montado um vetor com todos os grupos organizado em ordem decrescente de ocorrências desses grupos. Em seguida, os grupos que aparecem em poucos registros são descartados.

Uma vez obtido o vetor com os grupos mais relevantes, é feita uma verificação, com o objetivo de verificar se existe algum grupo que é o oposto de outro, ou seja, se a maioria dos elementos de um grupo não aparecem em outro grupo e vice-versa. Caso existam grupos muito diferentes essa tabela é adicionada ao vetor de tabelas com mau cheiro.

A refatoração para esse mau cheiro, seria o particionamento de tabela horizontal. No entanto não é possível fazê-lo automaticamente, uma vez que é necessário remodelar a tabela original e criar uma nova tabela, sendo assim, teria que ser decidido quais campos iriam ficar em qual tabela e realocar todas referências da tabela original para a nova tabela caso elas existissem nos campos que iriam compor essa tabela.

O método *refatora*, nesse caso apenas retorna uma *String* com a lista de todas as tabelas multiuso.

- ***TabelaSemChavePrimaria:***

O método *encontra* dessa classe, percorre todas as colunas de cada tabela e verifica se o método *getChavePrimaria* de alguma dessas colunas retorna um valor diferente de nulo.

Esse valor retornado é o obtido no momento em que as colunas das tabelas são carregadas no sistema, por meio de um *SQL* que verifica se essa coluna tem um registro correspondente na *view pg\_index* e se esse registro armazena *true* no campo *indisprimary*.

Caso nem uma coluna da tabela retorne um valor quando chamado o método *getChavePrimaria* a tabela é adicionada ao vetor de tabelas com esse *mau cheiro*.

Para adicionar uma chave primária na tabela deve-se então seguir convenção, onde as colunas chave primárias são as primeiras colunas da tabela, porém, para isso é necessário recriar a tabela. Nesse caso teria que dropar a tabela existente e criar uma nova com os mesmos campos e as mesmas configurações, porém com o campo chave primária adicionado antes de todos os outros campos.

Como umas das regras adotadas no desenvolvimento da ferramenta foi não desenvolver refatoração que fizesse *DROP* em tabela, uma vez que isso é muito arriscado já que podem ser *deletadas* as dependências dessa tabela. Nesse caso o método *refatora* retorna uma *String* com a lista de tabelas que não contem chave primária.

- ***TabelaSemRegistro:***

O método *encontra* dessa classe percorre todas as tabelas do banco e verifica, por meio do método *getDados* se a tabela contém registros, no caso de não conter, essa tabela é adicionada no vetor de tabelas com esse *mau cheiro*.

A refatoração para esse mau cheiro seria deletar a tabela, mas como já foi citado, no desenvolvimento da ferramenta foi optado por não fazer esse tipo de alteração, uma vez que ele pode impactar no sistema de forma muito negativa. Um exemplo seria uma tabela que não tem nem um registro, porém tem relação com *views*, *stored procedures* e outras tabelas.

Sendo assim, o método *refatora* dessa classe apenas retorna uma *String* com a lista de tabelas que não contém nem um registro, apontando para o usuário onde o banco de dados apresenta esse *mau cheiro*.

- ***TabelaTodosCamposAnulaveis:***

O método *encontra* dessa classe percorre todas as tabelas e para cada tabela ele verifica se existe pelo menos uma coluna além da chave primária com a restrição *NOT NULL*, ou seja, se todas as colunas podem ser nulas, nesse caso essa tabela é adicionada ao vetor de tabelas com esse *mau cheiro*.

O método *refatora* percorre os registros de cada uma dessas tabelas e para cada registro ele adiciona em um vetor as colunas que o valor não seja nulo. Dessa forma é montado um vetor onde o índice é o nome da coluna e o valor é o número de vezes que o valor dessa coluna foi diferente de nulo.

O sistema percorre esse vetor com a quantidade de vezes que cada coluna não apresenta um valor nulo e verifica quais colunas estão preenchidas em mais de 80% dos registros. Para cada uma dessas colunas é adicionado a *String* de retorno um *ALTER COLUMN* para tornar a coluna *NOT NULL*.

A execução desse comando pode resultar em erro, uma vez que, podem existir registros nulos nas colunas que serão alteradas. Nesse caso o usuário deve alterar esses registros com um valor apropriado. Foi tentado contornar esse problema, sugerindo ao usuário um *UPDATE* onde o valor é nulo, definindo o valor mais frequente encontrado na coluna, mas essa solução acabou se mostrando inadequada.

- ***NomeTabelaPalavraReservada:***

O método *encontra* dessa classe percorre todas as tabelas e verifica se o nome dessas tabelas é uma palavra reservada do *PostgreSQL*. Caso seja, ele adiciona essa tabela em um vetor de tabelas com esse *mau cheiro*.

O método *refatora* dessa classe então, percorre o vetor de tabelas com o *mau cheiro* e para cada tabela ele chama os métodos *refatoraChavesEstrangeiras*, *refatoraViews*, *refatoraRotinas*, além de obviamente adicionar na *String* de retorno o comando *ALTER TABLE* para renomear a tabela em questão.

O método *refatoraChavesEstrangeiras* percorre todas as colunas da tabela e caso ele encontre alguma coluna que contém uma restrição de chave estrangeira, ele verifica se o nome dessa chave contém o nome da tabela (a palavra reserva). Caso contenha adiciona a *String* de retorno um comando *RENAME CONSTRAINT* para renomear a restrição.

O método *refatoraViews* percorre todas as *views* que tem relação com a tabela que será renomeada e adiciona na *String* de retorno o *CREATE OR REPLACE VIEW* para recriar a *view* substituindo a palavra reservada por uma *String* de marcação, que será alterada pelo usuário.

O método *refatoraRotinas* percorre todas as rotinas e verifica quais delas utilizam a tabela que será renomeada em sua definição. Quando o sistema encontra uma rotina que em sua definição faz alguma referência a tabela renomeada ele adiciona na *String* de retorno *CREATE OR REPLACE FUNCTION*, substituindo a palavra reservada por uma *String* de marcação.

Além de alterar a definição da *stored procedure* o sistema também verifica se o nome dela faz referência a tabela renomeada. Caso faça, ele adiciona na *String* de retorno um comando *ALTER FUNCTION* para renomear a função, trocando a palavra reservada pela *String* de marcação.

Para cada rotina alterada dessa maneira, o método *refatoraTriggers* é chamado, caso essa rotina seja disparada por uma *trigger*, esse método verifica se o nome dessa *trigger* contém a palavra reservada. Na hipótese de conter, o sistema adiciona na *String* de retorno um *ALTER TRIGGER* para renomear a *trigger*, trocando a palavra reservada por uma *String* de marcação.

O método *refatora* então, retorna uma *String* com todos os comandos *SQL* gerados, para que o usuário analise e troque o *String* de marcação utilizada para substituir a palavra reservada por um nome apropriado.

### 5.2.3 Estrutura Interna da Ferramenta

A ferramenta foi desenvolvida utilizando o modelo *MVC*, que divide o sistema em três partes: dados e regras de negócio ficam no modelo (*model*), interfaces de usuário na visualização (*view*) e a camada intermediária (*controller*) (RAMOS, 2015). De modo prático, as três partes se resumem a:

- **Model:** O *model* pode ser entendido como a camada de domínio da aplicação. Nela pode conter a lógica de negócio, persistência de dados, etc.
- **View:** A camada *view* é responsável por apresentar os dados ao usuário. No caso de aplicações PHP, a *view* é o código HTML que o PHP irá montar.
- **Controller:** A camada *controller* processa e responde a eventos, recebe alterações no *model* e atualiza a camada *view* (RAMOS, 2015).

Na camada de apresentação foi usada uma classe de *template*, desenvolvida por Rael Cunha (CUNHA, 2008), com intuito de deixar toda a estrutura visual, ou seja, todo *HTML* separado da lógica de programação. O que melhorou muito, tanto a construção quanto a manutenção do sistema.

A estrutura interna da aplicação é composta por quatro classes que refletem as tabelas do sistema. Essas classes são *BDBanco*, *BDLog*, *BDUsuario* e *BDMauCheiro*. Para cada uma dessas classes existe uma classe DAO correspondente, que são *BDBancoDAO*, *BDLogDAO*, *BDUsuarioDAO* e *BDMauCheiroDAO*.

Além das classes de modelo, o sistema tem quatro controladores, *BancoControlador*, *LogControlador*, *MauCheiroControlador* e *UsuarioControlador*.

Todas as ações que podem ser tomadas pelo usuário do sistema passam por esses controladores, onde a *URL* é composta pelo controlador, por um parâmetro *acao* e pelos parâmetros pertinentes a ação.

O *UsuarioControlador* é composto por seis ações, *novo*, *salvar*, *login*, *logar*, *logout* e *erro*. O link cadastro, presente no menu aéreo direciona o usuário para esse controlador

e para a ação novo. O usuário então, será direcionado para a view que contém o *HTML* com o formulário para cadastro. Caso o usuário esteja logado os dados virão preenchidos, para que ele possa editá-los.

Quando o usuário submeter o formulário citado acima, ele será enviado para este mesmo controlador, porém com a ação *salvar*. Essa ação pega os dados do *POST*, instancia um objeto da classe *BDUsuario*, preenche esse objeto com esses dados e chama o método *salvar* deste objeto. Esse método por sua vez, chama o método *salvar* do *bdUsuarioDAO*, que faz a inserção ou a atualização dos dados do usuário no banco.

A ação *login* é responsável por montar a tela de login, já a ação *logar* é chamada quando o formulário da tela de login é submetido. Ela pega os dados(login e senha) enviados por *POST*, e insere em um objeto da classe *BDUsuario*.

Feito isso, o método *logar* dessa classe é chamado, esse método chama o método *logar* da classe *DAO*, que procura um registro no banco com login e a senha passada como parâmetro. Caso ele encontre esse registro o *id* do usuário é colocado na sessão.

Em todas as páginas é verificado se o *id* do usuário está na sessão, caso esteja, é instanciado um objeto da classe *BDUsuario* e carregado o usuário com os dados desse *id*. A ação *logout* chama o método *logout* dessa classe, esse método destrói a sessão e redireciona o usuário para a tela de login.

A ação *erro* é chamada quando o usuário tenta acessar uma página que não é permitida para ele, ela chama a *view* de acesso restrito, que exibe uma mensagem para o usuário.

O *BancoControlador* é composto por quatro ações, *lista*, *novo*, *salvar* e *deletar*. Quando o usuário estiver logado e clicar no link Bancos, disponível no menu aéreo, ele será redirecionado para a página de listagem de bancos. Essa página é acessada através do controlador de banco e da ação *lista*.

Essa ação percorre o vetor de bancos do usuário e monta uma listagem com os bancos cadastrados pelo usuário. Esses bancos são objetos da classe *BDBanco*, eles são o espelho da tabela bancos, onde estão as informações para a conexão com o banco, utilizadas pela classe *Banco* para fazer a conexão e carregar a estrutura da base de dados que será analisada.

Quando o usuário clicar no link para adicionar um banco novo, ou para editar um banco existente ele será direcionado para o controlador *BancoControlador* com a ação novo. O usuário então irá para a *view novo*, que será responsável por montar a tela com o formulário para o cadastro ou edição de um banco. No caso de uma edição é passado como parâmetro o *id* do banco, que será utilizado pelo método *carregaBanco* do objeto da classe *BDBanco*.

Depois de preencher os campos o usuário poderá submetê-lo, sendo assim as informações serão enviadas para o controlador do banco com a ação *salvar*. Essa ação é responsável por instanciar um objeto da classe *BDBanco* e preenche-lo com os dados vindos por *POST*. Após preencher o objeto, o método *salvar* desse objeto é chamado, que por sua vez chama o método *salvar* do objeto *bdBancoDAO*.

O método *salvar* da Classe *BDBancoDAO* recebe um objeto da classe *BDBanco* como parâmetro, e verifica, por meio do método *getIdBanco* se esse objeto tem um *id*, caso não tenha ele insere os dados no banco e caso tenha ele faz uma atualização no registro com os novos dados.

A ação *deletar* instancia um objeto da classe *BDBanco* e chama do método *carregaBanco* passando como parâmetro o *id* do banco passado por *GET*. Logo em seguida, é chamado o método *deletar* desse objeto, que por sua vez chama o método *deletar* da classe *DAO* desse objeto, que faz a conexão com o banco e deleta o banco da base de

dados.

O controlador *MauCheiroControlador* é composto por oito ações, *lista*, *novo*, *edita*, *salvar*, *deletar*, *exibe*, *encontra* e *refatora*. Na ação *exibe* a *view* *exibe* é carregada, o método *lista* do objeto da classe *BDMauCheiro* é chamado, que por sua vez, chama o método *lista* do objeto da classe *BDMauCheiroDAO*, responsável por conectar no banco de dados e executar uma *query* que retorna a lista de *maus cheiros*.

Essa lista é utilizada para criar um vetor de *BDMauCheiro* que será retornado para a *view*, essa por sua vez, irá percorrer esse vetor e montar a tela com a lista de *maus cheiros* nativos do sistema e mais os *maus cheiros* cadastrados pelo usuário logado.

Quando um usuário clicar no botão *encontra* de um *mau cheiro* listado, será enviada uma requisição para o controlador *MauCheiroControlador*, com a ação *encontra* e com o *id* do *mau cheiro* selecionado.

Essa ação é a mais importante desse controlador, ela faz a mesma coisa que a ação *exibe*, porém, além de listar os *maus cheiros* nativos e cadastrados pelo usuário, ela irá carregar um objeto da classe *BDMauCheiro*, que reflete um registro da tabela *maus\_cheiros*. Um dos atributos dessa classe é *nomeClasse*, que pode ser acessado a partir do método *getNomeClasse*.

Esse atributo é o nome da classe do *mau cheiro* selecionado, onde, por meio de *reflection* será instanciado o objeto da classe do *mau cheiro* escolhido. Após instanciar um objeto do *mau cheiro*, será chamado o método *encontra* desse objeto, caso esse método encontre esse *mau cheiro* no banco de dados ele retornará uma *string* apontando em que pontos esse *mau cheiro* foi encontrado, juntamente com as sugestões de refatorações, como já foi visto na seção anterior.

A ação *refatora* desse controlador, recebe por *POST* a *String* do text área com as refatorações sugeridas pela ferramenta e alteradas (caso haja necessidade) pelo usuário.

O próximo passo é executar a refatoração, para isso, é instanciado um objeto da classe *Banco*, que contém o método *refatora*. Esse método então é chamado, ele, por sua vez, chama o método *refatora* do objeto *bancoDAO*, que faz a conexão com o banco e executa a *SQL* com as refatorações.

Caso a execução dessa *SQL* resulte em erro, esse erro é retornado e exibido para o usuário, junto com a própria *SQL*, para que ele possa alterar onde o erro ocorreu. Dessa forma ele pode reenviar a *SQL* para que seja executada novamente. O usuário receberá uma mensagem caso a *SQL* tenha sido executada com sucesso.

Independente do resultado da execução da *SQL*, o método *refatora* da classe *bancoDAO* instancia um objeto da classe *BDLog* e chama o método *salvar* dessa classe, passando como parâmetro a *SQL* executada, a data da execução e o erro(caso ele exista). Esse método chama o *salvar* do objeto *bdLogDAO*, que faz a inserção do log no banco de dados.

Quando o usuário estiver logado e clicar no link *Maus Cheiros*, disponível no menu aéreo, ele será redirecionado para a página de listagem de *maus cheiros* cadastrados por ele. Essa página é acessada através do controlador de *maus cheiros* e da ação *lista*.

Essa ação percorre o vetor de *maus cheiros* do usuário e monta uma listagem com os *maus cheiros* cadastrados pelo usuário. Esses *maus cheiros* são objetos da classe *BDMauCheiro*, eles são o espelho da tabela *mau\_cheiro*, onde está armazenado, por exemplo, o nome da classe do *mau cheiro*, que será utilizada pelo sistema para fazer a análise.

Quando o usuário clicar no link para adicionar um *mau cheiro* novo, ou para editar um existente, ele será direcionado para o controlador *MauCheiroControlador* com a ação *novo* ou *edita*. O usuário então irá para uma *view*, que será responsável por montar a tela

com o formulário para o cadastro ou edição de um *mau cheiro*. No caso de uma edição os campos do formulário virão preenchidos com os dados do *mau cheiro* passado como parâmetro.

Esse formulário é composto por três campos, a descrição do mau cheiro, o tipo de refatoração utilizada para resolver o *mau cheiro* e o campo de upload do arquivo *php* que será utilizado pela ferramenta para encontrar e sugerir as refatorações.

Para implementar a classe de um *mau cheiro* específico, usuário deve utilizar a estrutura do banco de dados disponível pela *API* da ferramenta por meio da classe *Banco*. Sendo assim, a classe desenvolvida pelo usuário deve seguir os padrões das classes dos *mau cheiros* nativos, ou seja, deve ser criada uma classe que implemente *iMauCheiro*, com o métodos *encontra* e *refatora*.

Depois de preencher os campos, o usuário poderá submetê-lo, sendo assim as informações serão enviadas para o controlador do *mau cheiro* com a ação *salvar*. Essa ação é responsável por instanciar um objeto da classe *BDMauCheiro* e preenche-lo com os dados vindos por *POST*. Após preencher o objeto, o método *salvar* desse objeto é chamado, que por sua vez chama o método *salvar* do objeto *bdMauCheiroDAO*.

Após salvar as informações na tabela *maus\_cheiros* o arquivo é enviado para a pasta *mau\_cheiro\_usuario*, que armazena todos as classes cadastradas pelos usuários.

A ação *deletar* instancia um objeto da classe *BDMauCheiro* e chama do método *carregaMauCheiro* passando como parâmetro o *id* do *mau cheiro* passado por *GET*. Logo em seguida, é chamado o método *deletar* desse objeto, que por sua vez chama o método *deletar* da classe *DAO* desse objeto, que faz a conexão com o banco e executa a *SQL* para deletar o *mau cheiro* da base de dados.

Após deletar o *mau cheiro* da base de dados, o arquivo correspondente será removido do diretório *mau\_cheiro\_usuario*. Sendo assim, a análise desse *mau cheiro* não estará mais disponível na ferramenta.

O controlador *LogControlador* é composto por duas ações, *lista* e *visualiza*. A ação *lista* é responsável por instanciar um objeto da classe *DBLog*, onde será definido o banco carregado a partir do *id* passado por *GET*. O próximo passo então, é chamar o método *lista* dessa classe, que consequentemente chama o método *lista* da classe *DAO*, esse, por sua vez faz uma consulta no banco de dados da aplicação e retorna um vetor com os logs do banco definido, que serão exibidos para o usuário.

Quando o usuário clica na opção de visualizar um log, o controlador *LogControlador* será requisitado, junto com a ação *visualiza* e o *id* do log que deve ser exibido. Essa ação chama a *view* de visualização, instancia um objeto da classe *DBLog* e chama o método *carregaLog* desse objeto, passando com parâmetro o *id* que veio por *GET*. As informações desse log são setadas no template que será exibido para o usuário.

## 6 ESTUDO DE CASO

Como estudo de caso, foi usado o banco de uma loja, como pode ser visto na Figura 6.1.

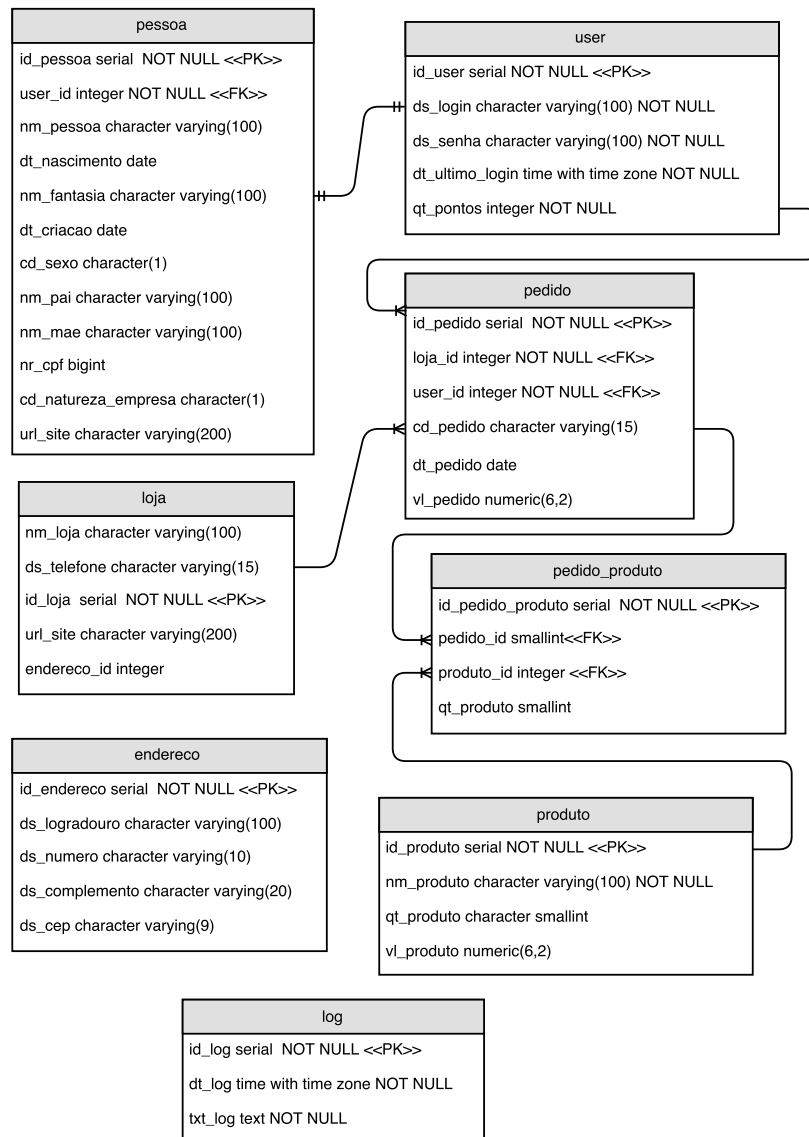


Figura 6.1: banco que será analisado pela ferramenta

Esse banco é composto pelas seguintes tabelas:



- **produto:** Essa tabela armazena as informações do produto da loja. Ela contém 189779 registros, o que torna o tamanho dela maior que o tamanho da memória do servidor informada pelo usuário. Ou seja, nela a ferramenta irá encontrar o mau cheiro de tabela com muitas linhas.
- **user:** Tabela que armazena as informações dos usuários, necessárias para fazer a autenticação deles no site da loja. O nome dessa tabela é uma palavra reservada e isso é um *mau cheiro* que deve ser encontrado e refatorado pela ferramenta.
- **pessoa:** Armazena o cadastro completo dos clientes da loja. Para cada *pessoa* deve existir um *user*, responsável apenas por armazenar as informações para autenticação no sistema, enquanto a tabela *pessoa* armazena os demais dados. Ela é uma tabela multiuso, pois armazena informações de pessoa física e pessoa jurídica. Logo, a ferramenta deve encontrar e informar esse mau cheiro dessa tabela.
- **loja:** Armazena as informações das filiais da loja, no entanto, os endereços dessas lojas são armazenados na tabela endereço e vinculados pelo campo *endereco\_id*. Porém, existe um problema nesse vínculo, pois não há restrição de chave estrangeira nesse campo. A ferramenta deve identificar esse *mau cheiro* e sugerir uma refatoração para ele. Outro *mau cheiro* encontrado nessa tabela é devido a chave primária não ser o primeiro campo da tabela.
- **pedido:** Tabela que armazena as informações do pedido, tais como o id do usuário, o id da loja, a data do pedido e outros campos. Essa tabela contém duas chaves estrangeiras, onde os campos pertencentes a essas chaves não fazem parte de nem um índice. Isso indica um mau cheiro que será encontrado e refatorado pela ferramenta.
- **pedido\_produto:**  
Essa tabela também apresenta outro mau cheiro, pois ela não contém uma chave primária. Nesse caso a ferramenta indicará este mau cheiro para o usuário.
- **endereco:** Tabela que armazena endereços, nesse caso, os endereços das lojas. Todos os campos dessa tabela, com exceção da chave primária podem ser nulos. Esse é um mau cheiro que a ferramenta pode encontrar e sugerir uma refatoração.  
Os campos *ds\_logradouro* e *ds\_cep* fazem parte de um índice, porém podem assumir um valor nulo. Isso é um mau cheiro que a ferramenta é capaz de encontrar e sugerir a refatoração.
- **Log:** Tabela criada para que fossem guardadas as informações referentes aos históricos de compras. No entanto, a ideia de utilizá-la foi abandonada e esta tabela permaneceu no banco de dados. A ferramenta é capaz de encontrar tabelas que estão vazias e que podem não estarem sendo utilizadas, como é o caso desta.

Como estudo de caso será demonstrado todos os passos desde o cadastro do usuário até a refatoração de um *mau cheiro*.

Para acessar seus bancos de dados, o usuário necessita realizar o login na aplicação, mas para isso é necessário realizar o cadastro como pode ser visto na Figura 6.2. Para isso, o usuário deve escolher um login e uma senha, de oito caracteres, contendo letras minúsculas, maiúsculas e números.

## Novo Usuário

Login

Senha

Confirma a senha

Figura 6.2: Página de cadastro do usuário

## Entrar

Login

Senha

Figura 6.3: Página de login

Após efetuar o cadastro o usuário pode efetuar o login, como pode ser visto na Figura 6.3.

Após logar no sistema o usuário será direcionado para a página com a listagem dos bancos de dados cadastrados por ele. Nesse caso não há bancos cadastrados, então para isso, o usuário deve cadastrar, clicando no ícone de adicionar banco nessa página.

Ao clicar no ícone para adicionar um novo banco, o usuário será direcionado para uma página com o formulário para cadastro de um novo banco, como pode ser visto na Figura 6.4. Nesse formulário o usuário deve informar os dados para a conexão com o banco que deve ser analisado pela ferramenta.

Além dos dados para a conexão com banco, o usuário deve informar dados que serão utilizados para encontrar alguns *maus cheiros*, esses dados são: padrão de chaves primárias, padrão de chaves estrangeiras e o tamanho da memória do servidor.

O formulário, intitulado "Novo Banco", contém os seguintes campos e botões:

- Host:** Campo de texto com o valor "localhost".
- Nome:** Campo de texto com o valor "banco\_teste".
- Usuário:** Campo de texto com o valor "usuario123".
- Senha:** Campo de texto vazio.
- Porta:** Campo de texto com o valor "5432".
- Padrão das Chaves Primárias:** Campo de texto com o valor "id\_XXXX".
- Padrão das Chaves Estrangeiras:** Campo de texto com o valor "XXXX\_id".
- Botões:** "Voltar" e "Enviar" no rodapé.

Figura 6.4: Página de cadastro de um banco de dados

Após cadastrar um banco de dados, o usuário será direcionado para a página com a listagem de seus bancos de dados, como pode ser visto na Figura 6.5.

## Lista


Host	Nome	Usuário	Porta	Editar	Apagar	Maus Cheiros	Log
localhost	loja	postgres	5432				

Figura 6.5: Página com a lista de bancos de dados do usuário


Nessa página, o usuário pode editar o banco recém-criado. Ao clicar no ícone de editar, o usuário acessará um formulário igual ao de cadastro de banco, porém este já virá com os dados do banco preenchido.

Além de editar o banco de dados, o usuário pode, clicando no ícone de excluir, apagar o banco recém-criado. Ao clicar nesse ícone a ferramenta avisará ao usuário que o banco será excluído juntamente com os seus logs.

Por meio dessa página que o usuário acessa a sessão mais importante da ferramenta, a página onde o usuário pode procurar os *maus cheiros* e refatorar. Ao clicar no ícone de *mau cheiro*, o usuário será direcionado para a página com a listagem dos *maus cheiros* que a ferramenta é capaz de encontrar.


A página de *mau cheiro*, contém a listagem dos maus cheiros nativos e dos cadastrados pelo usuário, que podem ser analisados pela ferramenta. Além disso, a página contém um text área, onde será exibido o resultado da busca pela opção selecionada. Cada opção contém um botão *encontra*, onde o usuário deve clicar para que a ferramenta procure aquele *mau cheiro* no banco de dados, como pode ser visto na Figura 6.12.

## loja em localhost




Colunas que são chaves estrangeiras, porém o tipo de dado é diferente da chave primária

Encontra



colunas que fazem parte de um índice, porém são anuláveis

Encontra



Tabelas com todos campos anuláveis

Encontra

Figura 6.6: Página com a lista de maus cheiros que a ferramenta é capaz de analisar

Ao clicar no botão *encontra*, a ferramenta fará uma análise e retornará para o text área o resultado dessa análise, conforme pode ser visto na Figura 6.7.

## loja em localhost

Colunas que são chaves estrangeiras, porém o tipo de dado é diferente da chave primária

Encontra Refatora

Resultado da busca do mau cheiro selecionado

```

1  /*Colunas que são chaves estrangeiras, porém o tipo de dado é diferente da chave primária:*/
2
3
4  /*Chave estrangeira -> Tabela: pedido_produto Coluna : pedido_id Tipo : smallint
5  Chave primária -> Tabela: pedido Coluna : id_pedido Tipo : integer
6  Refatoração:*/
7
8  ALTER TABLE "pedido_produto" ALTER COLUMN "pedido_id" TYPE integer;
9
10

```

Figura 6.7: Resultado da procura pelo primeiro mau cheiro da lista

Nesse caso, após analisar a sugestão de refatoração, o usuário poderá alterar o script sugerido, caso ache pertinente. Após validar as sugestões e alterá-las se necessário, o usuário poderá clicar no botão *refatora*, para que os comandos sejam executados no banco de dados.

Após tentar executar as refatorações a ferramenta retornará para a tela com a listagem dos *maus cheiros*, porém com uma resposta de sucesso (Figura 6.8), caso as alterações tenham sido realizadas com sucesso ou retornará uma mensagem de erro (Figura 6.9) de *SQL*. Em ambas as situações será armazenado um *log* da operação.

## loja em localhost

Refatoração realizada com sucesso.

Colunas que são chaves estrangeiras, porém o tipo de dado é diferente da chave primária

Encontra

Resultado da busca do mau cheiro selecionado

```

1
2  O banco de dados não encontrou nenhuma ocorrência da refatoração selecionada
3

```

Figura 6.8: Resultado obtido quando a SQL é executada com sucesso

## loja em localhost

ERROR: syntax error at or near "ALTR" LINE 8: ALTER TABLE "pedido\_produto" ALTR COLUMN "pedido\_id" TYPE in... ^

Colunas que são chaves estrangeiras, porém o tipo de dado é diferente da chave primária

Encontra Refatora

Resultado da busca do mau cheiro selecionado

```



1  /*Colunas que são chaves estrangeiras, porém o tipo de dado é diferente da chave primária:*/
2
3
4  /*Chave estrangeira -> Tabela: pedido_produto Coluna : pedido_id Tipo : smallint
5  Chave primária -> Tabela: pedido Coluna : id_pedido Tipo : integer
6  Refatoração:*/
7
8  ALTER TABLE "pedido_produto" ALTR COLUMN "pedido_id" TYPE integer;
9
10

```

Figura 6.9: Resultado obtido quando a SQL falha

Para consultar os logs, o usuário deve entrar na página com a listagem dos seus bancos e clicar na opção logs do banco desejado. O usuário então, será direcionado para a listagem dos logs daquele banco, como pode ser visto na Figura 6.10.

loja Log

Data	Log	Visualizar
24/05/2016 01:12:00	/*Colunas que são chaves estrangeiras, porém o tipo de dado é diferente da chave primária:*/	
24/05/2016 01:10:50	/*Colunas que são chaves estrangeiras, porém o tipo de dado é diferente da chave primária:*/	

[Voltar](#)

Figura 6.10: Lista de log

Ao clicar em um dos *logs*, o usuário irá para a página com o detalhe do *log*. Essa página contém a data do *log*, os comandos executados e o erro, caso a refatoração tenha falhado, como pode ser visto na Figura 6.11.

## Log 2016-05-24 01:10:50



**ERROR: syntax error at or near "ALTERA"**  
 LINE 7: ALTERA TABLE "pedido\_produto" ALTER COLUMN "pedido\_id" TYPE ...  
 ^



/\*Colunas que são chaves estrangeiras, porém o tipo de dado é diferente da chave primária:\*/

/\*Chave estrangeira -> Tabela: pedido\_produto Coluna : pedido\_id Tipo : smallint  
 Chave primária -> Tabela: pedido Coluna : id\_pedido Tipo : integer  
 Refatoração:\*/

ALTERA TABLE "pedido\_produto" ALTER COLUMN "pedido\_id" TYPE integer;

[Voltar](#)

Figura 6.11: Detalhe de um log

Além das funcionalidades supracitadas, a ferramenta permite também que o usuário cadastre novos *maus cheiros* identificados por ele. Para isso, o usuário deve acessar o link *maus cheiros*, no menu aéreo da ferramenta. Ao acessar este link o usuário será direcionado para uma página que contém a listagem dos *maus cheiros* cadastrados por ele, como pode ser visto na Figura 6.12.

Nessa página, o usuário pode apagar, editar ou inserir um novo *mau cheiro*. Ao clicar na opção de inserir *mau cheiro*, o usuário será direcionado para o formulário de cadastro

## Lista

+				
Classe	Descrição	Tipo	Editar	Apagar
TabelaTodosCamposAnulaveisPlus	teste	Arquitetural		

Figura 6.12: Lista dos *maus cheiros* cadastrados pelo usuário

de um novo *mau cheiro*, onde ele deve fazer o *upload* da classe implementada nos padrões da API da ferramenta, como pode ser visto na Figura 6.13.

Após submeter esse formulário com sucesso, a opção da análise desse novo *mau cheiro* estará disponível para que o usuário possa analisar os bancos cadastrados por ele, assim como as opções nativas da ferramenta.

## Novo Mau Cheiro

### Classe

```
<?php

class
TabelaTodosCamposAnula
veisPlus extends
MauCheiroTabela {
```

TabelaTodosCamposAnula...  
(3.34 KB)

Tab

Remove

Upload

Browse ...

### Descrição

teste

### Tipo de Refatoração

Arquitetural

Voltar

Enviar

Figura 6.13: Formulário para cadastro de um novo *mau cheiro*



## 7 CONCLUSÃO

Como já foi citado, um dos motivos que levou a comunidade de banco de dados deixar de adotar as metodologias ágeis foi a falta de ferramenta que desse o devido suporte às técnicas utilizadas por essas metodologias.

Conforme pode ser visto na Tabela 7.1, nenhuma das ferramentas citadas no presente trabalho contemplam a análise em busca de *maus cheiros* e a refatoração em uma única ferramenta. O SchemaCrawler, por exemplo, faz apenas a análise da base de dados em busca de *maus cheiros*, enquanto o dbForge Studio for MySQL é capaz de fazer algumas (Renomear Tabela e Renomear Coluna) refatorações e resolver as suas dependências.

Levando isso em consideração a ferramenta desenvolvida é uma contribuição a esse nicho pouco explorado, uma vez, que ela atua tanto como um *sniffer* e como uma ferramenta de refatoração.

A ferramenta desenvolvida no presente trabalho é capaz de automatizar a análise de alguns *maus cheiros* e sugerir algumas refatorações, bem como resolver as dependências dessas refatorações sugeridas, facilitando assim, o trabalho da comunidade de banco de dados, uma vez que, fazer isso de forma manual é um processo árduo e suscetível a falhas. Porém alguns *maus cheiros* e algumas alterações não podem ser sugeridas automaticamente, uma vez que, estas dependem de uma análise mais criteriosa de um ser humano especializado.

A ferramenta desenvolvida é expansível, quando se trata de *maus cheiros* que ela pode encontrar, uma vez que, por meio de uma interface amigável é possível que o usuário cadastre classes implementadas por ele, utilizando a API da ferramenta. Sendo assim, quando o usuário for analisar um bancos de dados, estarão disponíveis as opções de busca por *maus cheiros* nativas da ferramenta e as cadastradas por ele.

Conforme também pode ser visto na Tabela 7.1, a ferramenta desenvolvida é uma aplicação WEB, o que a torna mais versátil que as demais, uma vez que, ela funciona independente de plataforma, necessitando apenas de um *browser*. Outra vantagem, com relação as outras ferramentas é que ela conta com uma interface WEB intuitiva e amigável, o que não é o caso do SchemaCrawler e do Liquibase, já que nessas ferramentas é necessário escrever arquivos de configuração e executar a ferramenta por meio da linha de comando

Entretanto, a ferramenta desenvolvida é direcionada apenas para o PostgreSQL, já que ela consome informações específicas contidas nos catálogos desse SGBD, além de que as refatorações sugeridas são direcionadas para esse SGBD. Já o dbForge é para MySQL, enquanto SchemaCrawler e o Liquibase suportam todos os SGBDs que tenham um driver JDBC.

No entanto, a ferramenta desenvolvida não contempla algumas funcionalidades importantes, como por exemplo, o controle de versão, disponível no Liquibase. O controle

de versão de banco de dados é uma prática muito importante, principalmente quando desenvolvido por equipes que utilizam metodologias ágeis, uma vez que a base de dados está em constante mudança por meio das refatorações.

Outra funcionalidade importante, mas não contemplada pela ferramenta é a alteração das aplicações acopladas a base de dados onde a ferramenta aplicará as refatorações, ou seja, a ferramenta não ajusta as aplicações dependentes do banco de dados refatorado. Sendo assim, é necessário que o desenvolvedor ajuste essas aplicações manualmente.

Tabela 7.1: Tabela de comparação entre os sistemas existentes e a ferramenta desenvolvida no presente trabalho

	SchemaCrawler	dbForge Studio for MySQL	Liquibase	TADS Refactor
Plataforma	Todas que suportam Java 8 ou superior	Windows	Todas que suportam Java 1.5 ou superior	WEB
SGBD	Todos que tem um driver JDBC	MYSQL	Todos que tem um driver JDBC	PostgreSQL
Encontra <i>data-base smells</i>	Sim	Não	Não	Sim
Refatora	Não	Sim	Sim	Sim
Resolve dependências automaticamente	Não	Sim	Não, o desenvolvedor tem que escrever o arquivo de configuração para resolvê-las	Sim
Usabilidade	Ruim, configuração em XML e execução em terminal	Boa, interface amigável	Ruim, configuração em XML e execução em terminal	Boa, interface intuitiva e amigável
Expansível	Sim	Não	Não	Sim
Suporte ao processo de refatoração(Rollback e versionamento)	Não	Não, mas oferece algumas opções de suporte(Comparação e sincronização de banco)	Sim, por ser uma ferramenta de versionamento, faz o controle das alterações/refatorações	Não

Além disso, o sistema foi desenvolvido como uma atividade de ensino-aprendizagem, que emprega várias tecnologias, servindo como um laboratório para estas e seu emprego em sistemas reais, vinculando o aprendizado ao mundo real. Por fim, é uma contribuição tanto da Instituição de Ensino quanto do aluno à comunidade em que estão inseridos, abrindo precedentes para uma área pouco explorada.

## 8 TRABALHOS FUTUROS

Como a ferramenta não refatora todos *maus cheiros* encontrados, uma das próximas atividades seria desenvolver a refatoração para os *maus cheiros* que isso fosse possível. Além disso, podem ser desenvolvidas soluções para encontrar outros *maus cheiros*.

Outra melhoria significativa seria com relação a portabilidade, já que a ferramenta suporta apenas o SGBD PostgreSQL. Sendo assim, a ferramenta poderia ser adaptada para suportar outros SGBDs como MySQL, Oracle, SQL Server, entre outros.

Como a ferramenta pode acabar aplicando muitas alterações no banco de dados, é interessante que essas alterações sejam passíveis de serem revertidas, de forma harmônica e garantindo a integridade das informações. Sendo assim, seria importante a ferramenta fazer um controle de versão.

Considerando que o banco de dados é uma estrutura na maioria das vezes fortemente acoplada, seria interessante desenvolver uma solução que refatorasse não só o banco, mas também as aplicações que consomem as informações dessa estrutura.

Como já foi citado anteriormente, a ferramenta foi desenvolvida para refatorar um banco em um ambiente simples, onde apenas uma aplicação consome as informações desse banco de dados. Sendo assim, não há necessidade de existir um período de transição, porém, mesmo assim, também seria interessante a ferramenta encontrar e refatorar esta aplicação.

Com o intuito de aperfeiçoar mais a ferramenta poderia ser desenvolvido uma solução que contemplasse os ambientes multi-aplicação, nesse caso a principal diferença é o fato de que haveria um período de depreciação. Sendo assim, a ferramenta aplicaria a refatoração, mas manteria também a estrutura existente durante um determinado período, para que os desenvolvedores atualizassem as aplicações com intuito de adequá-las a nova estrutura.

## REFERÊNCIAS

AMBLER, S. W.; SADALAGE, P. J. **Refactoring Databases**: evolutionary database design. [S.l.]: Addison-Wesley, 2006.

BECK, K.; FOWLER, M. **Refactoring**: improving the design of existing code. Boston, MA, USA: Addison-Wesley, 1999.

CASTRO, V. A. **DEVMEDIA**. Refatoração e TDD (Feitos um para o outro) . Disponível em: <<http://www.devmedia.com.br/refatoracao-e-tdd-feitos-um-para-o-outro/4444>> . Acesso em: 05 out. 2015.

CUNHA, R. **Tutorial de Templates em PHP**. Tutorial de Templates em PHP Disponível em: <<http://raelcunha.com/template/>> . Acesso em: 30 abr. 2016.

DBFORGE Studio for MySQL. The Best MySQL GUI Tool You Can Find. Disponível em: <<http://www.devart.com/dbforge/mysql/studio/>> . Acesso em: 19 out. 2015.

DEVMEDIA. Refatoração em Banco de Dados. **SQL Magazine**, [S.l.], n.84, 2011. Refatoração em Banco de Dados. Disponível em: <<http://www.devmedia.com.br/refatoracao-em-banco-de-dados-sql-magazine-84/19065>>. Acesso em: 15 out. 2015.

DEVMEDIA. Refatoração aplicadas a bancos de dados. **Revista Engenharia de Software**, [S.l.], n.46, 2012. Refatoração aplicadas a bancos de dados. Disponível em:<<http://www.devmedia.com.br/refatoracao-aplicadas-a-bancos-de-dados-revista-engenharia-de-software-magazine-46/23815>>. Acesso em: 20 out. 2015.

DOMINGUES, M. B. P.; ALMEIDA JR., J. R. de; PALETTA, F. C. Modelagem De Processos De Refatoração De Banco De Dados Utilizando BPMNs. In: CONTECSI, CONFERÊNCIA INTERNACIONAL SOBRE SISTEMAS DE INFORMAÇÃO E GESTÃO DE TECNOLOGIA, 12., 2015, São Paulo, Brasil. **Anais...** [S.l.: s.n.], 2015.

FATEHI, S. **SchemaCrawler**. SchemaCrawler: Free database schema discovery and comprehension tool. Disponível em: <<http://sualeh.github.io/SchemaCrawler/>> . Acesso em: 21 out. 2015.

GROUP, T. P. G. D. **Chapter 44. System Catalogs**. Chapter 44. System Catalogs. Disponível em: <<https://www.postgresql.org/docs/8.4/static/catalogs.html>> . Acesso em: 10 abr. 2016.

GROUP, T. P. G. D. **Chapter 34. The Information Schema**. Chapter 34. The Information Schema. Disponível em: <<https://www.postgresql.org/docs/9.1/static/information-schema.html>> . Acesso em: 12 abr. 2016.

JÚNIOR, L. F. **Técnicas Evolucionárias para Refatoração em Banco de Dados**. 2013. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal De Santa Maria, Santa Maria, RS, Brasil.

KATAOKA, Y.; ERNST, M. D.; GRISWOLD, W. G.; NOTKIN, D. Automated support for program refactoring using invariants. In: ICSM 2001, PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 2001, Florence, Italy. **Anais...** [S.l.: s.n.], 2001. p.736–743.

LÂMPADA, T. **Evoluindo o banco de dados com o liquibase**. Evoluindo o banco de dados com o liquibase. Disponível em: <<https://dicasdolampada.wordpress.com/2012/03/19/evoluindo-o-banco-de-dados-com-o-liquibase/>> . Acesso em: 27 out. 2015.

PAPOTTI, P. E. **Liquibase**: gerenciamento de alterações em banco de dados. Liquibase: Gerenciamento de Alterações em Banco de Dado. Disponível em: <[http://www2.dc.ufscar.br/paulo.papotti/material\\_liquibase.pdf](http://www2.dc.ufscar.br/paulo.papotti/material_liquibase.pdf)> . Acesso em: 25 out. 2015.

PRITCHARD, G. **Pragmatic Data Warehousing**. Refactoring Legacy Databases. Disponível em: <<http://pragmaticdatawarehousing.blogspot.com.br/2013/02/refactoring-legacy-databases.html>> . Acesso em: 03 out. 2015.

RAMOS, A. **MVC ? Afinal, é o quê ?** MVC ? Afinal, é o quê ? Disponível em: <<http://tableless.com.br/mvc-afinal-e-o-que/>> . Acesso em: 25 abr. 2016.

SAAB, F. **DB1 Labs**. Refatorar seu código é uma boa prática! . Disponível em: <<https://db1labs.wordpress.com/2011/11/17/refatorar-seu-codigo-e-uma-boapratica-parte-1/>> . Acesso em: 05 out. 2015.

SILVA, R. F. **Refatoração de código**. Refatoração de código. Disponível em: <<http://ramonsilva.net/2015/06/10/refatoracao-de-codigo/>> . Acesso em: 06 out. 2015.

SOARES, I. Z. Refactoring de Banco de Dados. **SQL Magazine**, [S.l.], n.81, 2010. Refactoring de Banco de Dados. Disponível em: <<http://www.devmedia.com.br/refactoring-de-banco-de-dados-sql-magazine-81/18429>>. Acesso em: 10 out. 2015.

SOMMERVILLE, I. **ENGENHARIA DE SOFTWARE**. [S.l.]: PEARSON BRASIL, 2011.

TELLES, F. **Particionamento de Tabelas no postgres ? Quando?** Particionamento de Tabelas no postgres ? Quando? . Disponível em: <<http://savepoint.blog.br/particionamento-de-tabelas-no-postgres-quando/>> . Acesso em: 07 out. 2015.

TORRES, M. **O Programador Profissional**. Rio Grande, RS, Brasil: Learnpub, 2016.

## FOLHA DE APROVAÇÃO

Trabalho de conclusão de curso sob o título *Uma ferramenta para Refatoração em Bancos de Dados PostgreSQL*, defendida por Leandro Souza Marques e aprovada em 6 de julho de 2016, em Rio Grande, estado do Rio Grande do Sul, pela banca examinadora constituída pelos professores:

---

Prof. Msc. Igor Pereira  
Orientador  
IFRS - Rio Grande

---

Prof. Esp. Márcio Torres  
IFRS - Rio Grande

---

Prof. Dr. Tiago Telecken  
IFRS - Rio Grande