UNIVERSIDADE FEDERAL DE SERGIPE
CAMPUS ALBERTO CARVALHO
DEPARTAMENTO DE SISTEMAS DE INFORMAÇÃO

JULI KELLE GÓIS COSTA

UM *FRAMEWORK* PARA GERAÇÃO AUTOMÁTICA DE CARGAS SQL EM AMBIENTES *DATA WAREHOUSE*

ITABAIANA
2013

UNIVERSIDADE FEDERAL DE SERGIPE
CAMPUS ALBERTO CARVALHO
DEPARTAMENTO DE SISTEMAS DE INFORMAÇÃO

JULI KELLE GÓIS COSTA

UM *FRAMEWORK* PARA GERAÇÃO AUTOMÁTICA DE CARGAS SQL EM AMBIENTES *DATA WAREHOUSE*

Trabalho de Conclusão de Curso submetido ao Departamento de Sistemas de Informação da Universidade Federal de Sergipe, como requisito parcial para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Dr. METHANIAS COLAÇO RODRIGUES JÚNIOR
Coorientador: Msc. ANDRÉ VINÍCIUS RODRIGUES PASSOS NASCIMENTO

ITABAIANA
2013

Góis Costa, Juli Kelle.

Um *Framework* Para Geração Automática de Cargas Sql dm Ambientes *Data Warehouse* / Juli Kelle Góis Costa – Itabaiana : UFS, 2013. 57f.

Trabalho de Conclusão de Curso em Bacharel em Sistemas de Informação – Universidade Federal de Sergipe, Curso de Sistemas de Informação, 2013.

1. *Framework.* 2. *Data Warehouse*. 3. Sistemas de Informação.

I. Um *Framework* Para Geração Automática De Cargas Sql Em Ambientes *Data Warehouse.*

JULI KELLE GÓIS COSTA

UM *FRAMEWORK* PARA GERAÇÃO AUTOMÁTICA DE CARGAS SQL EM AMBIENTES *DATA WAREHOUSE*

Trabalho de Conclusão de Curso submetido ao corpo docente do Departamento de Sistemas de Informação da Universidade Federal de Sergipe (DSIITA/UFS) como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Itabaiana, 16 de abril de 2013.

BANCA EXAMINADORA:

_____

Prof. Methanias Colaço Rodrigues Júnior, Doutor
Orientador
DSIITA/UFS

_____

Prof. André Vinícius Rodrigues Passos Nascimento, Mestre
Coorientador
DSIITA/UFS

_____

Prof. Eugênio Rubens Cardoso Braz, Doutor
DSIITA/UFS

AGRADECIMENTOS

Sozinha seria impossível contabilizar mais essa vitória em minha vida. Então, hoje agradeço a todos que contribuíram de forma direta ou indireta para esta conquista.

Agradeço primeiramente a Deus, por permitir minha existência e pela proteção durante todos os momentos de lutas e vitórias.

Aos mestres-amigos, pela dedicação, compromisso e que, além de transmitirem informações e gerarem conhecimento, também foram nossos companheiros. Em especial aos que me "aturaram" por mais tempo, meu orientador, Methanias Colaço R. Jr. e meu coorientador André Vinícius, pela dedicação e paciência, obrigada por tudo. E também ao Prof. Eugênio pela disposição em colaborar com o projeto.

Aos meus pais, Inalda e Francisco, pelo precioso dom da vida, pelos ensinamentos de dignidade, e por revestir minha existência de amor, carinho e dedicação, amo vocês.

À minha irmã, Juliana, pelo companheirismo, união, palavras de conforto e amizade, estando sempre presente nos momentos difíceis, te amo.

A todos os amigos universitários, que compartilharam dos melhores momentos, da troca de ideias e experiências, bem como dos momentos de tensão e ansiedade, diante de alguns momentos, como as avaliações. Em especial ao meu grande amigo Igor Peterson, meu parceiro de Projeto de Pesquisa, de Itatech Jr., de trabalhos, de ideias, enfim, foi meu cumplice de pensamentos idênticos, sempre me entendeu. Damião com quem também firmei parceria em trabalhos, mesmo sendo implicante, houve uma grande colaboração bilateral e troca de conhecimentos, no final, sempre nos entendemos bem. Também a Lucas e Jéssica, os quais houve uma menor interação, mas que também tiveram uma grande participação e sempre dispostos em colaborar. Obrigada pelo apoio e amizade, terão sempre um lugar guardado no meu coração.

A todos os meus amigos, pela compreensão nos momentos difíceis, apoio e alegrias que passamos nesta jornada. Em especial a Janisson e Aédson, pela força durante muitos momentos difíceis desta trajetória e pelas palavras de conforto, sinceridade e autoestima, adoro vocês.

E finalmente, a todos que cruzaram meu caminho, de forma direta ou indireta, e auxiliaram-me, de alguma forma, a tornar-me uma pessoa cada vez melhor.

Meus sinceros agradecimentos!

“É exatamente disso que a vida é feita: De momentos!
Momentos os quais temos que passar, sendo bons ou não, para
o nosso próprio aprendizado, por algum motivo. Nunca
esquecendo do mais importante: Nada na vida é por acaso...”

(Chico Xavier)

COSTA, Juli Kelle Góis. **Um *Framework* para Geração Automática de Cargas Sql em Ambientes *Data Warehouse*.** 2013. Trabalho de Conclusão de Curso – Curso de Sistemas de Informação, Departamento de Sistemas de Informação, Universidade Federal de Sergipe, Itabaiana, 2011.

RESUMO

O Data Warehouse (DW) representa um banco de dados histórico, o qual encontra-se separado do ambiente de produção, concebido para armazenar dados extraídos deste ambiente. Mas, antes de serem armazenados no DW, os dados passam por uma área intermediária, a Área de Staging. O processo de carga de um ambiente para outro geralmente é codificado manualmente em SQL. A utilização de uma ferramenta que gere código de forma automática para estes casos pode substituir a codificação manual. Além disto, é interessante que sejam gerados códigos para diferentes dialetos da linguagem SQL, a fim de satisfazer à variedade de soluções apresentada pelo mercado de Banco de Dados. Este trabalho apresenta o processo de desenvolvimento de um framework que captura as características comuns entre os Sistemas Gerenciadores de Banco de Dados, facilitando a geração de procedimentos de cargas para diferentes ambientes. A fim de validar a viabilidade do mesmo, foram criadas classes que estenderam o framework para o Oracle e para o SQL Server. Os resultados indicaram viabilidade para o processo de geração automática de procedimentos SQL em diferentes dialetos.

Palavras-chave: *Data Warehouse*, *Framework*, ETL, Geração Automática de Código SQL.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1. INTRODUÇÃO

Atualmente, para auxiliar as áreas estratégicas das organizações, são criados grandes bases de dados nomeadas de *Data Warehouses*. Um *Data Warehouse* (DW) representa um banco de dados histórico, separado lógica e fisicamente do ambiente de produção de uma organização, concebido para armazenar dados extraídos deste ambiente. Antes de serem armazenados no DW, os dados são selecionados, integrados e organizados para que possam ser acessados da forma mais eficiente, auxiliando assim o processo de tomada de decisões (Kimball, 2008) (Inmon, 2005) (Colaço, 2004).

Embora o *Data Warehouse* seja um ótimo recurso para as empresas, estas encontram obstáculos tais como os problemas de qualidade dos dados em vários estágios, especialmente no estágio de carga de um *Data Warehouse* (Singh & Singh, 2010). Para efetuar esta carga, é necessário que os dados passem primeiramente por uma área intermediária, conhecida como Área de *Staging* (Kimball, 2008) (Colaço, 2004). Essa área intermediária representa a base do *Data Warehouse*, pois é na Área de *Staging* que ocorrem as principais transformações, integrações e tratamento dos dados provenientes do ambiente transacional.

As rotinas de povoamento de tabelas do DW (chamadas de dimensões e fatos), desenvolvidas na última fase do processo de carga desse ambiente, são, geralmente, codificadas manualmente em SQL com o objetivo de capturar os tratamentos específicos que devem ser dados a esses tipos de tabelas. A utilização de uma ferramenta para geração automática de código SQL pode substituir a codificação manual, aumentando a produtividade e reduzindo o número de erros de codificação, contribuindo dessa forma para uma melhor qualidade dos dados.

A fim de satisfazer à variedade de soluções apresentada pelo mercado de Banco de Dados, é interessante ter ferramentas que deem suporte a diferentes plataformas, gerando então, código para vários dialetos da linguagem SQL. Para facilitar sua criação, processos e códigos já desenvolvidos para um dialeto podem ser reutilizados para outros, pois, possuirão características e funcionalidades comuns entre eles.

A existência de aplicações pertencentes a um mesmo domínio de problemas caracteriza um ambiente propício à criação de um *framework*, já que, este tem o objetivo de

capturar as funcionalidades comuns entre as aplicações de um determinado domínio, assumindo o controle do fluxo de execução, cabendo ao desenvolvedor escrever seus pontos de extensão. Ou, em outras palavras: o *framework* é um componente de software reutilizável que possibilita, além da reutilização de código já implementado, o reuso das fases de análise e projeto. Nele, são incorporadas funcionalidades comuns a várias aplicações (Mendonça, 2002).

Este trabalho estende um projeto de pesquisa, o qual tem como objetivo o desenvolvimento de uma ferramenta de geração automática de código para ambientes de apoio à decisão. Neste contexto, será desenvolvido um *framework* que capture as características comuns de dois Sistemas Gerenciadores de Bancos de Dados, gerando automaticamente código SQL para os mesmos.

1.2 Motivação

A codificação manual de procedimentos que façam carga da Área de *Staging* para o DW pode ser demorada e ter uma má qualidade, então é interessante o aumento de desempenho com a geração automática deste código. Porém, é essencial também que exista a possibilidade de geração deste código para vários dialetos da linguagem SQL, a fim de atender um maior número de situações e reaproveitar código escrito com qualidade.

Esse ambiente singular de carga tem motivado pesquisadores, tais como a autora e orientadores deste trabalho, a encontrar soluções para reutilização em dois níveis: tanto do ponto de vista dos procedimentos SQL, como do ponto de vista do código das ferramentas criadas para geração automática destes procedimentos.

1.3 Trabalhos Relacionados

Alguns trabalhos já abordaram a reutilização de software como um fator de qualidade e redução no tempo de desenvolvimento. Por exemplo, Mendonça (2002) apresenta um estudo nesta área, abrangendo aspectos de reutilização de software, padrões de projeto e a descrição de um processo de desenvolvimento de *frameworks*.

Em (Silva, Times & Kwakye, 2012), é apresentado um *framework* ETL, o FramETL, programável e extensível, que oferece uma plataforma para projetar distintas aplicações ETL, especializando processos em um ambiente integrado.

Já em (Thomsen & Pedersen, 2009), é proposto um *framework* para o desenvolvimento de processos ETL utilizando programação ao invés de ferramentas com interface gráfica. O *framework* é baseado em python. O mesmo apresenta procedimentos prontos para alguns processos bastante conhecidos como a carga de dimensões e fatos. Os tipos de dimensões suportados são 1 e 2 apenas.

Este trabalho segue a linha da pesquisa feita por Mendonça (2002), no que diz respeito à abordagem do processo de desenvolvimento de um *framework*, mas, neste caso, acontecerá o desenvolvimento de um *framework* para área de Inteligência Aplicada aos Negócios. O mesmo auxiliará o desenvolvimento de processos de carga para *Data Warehouses*, gerando código SQL automático. Diferindo-se também da abordagem de (Silva, Times & Kwakye, 2012), a qual se trata de um processo genérico para modelagem de processos ETL, não citando as especificidades que são aplicadas no mesmo, ou seja, não mostra presença de código SQL, para ajudar neste processo. Enquanto que o presente trabalho se trata de um *framework* para geração de código SQL para cargas.

Por fim, esta pesquisa traz um incremento ao trabalho descrito em (Thomsen & Pedersen, 2009), cujas cargas para dimensões suportam apenas os tipos 1 e 2. O trabalho aqui apresentado gera cargas para os tipos 1, 2 e 3.

1.4 Justificativa

Este trabalho estende um projeto de pesquisa sobre desenvolvimento e avaliação de uma Ferramenta de Geração Automática de Código para Ambientes de Apoio à Decisão, FGCod (Santos et al., 2012). Será desenvolvido um *framework* que dê suporte às ferramentas de geração automática de código SQL, exigindo conhecimentos adquiridos em disciplinas como: Programação Orientada a Objetos, Engenharia de Software, Sistemas de Apoio à Decisão e Banco de Dados.

1.5 Objetivos do trabalho

1.5.1 Objetivo Geral

O objetivo deste projeto é desenvolver um *framework* de suporte a ferramentas de geração de código automática para ambientes de apoio à decisão, mais especificamente para geração de procedimentos SQL, que farão a carga de dados da Área de *Staging* para o DW.

1.5.2 Objetivos Específicos

- Realizar pesquisa bibliográfica sobre *Data Warehouse* – rotinas de povoamento – e *framework*;
- Fazer uma análise de Domínio – identificar as classes e objetos que são comuns para todas as aplicações, dentro do domínio estudado;
- Realizar o levantamento de Requisitos e Análise;
- Elaborar Projeto do *Framework*;
- Implementar o *framework*, que será a correspondência à codificação e "materialização" de tudo o que foi documentado;
- Testar o *framework*, utilizando, pelo menos, dois Sistemas Gerenciadores de Banco de Dados.

1.6 Organização do Trabalho

A forma como o trabalho está organizado será descrita abaixo.

No Capítulo 2, é apresentada a revisão bibliográfica e os conceitos necessários para a realização do trabalho. O mesmo inicia descrevendo a Arquitetura de um *Data Warehouse* e é finalizado com o processo de desenvolvimento de um *framework.*

No Capítulo 3, é apresentado todo o processo de desenvolvimento do *framework.* São abordados a análise de domínio, levantamento de requisitos e o projeto do *framework*, o qual apresenta o projeto arquitetural, diagrama de classes e diagrama de sequencia.

Já o Capítulo 4 descreve o Estudo de Caso, definição de seu objetivo, planejamento para execução e validação dos resultados.

Finalmente, no Capítulo 5, serão expostas as conclusões sobre o trabalho realizado e os possíveis trabalhos futuros relacionados.

2. REVISÃO BIBLIOGRÁFICA E CONCEITOS RELEVANTES AO TRABALHO

2.1 Arquitetura de *Data Warehouse*

Kimball define um *Data Warehouse* (DW) como "uma cópia das transações de dados especificamente estruturada para consultas e análises" (Kimball, 2008). Inmon, a quem é creditado o termo DW, dá uma definição mais detalhada. Ele diz que o DW é uma coleção de dados orientada por assunto, integrada, não-volátil, variante no tempo, que dá apoio às decisões da administração (Inmon, 2005). Mas, as duas definições enfatizam a análise de dados e suporte à tomada de decisões gerencias, utilizando o histórico de dados presente em um DW.

Na Figura 1 é ilustrada a arquitetura genérica de um ambiente de *Data Warehouse*. Nela, podemos ver os principais componentes da arquitetura: o ambiente OLTP, a Área de *Staging* e o *Data Warehouse*.

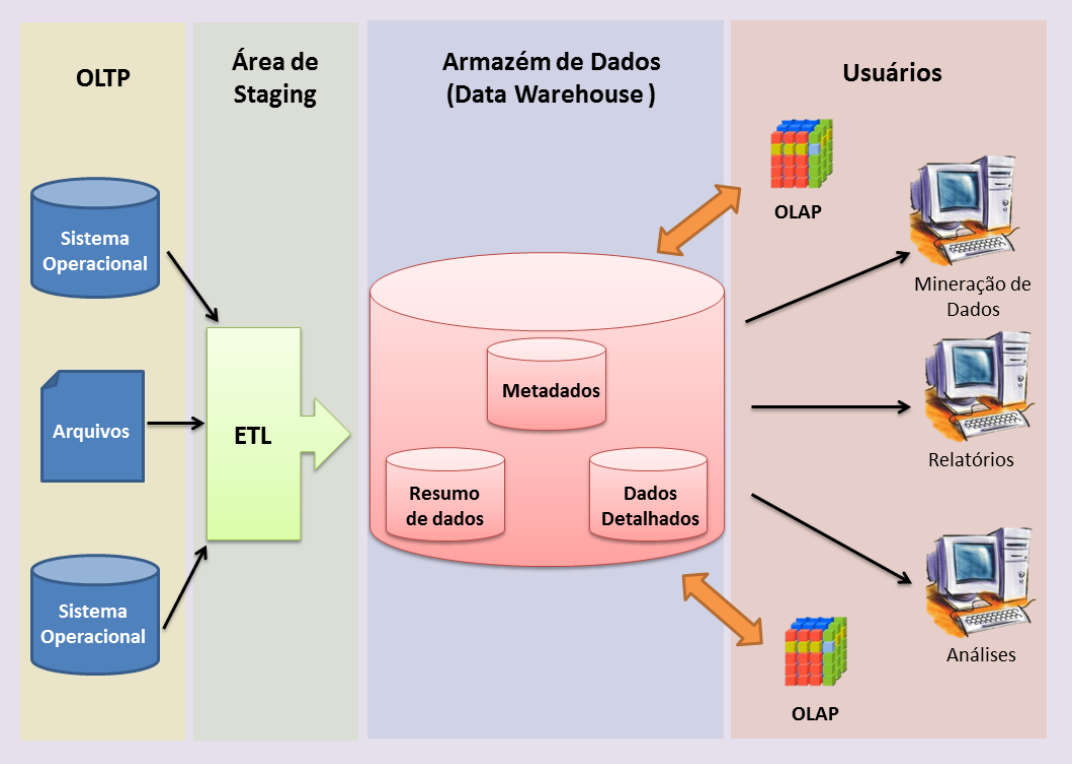

Figura 1: Arquitetura de um DW. Adaptada de (Kimball, 2002).

O ambiente que representa os sistemas envolvidos nos processos diários das organizações é conhecido como **OLTP** (Online Transaction Processing, em português, Processamento de Transações em Tempo Real). Esses sistemas representam as fontes de informações para o *Data Warehouse*. Antes de serem carregados para as estruturas finais e disponibilizados para os tomadores de decisões, os dados operacionais passam por um processo conhecido como ETL (Acrônimo de Extract-Transform-Load), responsável pelas principais transformações, integrações e tratamento dos dados provenientes do ambiente transacional. Esse processo ocorre em uma área intermediária conhecida como **Área de *Staging*** (Kimball, 2008) (Colaço, 2004). Essa área intermediária representa a base do DW. A fase final do processo de ETL consiste na carga ou povoamento dos esquemas de dados do *Data Warehouse* (Santos et al., 2012).

Depois que os dados já estiverem carregados no DW, estes serão submetidos a análises e manipulações através de ferramentas OLAP (On-line Analytical Processing). Estas ferramentas são utilizadas por gestores em qualquer nível da organização, permitindo-lhes análises comparativas que facilitam à tomada de decisões, pois, a partir destas, é possível realizar mineração de dados, análise estratégica e elaboração de relatórios.

2.1.1 Modelagem Dimensional

A Modelagem Dimensional (MD) é a técnica de projeto lógico de banco de dados mais recomendada para aplicações de suporte à decisão. Os esquemas criados com essa técnica contêm a mesma informação que os esquemas derivados de um DER (Diagrama Entidade Relacionamento), mas, agrupam os dados em uma forma que melhora a clareza do usuário e desempenho das consultas através da diminuição do grau de normalização. Os esquemas de dados criados com a Modelagem Dimensional recebem o nome de Esquema Estrela (Kimball, 2008).

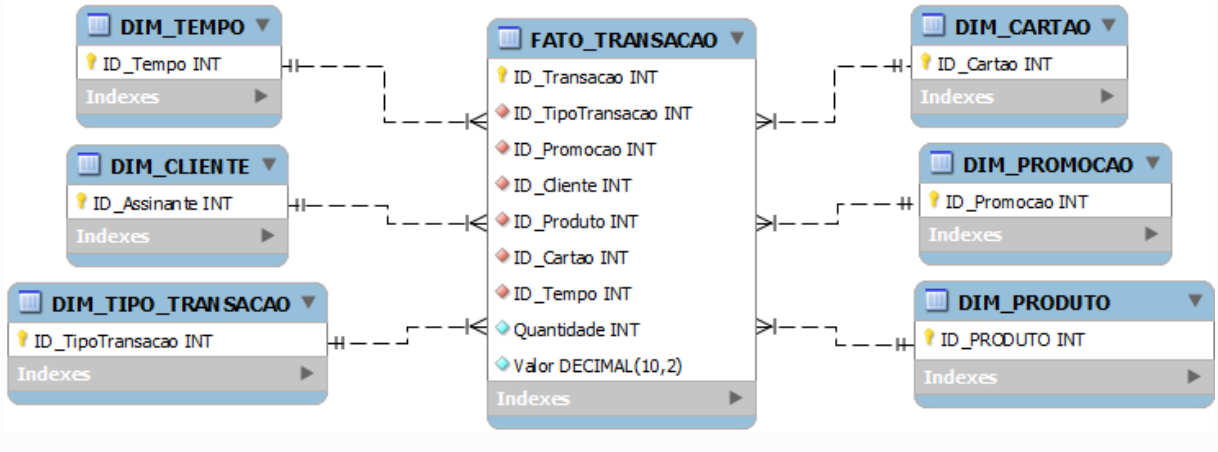

Figura 2: Exemplo de um Esquema Estrela Produzido através da Modelagem Dimensional. (Santos et al., 2012).

Na Figura 2 é ilustrado um exemplo de um esquema estrela. Um esquema estrela consiste de uma tabela principal, denominada tabela de fatos, e de tabelas auxiliares conhecidas como dimensões. As tabelas de fatos são responsáveis por registrar as transações, ocorrências ou fatos de um processo de negócio e as dimensões representam os elementos que irão contextualizar os fatos registrados. É na fase de povoamento do processo ETL que as tabelas de fatos e dimensões são povoadas. As rotinas responsáveis por essa fase são as principais responsáveis pelo tratamento e armazenamento correto do histórico de dados dentro de um ambiente de suporte à decisão (Santos et al., 2012). Nas próximas seções, são apresentadas as principais características dos fatos e dimensões.

2.1.2 Tabela de Fatos

São nas tabelas de fatos que são guardadas as medidas numéricas mais importantes do processo de negócio sendo modelado. Elas são as principais tabelas em um esquema estrela (Kimball, 2008). As tabelas de fato armazenam grande quantidade de dados históricos em função do tempo, obtidos a partir da intersecção de todas as dimensões da estrela (Colaço, 2004). As dimensões são ligadas aos fatos através das suas Surrogate Keys (chaves primárias).

As tabelas de fatos possuem uma propriedade conhecida como grão ou granularidade. A granularidade de uma tabela de fatos representa o nível de detalhe associado com as medidas armazenadas. Ao centro da Figura 2 é ilustrado um exemplo de tabela fatos,

FATO_TRANSACAO, cuja granularidade é a venda de um produto a um cliente em determinado dia. Uma tabela de fatos é composta por dois tipos de atributos: a) Os atributos que representam as ligações com as dimensões; b) Os atributos que representam as medidas do negócio. Na tabela de fatos da Figura 2, por exemplo, temos os seguintes atributos que representam as ligações com as dimensões: Id_TipoTransacao, Id_Promocao, Id_Tempo, Id_Cliente, Id_Produto, Id_Cartao. Os atributos Quantidade e Valor representam as medidas para o fato registrado.

2.1.3. Dimensões

As dimensões são a principal fonte de informações e restrições no processo de análise. Por isso, seus atributos devem ser valores descritivos (Imhoff, Galemmo e Geiger 2003). As dimensões representam tabelas cujos atributos são utilizados para contextualizar, restringir e agrupar consultas direcionadas para tabelas de fatos dentro de um esquema dimensional (Santos et al., 2012).

As tabelas de dimensões são compostas basicamente por colunas, nas quais são contidos elementos textuais que descrevem o negócio, e uma chave primária, na forma de uma Surrogate Key. Outras colunas podem surgir como padrão para compor as tabelas de dimensões, a depender da abordagem para tratamento de histórico (Santos et al., 2012). A Figura 3 ilustra um exemplo de uma dimensão. Nela, podemos identificar o atributo que representa a chave primária, Id_cliente, e os atributos que descrevem as características do cliente modelado.

A Figura 3 representa um exemplo de tabela dimensão de cliente.

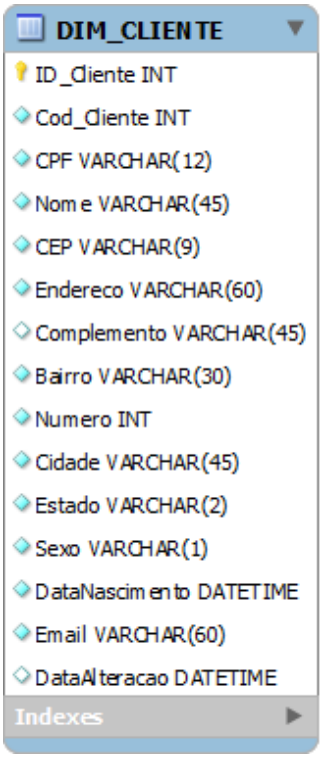

Figura 3: Exemplo de Tabela Dimensão (Tabela Dimensão Cliente).

2.1.3.1 Tratamento de Histórico em Dimensões

As mudanças dos valores dos atributos das dimensões podem ser relevantes para a análise de dados e tomada de decisões. A maneira como essas mudanças são tratadas pelo esquema de dados do *Data Warehouse* determina o tratamento de histórico que será dado para cada atributo da dimensão. Os atributos de uma dimensão podem ser classificados, em função do tratamento de histórico, nos seguintes tipos: Tipo 0, Tipo 1, Tipo 2 e Tipo 3 (Santos e Bello, 2011). Embora a literatura (Kimball, 2002) apresente outros tipos de dimensões, Tipo 4 e Tipo 6, esses últimos são apenas variações envolvendo os três primeiros tipos.

2.1.3.1.1 Tipo 0

Quando um atributo tem seu comportamento, em relação ao histórico, classificado como Tipo 0 significa dizer que o atributo armazena sempre o valor original. Em outras palavras, será descartada qualquer mudança que possa ocorrer com o atributo (Santos e Belo, 2011).

2.1.3.1.2 Tipo 1

Não há armazenamento de histórico para atributos da dimensão do Tipo 1. O comportamento do atributo, para este tipo, sobrescreve o valor antigo de uma linha da dimensão pelo valor atual proveniente da Área de *Staging*. Dessa forma, o atributo sempre irá refletir o valor mais recente proveniente do ambiente operacional. Ao serem identificados novos valores, que ainda não constam na dimensão, o procedimento cria um novo registro no DW.

2.1.3.1.3 Tipo 2

O Tipo 2, ao contrário do comportamento do tipo anterior, é utilizado quando há a necessidade de armazenar histórico dos atributos da dimensão. O tratamento de histórico é realizado através da criação de novos registros. É gerado um novo valor do identificador da dimensão (Surrogate Key) quando há alteração em atributos que necessitam armazenar todo o seu histórico.

No esquema da dimensão que possui atributos do Tipo 2 surgem novos atributos que não existem na dimensão que contempla somente atributos do Tipo 1. Esses novos atributos são utilizados para o tratamento dado ao histórico nesse tipo de dimensão. São eles: a data inicial, que representa a data em que o registro foi gravado; a data final, que representa a data em que o registro deixou de ser corrente; e por fim, o atributo que identifica se é um registro atual ou não (Santos et al., 2012).

2.1.3.1.4 Tipo 3

O tratamento de histórico do Tipo 3 é utilizado quando há uma mudança e existe a necessidade de manter o histórico sem a criação de um novo registro. Essa característica é alcançada através do armazenamento do histórico em vários atributos de um mesmo registro de dados (Santos et al., 2012). Logo, a partir da necessidade, podem ser criadas quantas colunas forem desejadas para a dimensão.

2.2 Desenvolvimento de um *Framework*

O processo de desenvolvimento de software é difícil, lento e caro. E ainda não se tem tecnologia para fazer software grande do zero, rapidamente e com poucos bugs. Estes são os problemas apontados por Sauvé para construir um software (Sauvé, 2002). Para facilitar este processo, o caminho frequentemente mais apontado é o reuso de software.

De acordo com (Rumbaugh et al., 1994 apud Mendonça, 2002), o projeto de software reutilizável reduz o custo do projeto, da programação e dos testes. Com a reutilização de código, existe uma redução considerável da codificação, simplificando a compreensão e a confiabilidade no mesmo.

Existem técnicas, historicamente empregadas, para se atingir a reutilização, definidas em (Sauvé, 2002):

- Fase infantil (500 AC até anos 1960) - Reuso da solução do vizinho numa prova (conhecida como cola).
- Fase pré O.O. (Orientação a Objeto): 1960-1970 - Linhas de código copiadas de um programa e usadas em outro (Copiar e Colar); Reuso de código comum de um programa (Subrotinas); Reuso de funções inteiras genéricas, relacionadas, para servir em várias situações em programas diferentes (Bibliotecas).
- Fase da revolução O-O: 1980 – Reuso através de herança, composição/delegação, Interfaces e Polimorfismo.
- Fase após o "deslumbramento OO": 1990 – Reuso através de Padrões de Projeto, *frameworks* e Componentes.

Em (Gimenes & Travassos, 2002), é constatado que reutilizando partes bem especificadas, desenvolvidas e testadas, pode-se construir software em menor tempo e com maior confiabilidade.

O *framework* é uma das técnicas utilizadas para reutilização de software. Este reuso é permitido pelo fato de que o mesmo tem como objetivo capturar as funcionalidades comuns entre aplicações de um determinado domínio. Mas, para um bom projeto de *framework* é essencial a utilização de Padrões de Projeto.

2.2.1 Padrões de Projeto

Padrões de Projeto são modelos de soluções para problemas que já foram resolvidos anteriormente e os mesmos foram sendo melhorados, de modo a se tornarem eficazes. Estas soluções são conhecidas e desenvolvidas por especialistas, tornando-se padrões, pelo fato de

serem utilizadas várias vezes em vários projetos, sendo adaptado integralmente ou de acordo com a necessidade de cada solução.

Embora muitos confundam os conceitos de Padrões de Projeto e *Framework*, os mesmos se diferenciam. Muitos problemas a serem resolvidos no desenvolvimento de um *framework* já foram solucionados por outros projetistas. Daí surge o uso de padrões de projeto durante o desenvolvimento de *frameworks*. Os padrões de projeto são uma parte natural da documentação de um *framework*, desde que eles motivem decisões durante o projeto.

De acordo com (Gamma et al., 2005), os padrões de projeto trazem benefícios ao projeto, pois, o torna mais fácil de ser reutilizado e melhoram a documentação e manutenção, já que, quem conhece os padrões de projeto já tem uma ideia de como está projetado o sistema.

No levantamento feito por (Gamma et al., 2005), foram catalogados 23 padrões, que são os mais aplicados em sistemas e, consequentemente, os mais testados e confiáveis. No entanto, o estudo do padrão de projeto nesta pesquisa irá se restringir à explanação de um dos principais padrões de projeto, que é geralmente utilizado no desenvolvimento de um *framework*, o *Template Method*.

2.2.1.1 *Template Method*

Segundo Sauvé (Sauvé, 2002), o padrão de projeto *Template Method* define o esqueleto de um algoritmo numa operação, deixando que subclasses completem algumas das etapas, permitindo que subclasses redefinam determinadas etapas de um algoritmo sem alterar a estrutura do mesmo.

A figura a seguir, Figura 4, mostra a estrutura padrão do *Template Method*.

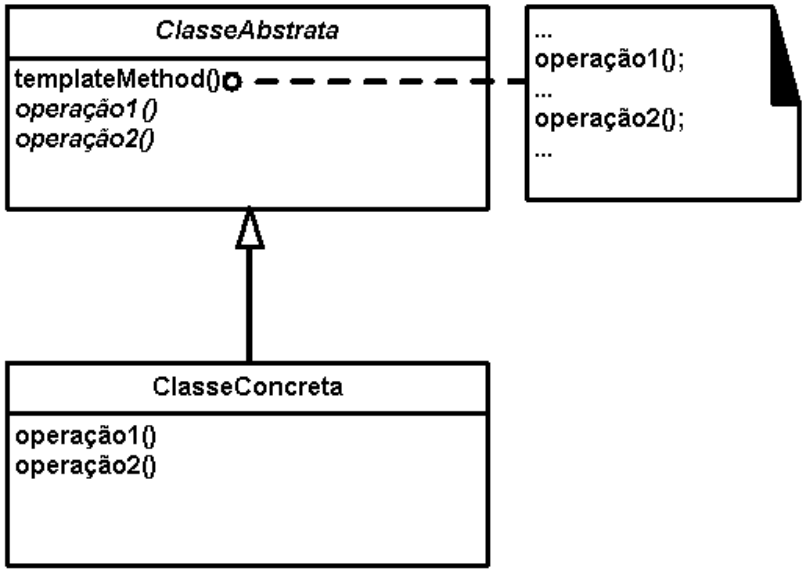

Figura 4: Estrutura do Padrão *Template Method* (Sauvé, 2002).

A classe abstrata define as operações abstratas que as subclasses concretas definem para implementar certas etapas do algoritmo. Ela contém a implementação de um *Template Method*, que é feita definindo um esqueleto de um algoritmo, chamando várias operações abstratas da classe. Neste modelo (Figura 4), o métodos *Operação1()* e *Operação2()* são abstratos e fazem parte do algoritmo do *TemplateMethod()*, sendo que a execução realizada sofrerá variação de acordo com o que foi implementado em cada subclasse. A ClasseConcreta implementa as operações abstratas para desempenhar as etapas do algoritmo que tenham comportamento específico a esta subclasse (Sauvé, 2002).

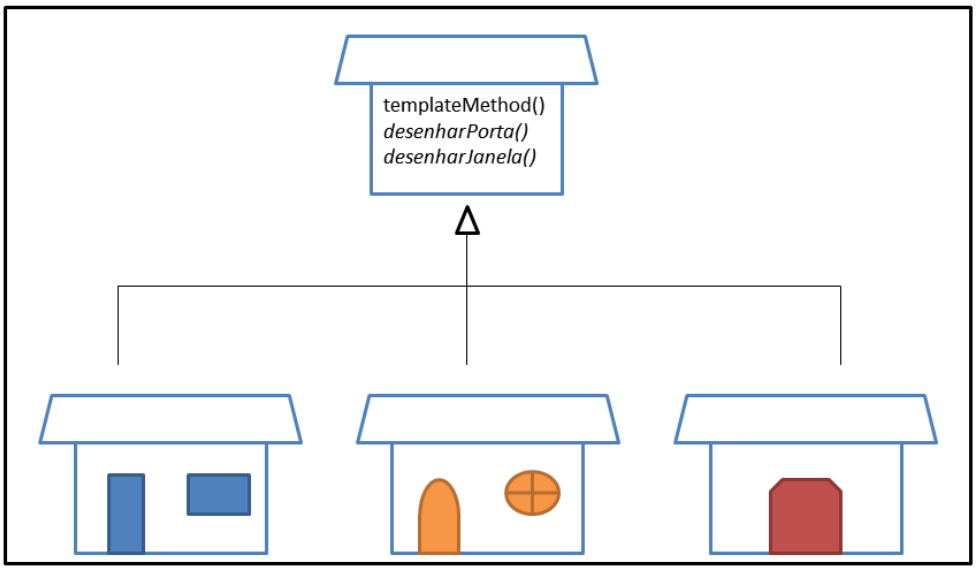

Figura 5: Variações adicionadas ao *Template* da Planta de uma casa.

O *Template Method* define um esqueleto de um algoritmo em uma operação, e adia alguns passos para subclasses. Construtores de casas podem usar este método ao desenvolver uma nova subdivisão. Dentro de uma planta de casa, a fundação, enquadramento, encanamento e fiação serão idênticos para todas as casas. As variações são adicionadas nas fases posteriores da fabricação, a fim de produzir uma ampla variedade de modelos (Figura 5).

2.2.2 *Framework*

Framework [1] é um componente de software reutilizável que fornece larga escala de reuso. Em (Johnson, 1992) *framework* é uma estrutura de classes inter-relacionadas, que corresponde a uma implementação incompleta para um conjunto de aplicações de um domínio. Esta estrutura de classes deve ser adaptada para a geração de aplicações específicas.

Um *framework* captura a funcionalidade comum a várias aplicações, estas devem ter algo razoavelmente grande em comum: pertencerem a um mesmo domínio de problema (Sauvé, 2002). A Figura 6 ilustra os ambientes que são propícios para criação ou não de um *framework.*

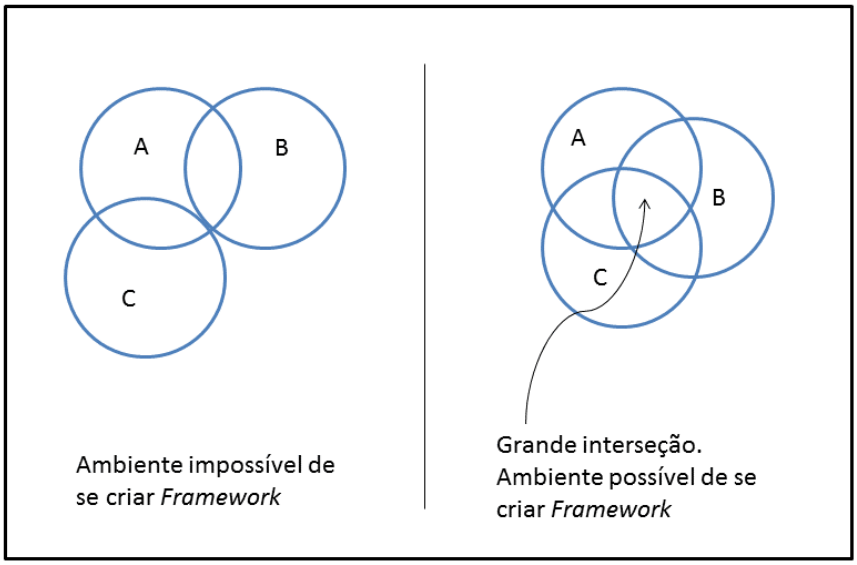

Figura 6: Ambientes possíveis de um domínio. Adaptada de (Sauvé, 2002).

[1] No presente trabalho a expressão *framework* se refere a *framework* orientado a objetos.

Pelo que pode ser visto, na Figura 6, um ambiente no qual há a possibilidade de criar um *framework* é aquele que possui funcionalidades e características em comum. O *framework* é uma aplicação quase completa, mas, com pedaços faltando, desta forma, o objetivo do mesmo é prover as partes generalistas, e a aplicação que for utilizá-lo implementará as partes específicas ao seu negócio.

2.2.2.1 Processo de desenvolvimento de um *framework*

Neste tópico, será descrito o processo de desenvolvimento de um *framework* orientado a objetos, proposto por LANDIN e NIKLASSON em sua tese de doutorado (Landin & Niklasson, 1995).

A Figura 7 ilustra o ciclo do processo de desenvolvimento de um *framework* orientado a objetos e os relacionamentos de suas fases. A seguir, serão descritas as fases defendidas por (Landin & Niklasson, 1995).

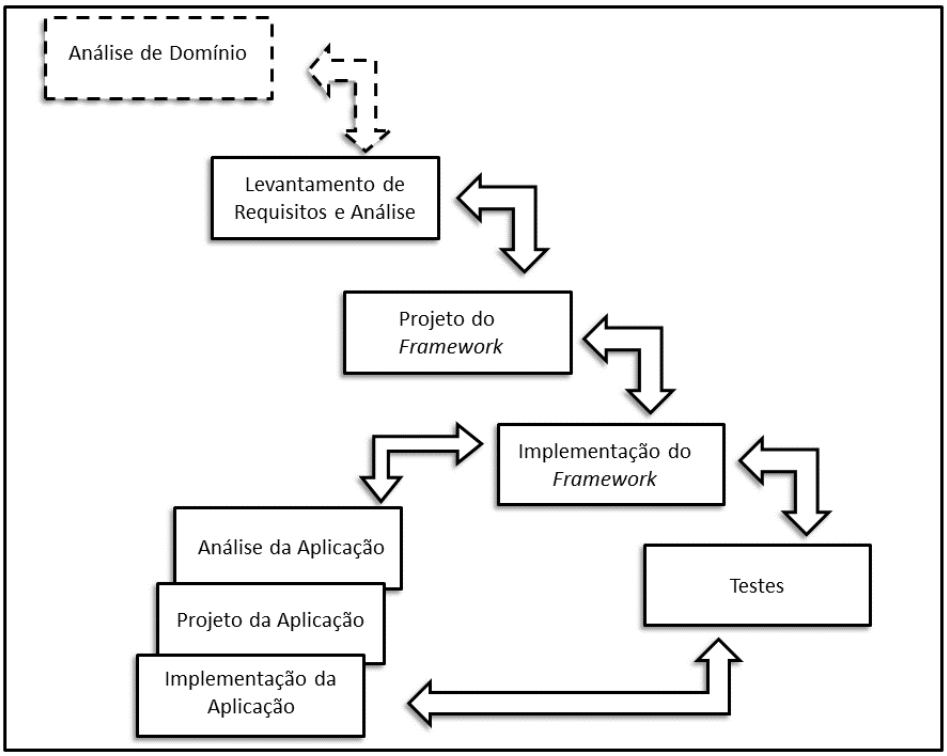

Figura 7: Fases de Desenvolvimento de um *Framework* Orientado a Objetos. Adaptada de (Landin & Niklasson, 1995).

2.2.2.1.1 Análise do Domínio

Para dar inicio ao desenvolvimento de um *framework* é necessário, primeiramente, que seja definido qual o domínio que o mesmo deve abranger. Para isto, é feito um estudo, no qual será definido o domínio da arquitetura a ser criada. Esta é a primeira fase no desenvolvimento de um *framework*, a qual é chamada de Análise de Domínio.

É nesta fase que são identificados os objetos e classes em comum entre as aplicações. Facilitando assim, a criação dos casos de uso, que serão abordados nas próximas fases. Como resultado desta fase, serão produzidos dois documentos: o escopo do domínio e um modelo estático, contendo as classes e objetos levantados. No primeiro, serão descritos os limites de abrangência que o *framework* terá, este escopo será importante na fase de levantamento de requisitos. O segundo documento deve conter as principais classes e objetos de domínio, este modelo será um instrumento de comunicação, para os futuros desenvolvedores de aplicações e seus usuários.

2.2.2.1.2 Levantamento de Requisitos e Análise

O objetivo da segunda fase de desenvolvimento de um *framework* é levantar todos os requisitos e esquematizar o sistema que realizará esses requisitos. A Figura 8 ilustra esta fase com seus produtos e subprocessos (Landin & Niklasson, 1995).

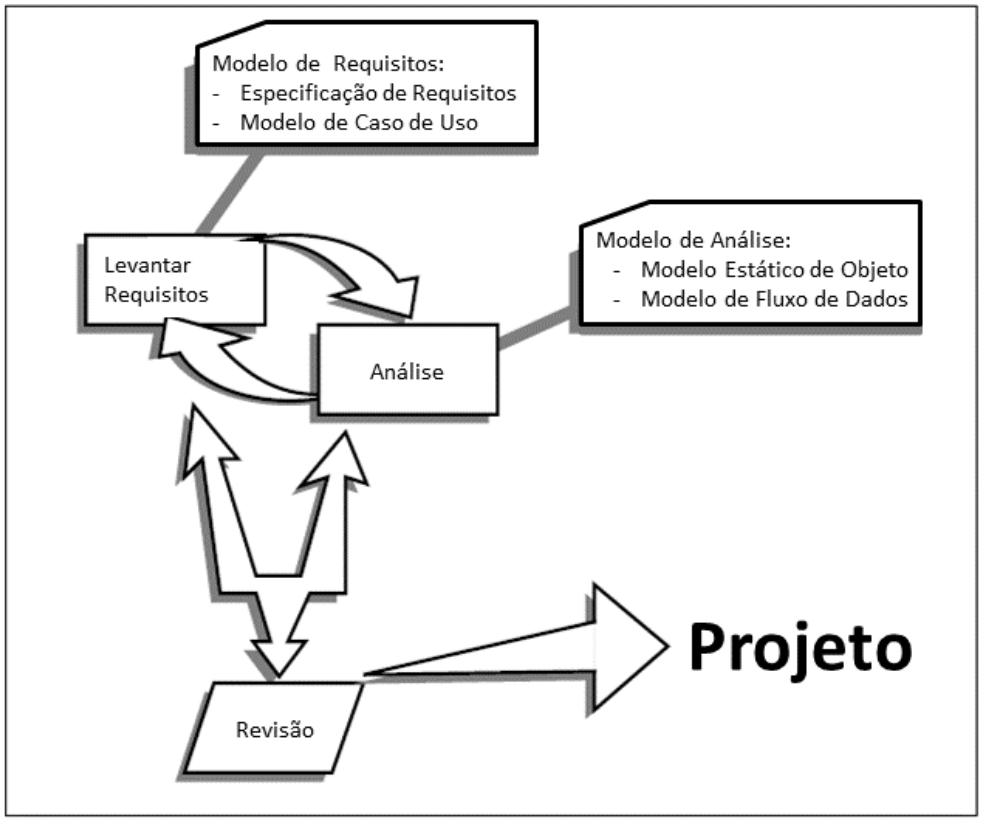

Figura 8: Fase de Levantamento de Requisitos e Análise com seus subprocessos e produtos (Traduzida de (Landin & Niklasson, 1995)).

Os requisitos expressam as características e restrições de um sistema ou de um serviço que o mesmo irá oferecer. Nesta fase serão desenvolvidos dois produtos: Modelo de Requisitos e Modelo de Análise. O primeiro especificará os requisitos exigidos ao sistema (casos de usos e especificação dos requisitos) e o seguindo descreverá os principais conceitos do sistema (modelo conceitual). O modelo de análise precisa ser atualizado, sempre que houver mudanças nos requisitos.

Levantamento de requisitos

Esta atividade tem como objetivo o levantamento de todos os requisitos necessários no desenvolvimento do *framework*. Nesta fase, são construídos dois documentos, que formam o modelo de requisitos: Especificação Detalhada de Requisitos e o Modelo de Caso de Uso. Este modelo forma uma base para a fase de testes.

Os requisitos serão separados em requisitos das aplicações envolvidas e requisitos do *framework.* Ainda serão divididos em requisitos funcionais e não-funcionais. Os requisitos funcionais são aqueles que irão especificar as funcionalidades que o sistema fornecerá, e os

requisitos não funcionais são os que estão relacionados ao uso da aplicação, em termos de disponibilidade, desempenho, confiabilidade, segurança, manutenibilidade, usabilidade e tecnologias envolvidas. A Figura 9 mostra essa separação.

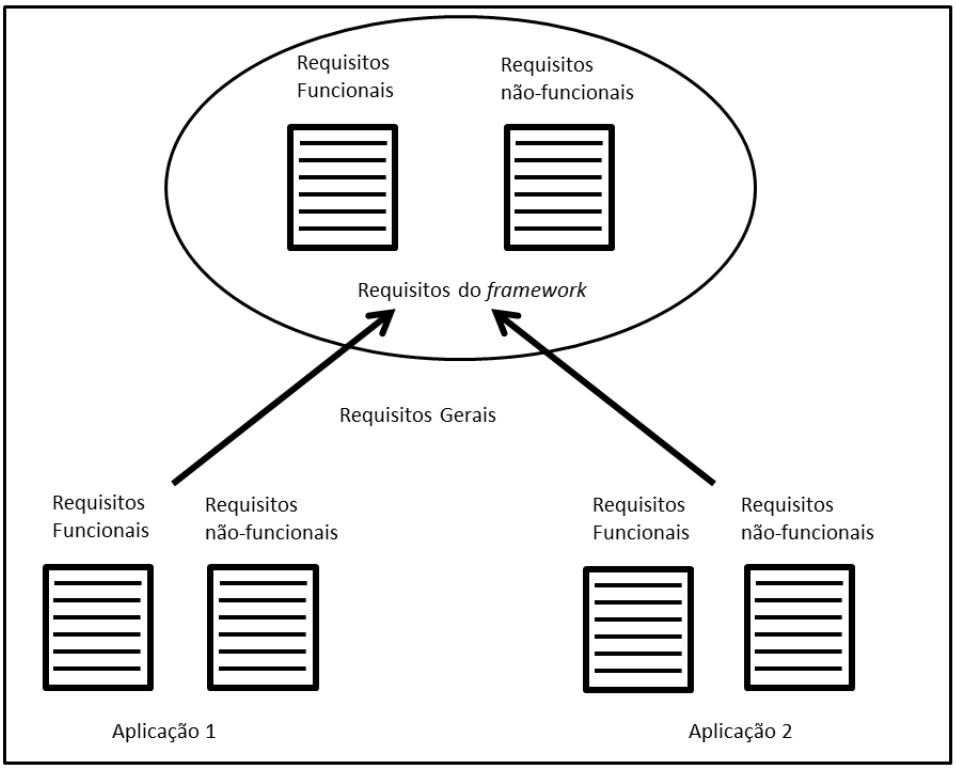

Figura 9: Divisão dos Requisitos em funcionais e não-funcionais das Aplicações e do *Framework*. Adaptada de (Landin & Niklasson, 1995).

O modelo de caso de uso é outro documento importante nessa fase. O caso de uso definirá de que forma o sistema será usado e o que será executado para cada entrada. É interessante que sejam separados os casos de uso do *framework* dos casos de uso da aplicação. Os atores podem representar os usuários do sistema ou um outro sistema.

Análise

O objetivo desta atividade é elaborar um modelo de um sistema que execute os requisitos. Para uma boa modelagem, em (Landin & Niklasson, 1995) são descritos alguns passos a serem executados:

- Esquematizar a situação e o problema;
- Examinar soluções existentes;
- Identificar abstrações chave;
- Identificar abstrações de alto nível;

- Identificar quais partes do problema o *framework* abrangerá;
- Questionar a entrada originada dos clientes e refinar a abordagem.

Como resultado da atividade de análise está o modelo de análise, no qual faz parte o modelo conceitual. Nele são modelados os objetos, a relação entre eles e os conceitos importantes para a aplicação a ser construída.

De acordo com (Landin & Niklasson, 1995), em todos esses modelos estudados, é necessário que os mesmos sejam feitos com notações gráficas, facilitando assim o seu entendimento.

2.2.2.1.3 Projeto

Após definir o domínio de abrangência do *framework* e fazer coleta e análise dos requisitos, é o momento de elaborar o projeto do mesmo. Esta fase abrange a fase do projeto arquitetural, no qual os objetos e suas colaborações são definidos e a fase do projeto detalhado, de forma que as classes e seus métodos sejam descritos com maior detalhe (Landin & Niklasson, 1995).

As fases anteriores não têm um detalhamento propício para iniciar a implementação do *framework*, por isso a necessidade de existir uma fase de projeto. Nesta fase, os modelos de análise são estendidos, sendo levada em consideração a tecnologia a ser utilizada na implementação do sistema, servindo também para validar a análise já realizada.

A ordem das atividades da fase de projeto é exibida na Figura 10 (Landin & Niklasson, 1995).

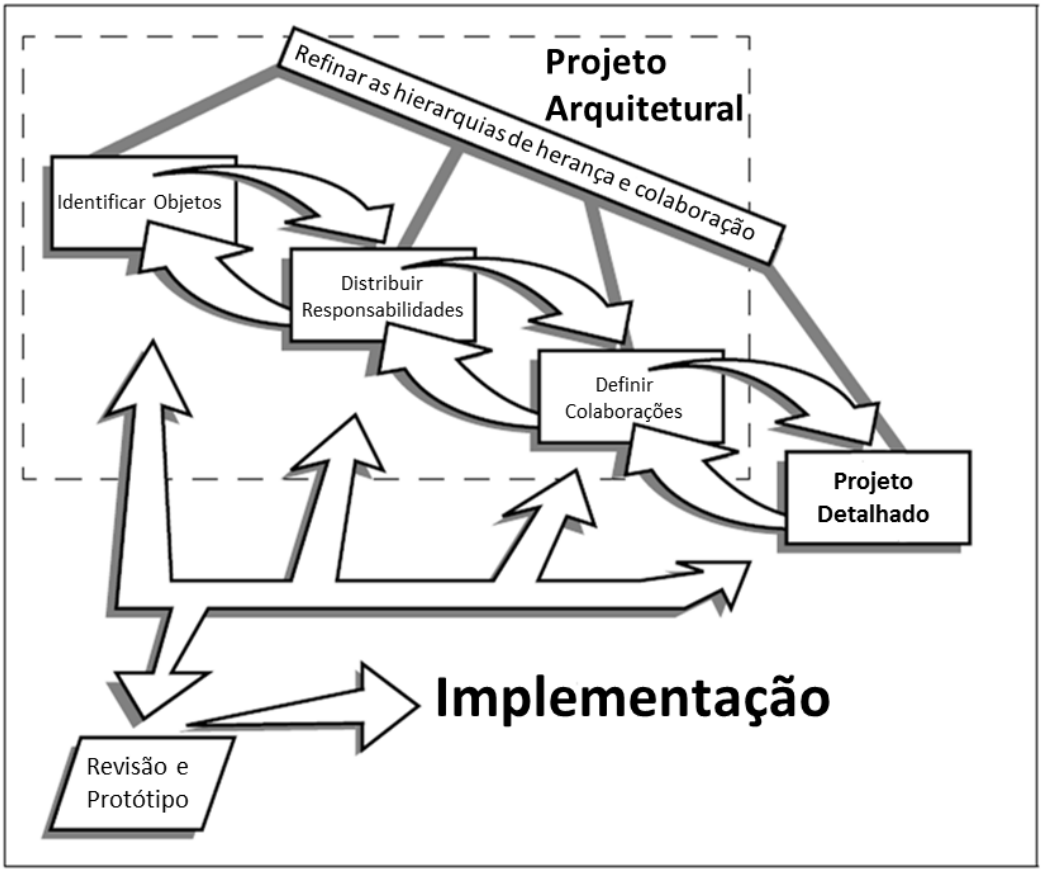

Figura 10: Atividades da fase de projeto (Traduzida de (Landin & Niklasson, 1995)).

O projeto é continuamente revisado e as soluções podem ser validadas através de protótipos.

Projeto Arquitetural

O objetivo do projeto arquitetural é identificar os objetos necessários para implementar o sistema e a forma como os objetos colaboram. Se necessário, o sistema é dividido em subsistemas durante esta fase (Landin & Niklasson, 1995).

As subatividades do projeto arquitetural são descritas na Figura 11 (Karlssom, 1995 apud Landin & Niklasson, 1995).

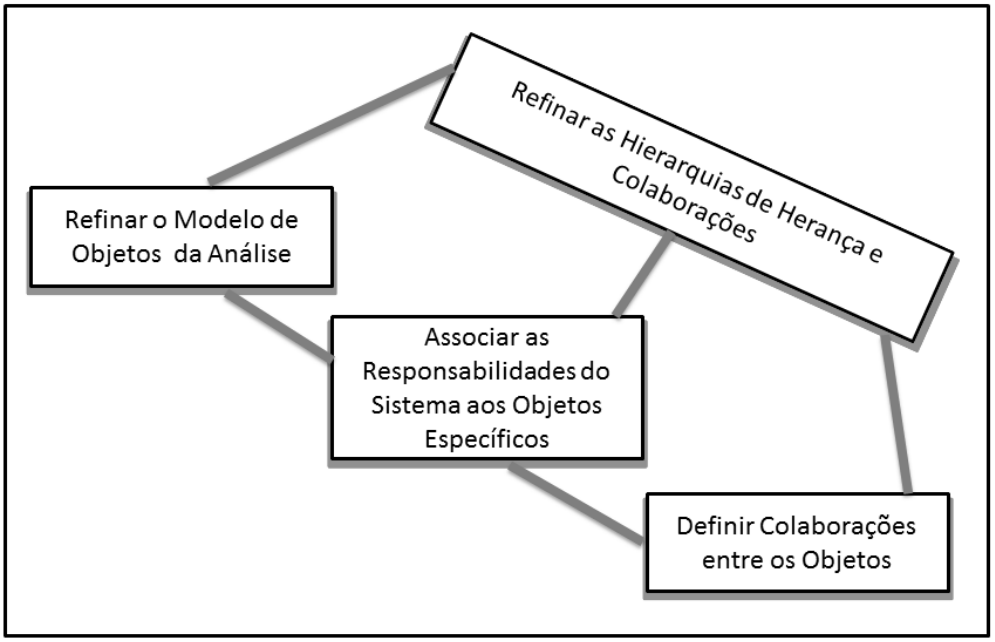

Figura 11: Subatividades do Projeto Arquitetural. Adaptada de (Landin & Niklasson, 1995).

Em (Landin & Niklasson, 1995), as entradas nessa atividade são os requisitos e os modelos da análise. Os seus resultados são os modelos estáticos e dinâmicos de objetos (diagramas de interação, gráficos de transição de estado e modelos de fluxo de dados).

Refinar o Modelo de Objetos da Análise: nesta subatividade, novos projetos podem ser criados para adaptar o sistema ao ambiente de implementação, e outros objetos, já presentes no modelo de análise, podem ser apagados, unidos ou divididos. Em (Landin & Niklasson, 1995), é dito que as classes devem ser apropriadamente pequenas. Classes com mais de 25 métodos são candidatas a serem reestruturadas.

Associar Responsabilidades do Sistema aos Objetos Específicos: as responsabilidades do sistema, nessa subatividade, são distribuídas entre os objetos nas fases anteriores. Durante a especificação das operações, um objeto é responsável por executar um caminho comum, expressando responsabilidades similares, no intuito de encontrar abstrações (Mendonça, 2002).

Definir colaborações entre os objetos: existe a colaboração de objetos quando um invoca em um ou mais métodos de outro para executar suas responsabilidades. Para cada objeto e sua responsabilidade, deve saber se o próprio objeto pode realizar suas atividades, ou se é necessário que o mesmo receba colaboração de outro. Existem ferramentas gráficas, os Diagramas de Interação, que mostram como os objetos colaboram entre si num determinado

sistema. Em (OMG, 2003), são definidos dois diagramas de interação: digrama de sequência e diagrama de colaboração.

Um diagrama de sequência mostra, explicitamente, a sequência das comunicações e é melhor para especificação de sistemas em tempo-real e cenários complexos. Um diagrama de colaboração, por sua vez, mostra uma interação organizada baseada em suas regras e seus relacionamentos, não mostrando o tempo como dimensão separadora (OMG, 2003).

Refinar as hierarquias de herança e colaborações: esta subatividade tem como objetivo promover as abstrações existentes durante o processo, pois, nem todas são identificadas nas fases anteriores. As hierarquias de classes devem ser, razoavelmente, profundas e estreitas. Heranças superficiais e largas indicam que abstrações ainda podem ser encontradas (Landin & Niklasson, 1995). Para dar início a esse refinamento, um caminho que pode ser seguido é observar as subclasses que implementam funcionalidades similares e tentar migrá-las para uma superclasse. No entanto, essa abordagem resulta em herança profunda e de difícil compreensão. Sempre que possível, a herança deve ser substituída por composição. Os projetistas devem se atentar às classes que possuem nomes diferentes com funcionalidades iguais. Neste caso, deve ser investigada a possibilidade de parametrização. Múltiplas Heranças e Múltiplas Heranças Duplas causam dificuldade no seu entendimento, então devem ser evitadas.

Projeto Detalhado

Durante esta fase do projeto, todas as classes com atributos e métodos são identificados e descritos utilizando a linguagem de implementação a ser utilizada (Karlsson, 1995 apud Landin & Niklasson, 1995). As entradas são os objetos e colaborações identificados no projeto arquitetural.

Nesta atividade, é válido observar alguns aspectos. Métodos com mais de cinco parâmetros, são candidatos a serem refinados e divididos em outros métodos, tendo como exceção, para esta regra, os métodos construtores. Um método deve executar apenas uma tarefa, pois, caso contrário, ele pode estar executando atividades que fogem de sua responsabilidade, gerando um acoplamento forte[2]. Classes com mais de vinte e cinco métodos são candidatas a serem reestruturadas, pois, devem estar reunindo várias abstrações. Novas abstrações podem ser encontradas no projeto detalhado, por isso, estas abstrações devem ser

[2] *Acoplamento forte*: nível de dependência ou relacionamento entre métodos, classes ou subsistemas.

documentadas nos modelos apropriados: abstrações conceituais, no modelo de análise, e abstrações de baixo nível, no modelo de projeto (Mendonça, 2002).

2.2.2.1.4 Implementação

Após a fase de projeto, é realizada a implementação. Ela corresponde à codificação e "materialização" do que foi documentado. Seu principal objetivo é implementar os objetos, os relacionamentos e as colaborações entre eles, identificadas na fase de projeto (Mendonça, 2002). Não existe uma fronteira bem definida entre as fases de projeto detalhado, implementação e testes, pois, durante a implementação, os componentes também já são testados, e qualquer inconsistência encontrada forçará a volta ao projeto para fazer a modificação.

A entrada desta fase é a descrição detalhada das classes, suas interfaces e definições externas, especificadas na fase de projeto, tomando como base a linguagem de programação a ser utilizada. A saída é um conjunto de classes implementadas e prontas para serem testadas (Karlsson, 1995 apud Landin & Niklasson, 1995).

A estratégia de implementação mais indicada é a top-down[3], na qual se implementam as classes de alto nível primeiro e, posteriormente, as de baixo nível. Isso possibilita a prototipação, desde que as funcionalidades principais tenham sido definidas num estágio anterior (Karlsson, 1995 apud Landin & Niklasson, 1995).

Para uma melhor manutenção de código e legibilidade do mesmo, é interessante que seja utilizado padrões na implementação. Envolvendo padrão de nomenclatura dos métodos, atributos, classes e áreas de comentários, explicando, de forma resumida, a funcionalidade das classes, métodos ou qualquer outro lugar que seja necessário documentar.

Para um *framework* desenvolvido em uma linguagem Orientada a Objetos, os métodos de uma subclasse não devem ser chamados diretamente, pelo fato de causar uma dependência na estrutura da herança. Então, é importante que os métodos sejam acessados pela interface da superclasse abstrata e, em tempo de execução, o sistema faz a ligação dinâmica, chamando o

[3] *Top-down*: abordagem onde o foco deve ser do nível mais alto até o mais baixo.

método da subclasse. Isso pode ser visto de forma prática utilizando o padrão de projeto *Template Method*, o qual é descrito no tópico 2.2.1.1.

Herança privada não deve ser utilizada, pois não é um conceito orientado a objetos e dificulta a utilização do *framework*. Classes abstratas não devem ser instanciadas. Os métodos das classes dos *frameworks* que podem ser redefinidos devem ser declarados como virtual. Isso possibilita que o comportamento seja estendido pela aplicação que implementa as subclasses que utilizam o *framework*. (Mendonça, 2002).

Métodos pequenos são mais fáceis de entender e modificar. Métodos grandes são difíceis de dar manutenção e podem acoplar responsabilidades de outros métodos. Um método com mais de 20 linhas pode ser candidato a ser reestruturado em vários outros (Landin & Niklasson, 1995).

Para não violar um dos conceitos da programação orientada a objetos, o encapsulamento, especificar os atributos da classe com visibilidade private e definir dois métodos public, para cada atributo, que retornem e alterem o seu valor. Com relação aos construtores e destrutores das classes, a implementação deve se concentrar apenas em definir um construtor default (sem parâmetros) e um ou mais construtores com parâmetros que serão atribuídos aos atributos da classe. Em classes abstratas, o construtor deve ser construído com visibilidade protected, pois, as mesmas não devem ser instanciadas.

2.2.2.1.5 Testes

A fase de testes é a última, no processo de desenvolvimento de um *framework*. Esta fase contempla os conceitos de Verificação e Validação (Sommerville, 2006). A Verificação se preocupa com a construção correta do software, analisando se o *framework* atende a sua especificação, ou seja, se o mesmo está sendo construído de acordo com o que foi documentado nas fases anteriores do processo desenvolvimento. A Validação se destina a mostrar que o *framework* realiza o que o usuário necessita.

Duas questões eliminam as dúvidas com relação a esses dois conceitos (Sommerville, 2006).

- Verificação: Estamos construindo certo o produto?
- Validação: Estamos construindo o produto certo?

A fase de testes é categorizada em Teste de Unidade, Teste de Integração e Teste de Regressão. Nesta fase também podem ser feitas a inspeção e revisão do código e testes estatísticos.

Testes de Unidade: é feito um teste em uma unidade do *framework* de cada vez. Esta unidade pode ser uma classe, uma operação ou um módulo composto de várias classes. A intenção é verificar se as responsabilidades impostas a essas unidades estão sendo realizadas.

Teste de Integração: responsável por testar as funcionalidades das unidades em grupo. É alimentado pelos módulos previamente testados individualmente, pelo teste de unidade, agrupando-os assim em componentes, no intuito de atingir os objetivos previamente definidos. Segundo (Landin & Niklasson, 1995), os casos de uso são uma boa ferramenta para executar os testes de integração. O teste de integração final é feito combinando todas as unidades componentes do sistema e testando todas elas de forma global. Este teste também é chamado de Teste do Sistema.

Teste de Regressão: é uma técnica aplicável a uma nova versão do *framework* ou à necessidade de se executar um novo ciclo de teste, para garantir que não surgiram novos defeitos em componentes já testados, medindo assim a qualidade do produto produzido.

Não existe uma diferença substancial entre testar unidades de um *framework* e testar unidades de um sistema comum. Contudo, existem algumas diferenças dependendo do uso das técnicas orientadas a objetos e o fato de que os *frameworks* são desenvolvidos para serem reutilizados (Mendonça, 2002).

Existem duas principais questões que os testes, aplicados a um *framework*, devem responder. Primeira: O *framework* realmente abrange o domínio pretendido? Esta pergunta pode ser respondida à medida que o mesmo é reutilizado. Se um *framework* é inadequado para o desenvolvimento de uma aplicação, dentro do seu domínio, ele não atende aos seus requisitos e precisa ser reprojetado ou o escopo do seu domínio precisa ser modificado. Segunda: O *framework* realiza as partes dos requisitos da aplicação que está dentro da área de responsabilidade do *framework*? Se novas classes concretas, que estendem o *framework*, forem criadas para aplicações específicas, essas precisam de testes de unidade. Mas, o *framework* não precisa ser testado novamente, como um componente isolado. No entanto, precisa ser feito um teste de integração, a fim de verificar como as novas classes interagem com o *framework*. A herança reduz a quantidade de código, mas não a quantidade de testes.

Métodos herdados devem ser testados como métodos implementados na própria classe (Mendonça, 2002).

O desenvolvedor que irá produzir um novo sistema e deseja reutilizar algum componente já desenvolvido, necessita ter um certificado de confiança, para que o mesmo tenha a segurança de que o sistema não apresentará falhas. Para que haja maior confiabilidade, quatro medidas foram definidas. São elas (Sommerville, 2006):

- Probabilidade de Falha por Demanda: avalia a probabilidade do sistema em responder a situações inesperadas.
- Taxa de Ocorrência a Falhas: mede, em unidades de tempo, quanto tempo o sistema se comporta sem ocorrer situações inesperadas.
- Tempo significativo à falhas: mede o tempo significativo da ocorrência entre duas falhas no sistema.
- Disponibilidade: A probabilidade de que o sistema está disponível para uso em um dado tempo.

O Teste estatístico é um método de teste que tem como objetivo avaliar a confiabilidade do sistema (Sommerville, 2006). Existem quatro passos para se executar um teste estatístico (Landin & Niklasson, 1995):

- Determinar o padrão de como as unidades serão utilizadas;
- Reunir e selecionar dados para teste, identificando testes de caso de acordo com este padrão;
- Executar o teste de caso de acordo com o padrão usado. Registra o tempo de execução até uma falha ocorrer;
- Calcular a confiabilidade do software de acordo com os resultados dos testes.

A proposta de um *framework* é que o mesmo seja reutilizado em diversas novas aplicações. Diferentes aplicações podem ter padrões de uso muito diferentes, o que pode limitar o uso de testes estatísticos. No entanto, existem trabalhos que argumentam o uso de teste como método para certificação de confiabilidade de componentes de software. O uso de testes também pode ser um método adequado para certificar *frameworks* (Landin & Niklasson, 1995).

3. IMPLEMENTAÇÃO DO *FRAMEWORK* PARA GERAÇÃO AUTOMÁTICA DE CÓDIGO

Este capítulo tem como objetivo principal mostrar todos os conhecimentos adquiridos, durante toda a pesquisa, de forma prática. A proposta é de desenvolver um *framework* de suporte a ferramentas de geração de código automática para ambientes de apoio à decisão. Este *framework* implementará a funcionalidade de geração de procedimentos SQL, que terão como função a carga de dados da área de *Staging* para o DW.

3.1 Análise de Domínio

O escopo de domínio do *framework GerarProcedimentoSQL* é abranger aplicações que necessitam gerar procedimentos SQL para cargas de dados, sendo estas da área de *Staging* para dimensões de um DW, de tal forma que utilizem da reutilização de uma estrutura já existente, ou seja, estendendo o *framework*. Este *framework* poderá apenas ser estendido por aplicações desenvolvidas na linguagem Java.

O modelo de classes desta fase está representado abaixo, na Figura 12.

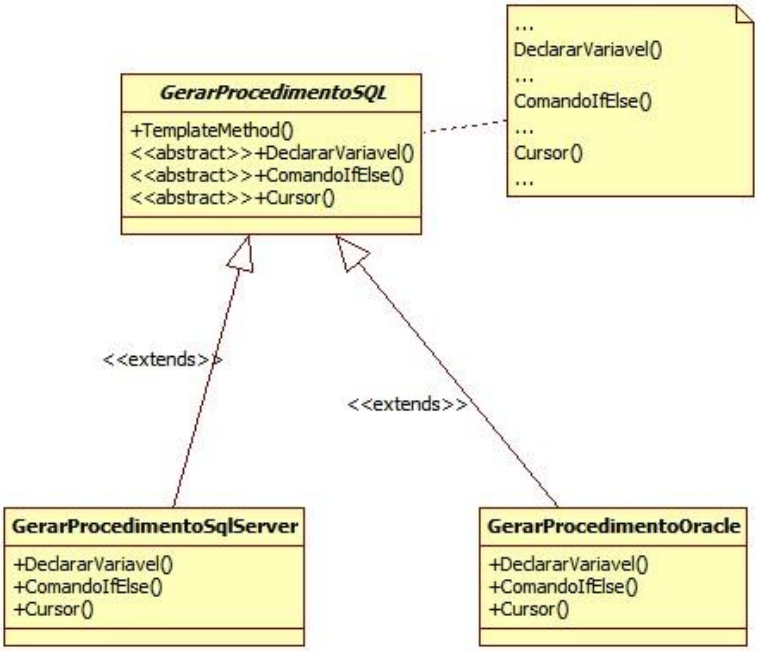

Figura 12: Modelo de Classes do *framework.*

3.2 Levantamento de Requisitos e Análise

Prosseguindo com o processo de desenvolvimento do *Framework*, passamos agora para fase de Levantamento de Requisitos e Análise. Nesta fase serão descritos o modelo de Requisito e o Modelo de Análise.

Modelo de Requisitos

Para o Modelo de Requisitos serão criados dois documentos, Modelo de Caso de Uso e Especificação dos Requisitos.

A Figura 13 mostra o modelo de Caso de Uso do *Framework GerarProcedimentoSQL*. Os Casos de Uso de cor branca são abstratos.

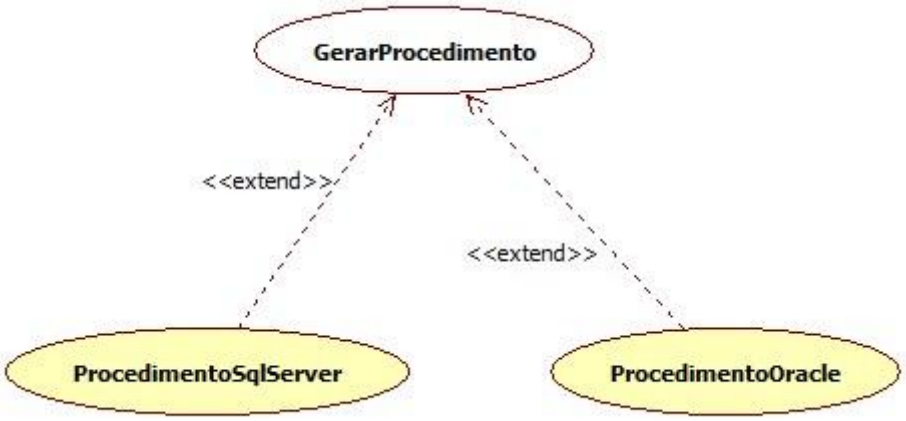

Figura 13: Modelo de Caso de Uso.

O *Framework* deve atender às seguintes especificações de requisito:

- Requisitos Funcionais
  - Suporte à criação de procedimentos para dimensões do tipo 1, 2 e 3.
  - Encapsular lógica independente do banco.
- Não funcionais
  - Desenvolvido em Java.
  - Deve ser estendido por aplicações que sejam desenvolvidas em Java;
  - Utiliza o SQL Server para armazenar dados.

Modelo de Análise

O modelo de análise sofreu algumas alterações, em relação ao que foi criado na fase de análise de domínio. Essas alterações foram frutos do detalhamento do modelo, ou seja, os objetos e suas relações foram modelados de forma minuciosa.

O modelo de classes desta fase está representado abaixo, na Figura 14.

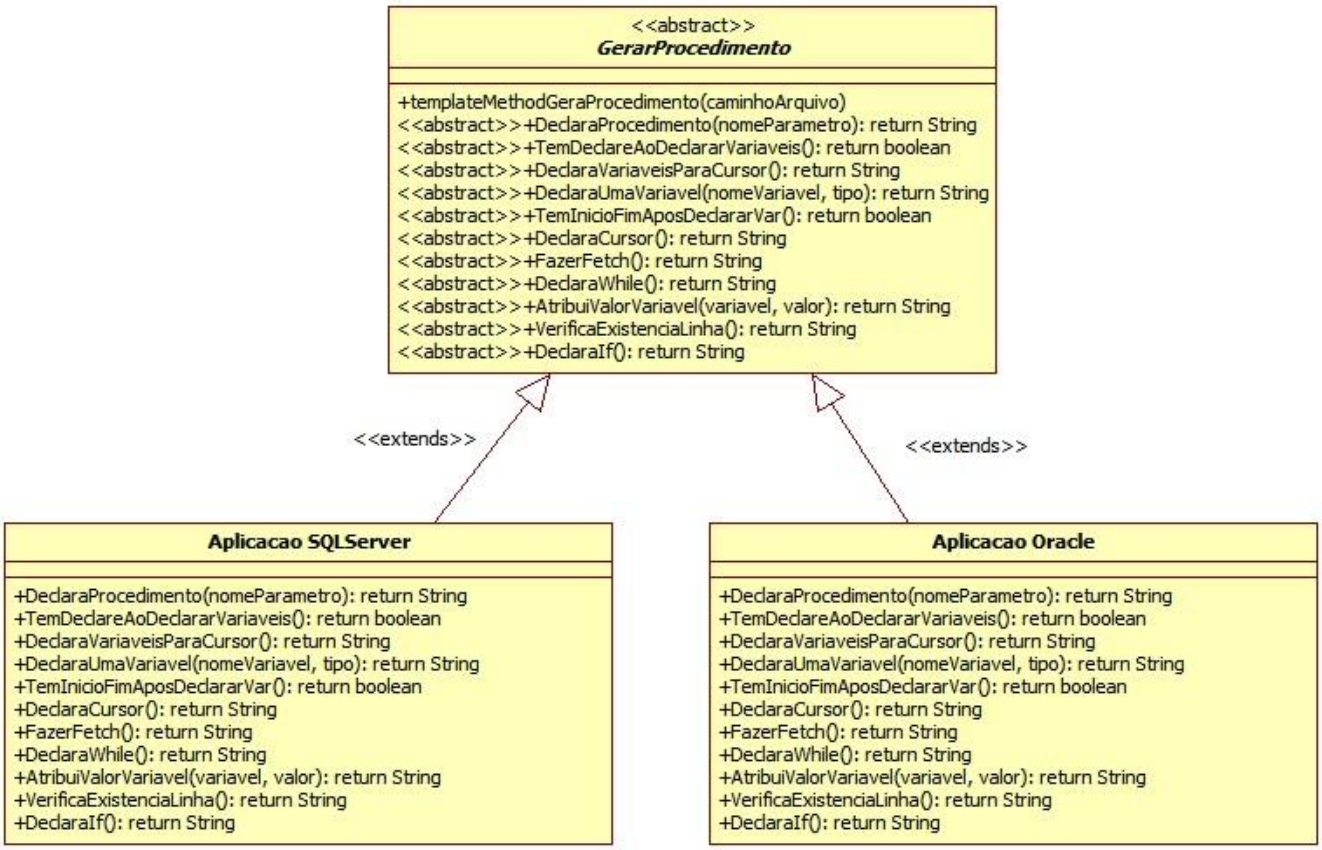

Figura 14: *Modelo de Classes, com melhor modelação do framework.*

3.3 Projeto

O projeto do *Framework GerarProcedimentoSQL* é formado pelo projeto arquitetural, no qual será demonstrado a forma que se dispõe a arquitetura do *framework*, e pelo projeto de baixo nível, o qual é responsável por detalhar o modelo conceitual, mostrando as classes que serão implementadas e a interação entre os objetos.

3.3.1 Projeto Arquitetural

O projeto Arquitetural foi desenvolvido com o intuito de documentar as decisões de alto nível. Sendo identificadas as camadas, módulos e suas interdependências. Visando

atender aos requisitos não-funcionais, detectados na fase de análise. Além disso, mostra a tecnologia utilizada para comunicação entre camadas.

A arquitetura do *Framework GerarProcedimentoSQL* está representada em três camadas: Interface, Aplicação e Dados. O Projeto Arquitetural do *Framework GerarProcedimentoSQL* pode ser visualizado abaixo, Figura 15.

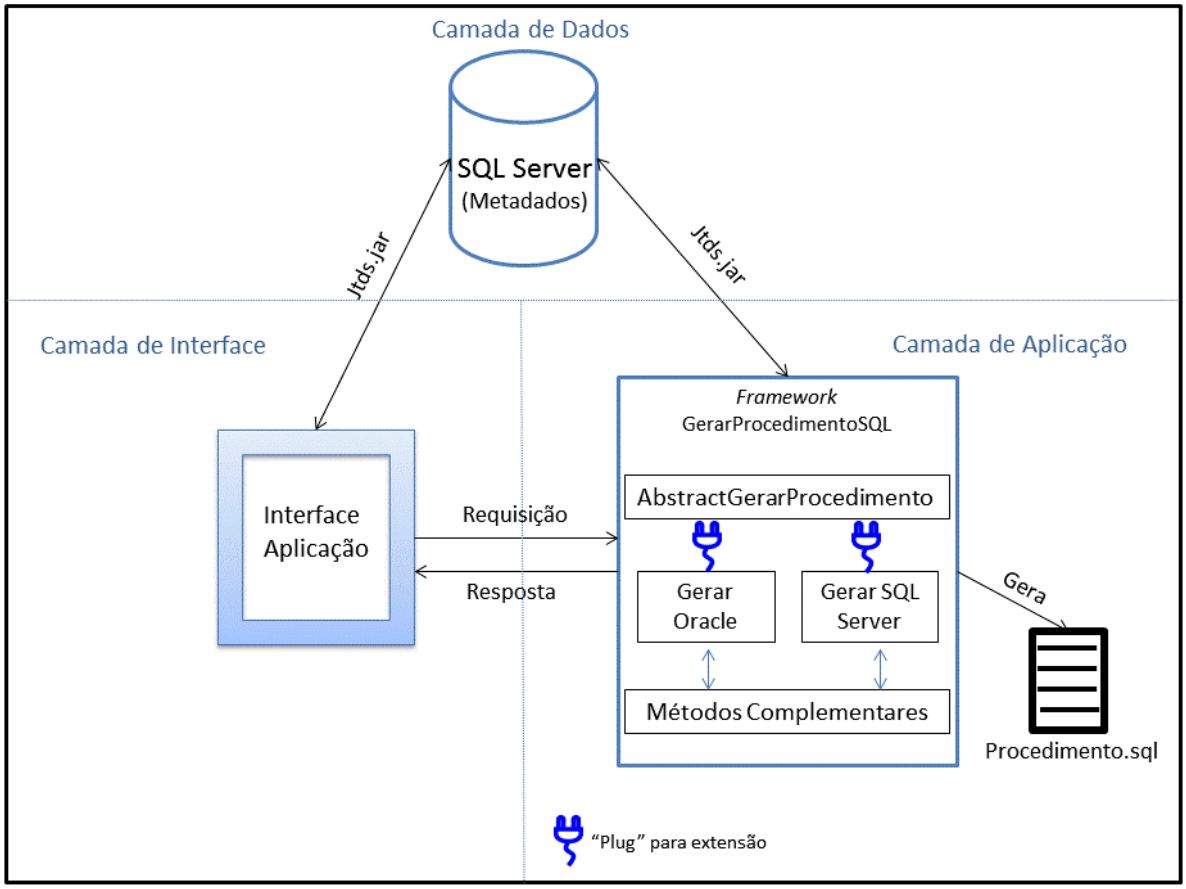

Figura 15: *Projeto Arquitetural do Framework GerarProcedimentoSQL.*

A camada que interage com o usuário final é a camada de Interface. É nela que o usuário poderá escolher para qual banco de dados, diante dos já implementados, ele vai gerar o procedimento. Além disso, o usuário escolherá qual o destino em que o arquivo, com a extensão .sql, será armazenado. É este arquivo que conterá o procedimento em SQL, o qual será utilizado para realizar a carga de uma tabela auxiliar, da Área de *Staging* para uma dimensão no DW. Nesta etapa, os metadados necessários para criação do procedimento já devem estar armazenados no Banco de Dados, pois neste momento serão informadas quais tabelas auxiliar e dimensão farão parte do procedimento.

A camada de aplicação contém toda regra de negócio necessária para funcionamento do *framework*. Este foi dividido em duas partes, uma classe abstrata, na qual é feita toda a lógica e sequencia de métodos em comum a serem chamados, para formação do procedimento, e uma classe concreta, na qual tem o objetivo de realizar as instanciações para captura dos metadados utilizado nas classes que fazem extensão da classe abstrata.

A camada de dados é responsável por fornecer os metadados necessários para montagem do procedimento.

Além dos módulos existentes, o projeto arquitetural documenta os pontos de extensão existentes no *Framework GerarProcedimentoSQL*. Desta forma facilita o desenvolvimento de novas aplicações que irão estender este *framework*. Estas devem se preocupar em estender a classe abstrata, *AbstractGerarProcedimento*, e também utilizar alguns métodos fornecidos pela classe concreta, *MetodosComplementares*.

3.3.2 Projeto de Baixo Nível

Este projeto, de baixo nível, define os modelos que auxiliarão na implementação do *framework*. Será ilustrado o diagrama de classes detalhado e diagramas de sequencia, os quais mostrarão a interação dos objetos de uma forma mais clara.

3.3.2.1 Diagrama de Classes

O diagrama de classes modela as classes, com métodos e objetos a serem utilizados.

A Figura 16 ilustra o diagrama de Classes do *Framework* em questão.

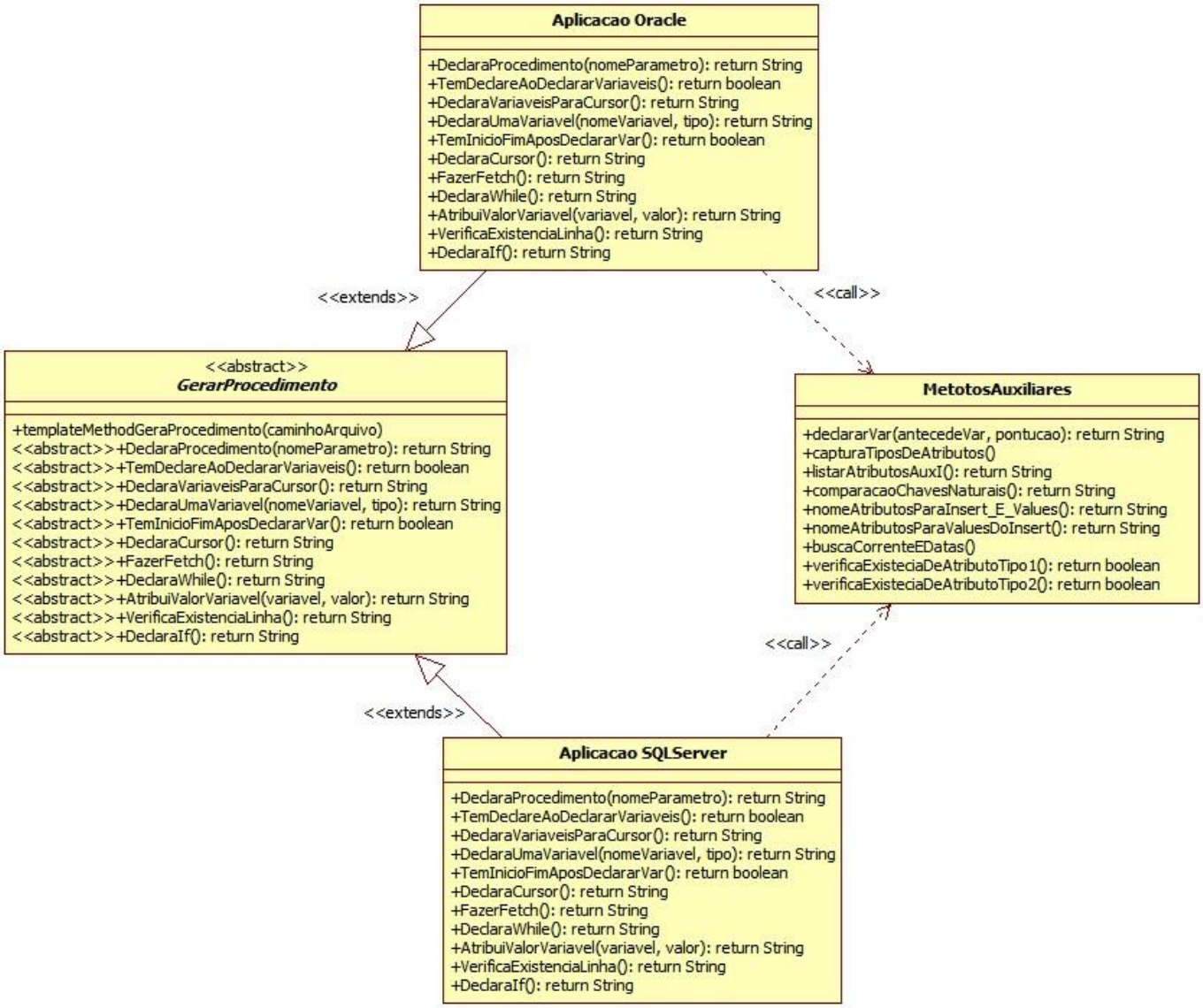

Figura 16: *Diagrama de Classes do Framework GerarProcedimentoSQL.*

Este modelo sofreu alterações, com relação ao definido no Modelo de Análise (ver Figura 14). Inicialmente, foi definido que existiria apenas a classe abstrata, que conteria o método *templateMethodGeraProcedimento*, e as classes concretas das aplicações, que estenderiam a classe abstrata. No entanto, para criação do arquivo.sql, que conterá o procedimento, é preciso carregar os metadados que estão armazenados no Banco de Dados. Como a classe abstrata não pode instanciar objetos, elas estavam sendo feitas separadamente, ou seja, cada classe concreta faria as implementações com as instancias necessárias. Então, as diferentes classes estavam implementando métodos que continham a mesma funcionalidade, e

a classe abstrata se responsabilizava praticamente apenas em ordenar a chamada destes métodos.

Desta forma, não se via muita viabilidade na criação do *framework*, já que existiriam funcionalidades idênticas, sendo implementas mais de uma vez. A duplicação de código é desnecessária e fere o objetivo inicial, o qual diz que as classes concretas que estenderão o *framework* implementem apenas as características que as diferenciam.

Para tentar resolver este problema, sem que houvesse a necessidade de muita mudança, foi criada uma nova classe concreta, *MetodosComplementares*, para auxiliar a classe abstrata, implementando assim os métodos que contêm as funcionalidades idênticas, podendo ser chamada pelas classes que estenderão a classe *AbstractGerarProcedimento*.

3.3.2.2 Diagramas de Interação

Foi desenvolvido um diagrama de sequencia para ilustrar os passos realizados, desde a solicitação de geração do procedimento, feita pelo usuário, até a recuperação dos metadados necessários para montagem e a criação do arquivo.

A seguir, na Figura 17, pode ser visto o diagrama de sequencia que explicita a interação e a colaboração entre os objetos.

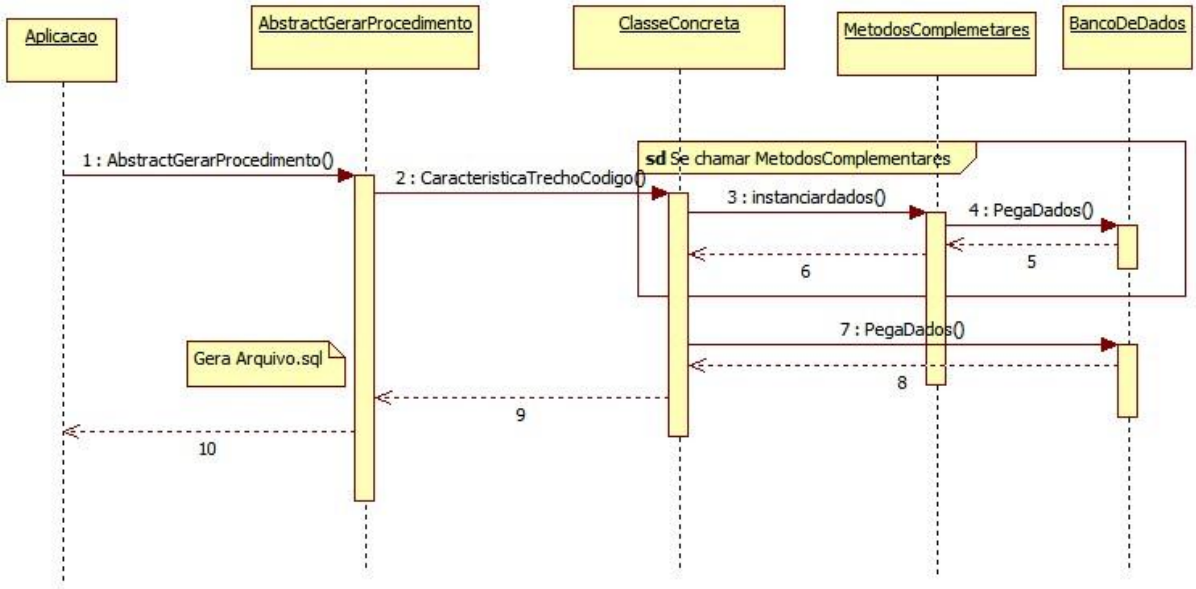

Figura 17: Diagrama de sequencia da Geração de Procedimento SQL.

3.4 Implementação

Na fase implementação do *framework* será materializado, em código, tudo o que foi modelado nas fases anteriores, principalmente nos modelos do projeto, os quais estão descritos de forma mais aprofundada.

As classes implementadas são as seguintes:

- *AbstractGerarProcedimento*: classe abstrata que deve ser estendida para a reutilização das funcionalidades desenvolvidas;
- *MetodosComplementares*: classe concreta que contem instanciações para obtenção dos metadados necessários à geração do procedimento em SQL.

Para reutilização do *Framework GerarProcedimentoSQL*, foram implementadas duas classes, uma contendo as particularidades do dialeto da linguagem SQL da Oracle (PL/SQL) e outra contendo as particularidades do SQLServer (Transact-SQL, ou apenas T-SQL).

Estas classes devem estender a classe abstrata *AbstractGerarProcedimento*, ela conterá o método *templateMethodGeraProcedimento*, o qual tem como parâmetro o caminho em que o arquivo.sql deve ser salvo. Este e os outros parâmetros (nome do procedimento e dialeto da linguagem SQL) serão informados em uma ferramenta já existente.

3.5 Testes

Após da realização da implementação, deve-se garantir que o *framework* está se comportando como realmente deve. Para isso, foram realizados testes a partir das classes concretas, implementando as funcionalidades específicas, à medida que o *framework* era construído, com a finalidade de constatar se o comportamento do mesmo correspondia ao resultado esperado.

4. ESTUDO DE CASO

Nesta seção, são abordadas as atividades necessárias para a execução do estudo de caso realizado. Na Seção 4.1, é apresentado o objetivo do estudo realizado, em seguida, na seção 4.2, é apresentada a definição do planejamento com a seleção dos bancos de dados utilizados e os casos de uso criados. Por fim, na Seção 4.3, é apresentado o processo de operação que consiste na execução e validação dos dados processados.

4.1 Definição de Objetivo

A realização do estudo de caso tem como objetivo principal a validação dos resultados emitidos pelo *framework* produzido neste trabalho. Contudo, para atingir este objetivo, é necessária a realização dos seguintes objetivos específicos:

- Escolher pelo menos dois dialetos da linguagem SQL;
- Implementar classes que atendam às especificidades de cada dialeto, utilizando o *framework*;
- Fazer execução do *framework*, a fim de que o mesmo gere os códigos para os dialetos implementados.
- Criar ambiente, nos bancos de dados escolhidos, para testar o código gerado;
- Verificar e validar os resultados obtidos;
- Realizar reajustes no *framework*, se necessário.

Definindo os objetivos, o planejamento está descrito abaixo.

4.2 Planejamento

Para alcançar os objetivos listados anteriormente é preciso traçar uma estratégia de execução. Começando pela seleção dos bancos de dados, ou seja, os dialetos da linguagem SQL seguindo pela implementação das classes utilizando o *framework* e testando se o código gerado faz a carga desejada.

Os dialetos que serão utilizados para teste do *framework* já foram definidos desde o início do projeto, será o dialeto do SQL Server, T-SQL, e o do Oracle, PL/SQL. Estes foram escolhidos devido à grande popularidade e pelo fato de ambos terem características diferentes entre si. Muitos outros dialetos SQL existentes assemelham-se aos mesmos, ajudando, desta forma, na veracidade dos testes a serem realizados.

Para implementação das classes, que utilizarão o *framework*, devem ser observados os pontos de extensão e quais métodos específicos a classe, de cada dialeto, precisa implementar.

Após implementar as classes e verificar a possibilidade da geração de código, é preciso testá-los. Para isto, serão criados cenários de uso para cada banco de dados. Ou seja, serão criadas tabelas auxiliares e suas respectivas dimensões para realizar a execução do procedimento com as mesmas. Inicialmente serão criados ambientes que tenham armazenamento de histórico até o tipo 2.

Diante disto, será possível verificar se foi possível plugar uma classe para implementação de um determinado dialeto SQL e, caso resultado positivo, se, com a utilização do procedimento gerado, a carga dos dados foi realizada de forma correta, validando ou não a eficiência do *framework*. Com o resultado, caso necessário, o *framework* poderá sofrer ajustes.

4.3 Operação

A operação para a realização do estudo de caso foi dividida em duas etapas. A etapa de Execução e a etapa de Validação dos dados. Cada uma delas é descrita a seguir.

4.3.1 Execução

Para realizar a execução dos objetivos especificados foi necessário, primeiramente, implementar as duas classes concretas, com especificidades do T-SQL e PL/SQL. Estas estenderiam a classe abstrata, na qual já estava implementado os passos comuns aos dois dialetos. Para isto, foi preciso analisar o ponto de extensão do *framework* e seus métodos abstratos, os quais necessitariam de implementação nas duas classes concretas, para verificar se realmente seria possível a implementação para tal dialeto utilizando o *framework.*

A implementação seguiu de acordo com o exemplo abaixo, demonstrado no Quadro 1, Quadro 2 e Quadro 3:

```
public abstract class AbstractGerarProcedimento {

public abstract String DeclaraCursor();
...

public final void templateMethodGeraProcedimento(String caminhoArquivo){

...

String declaracaoCursor = DeclaraCursor();
...

}
}
```

Quadro 1: Classe Abstrata – *AbstractGerarProcedimento.*

```
public class GerarOracle extends AbstractGerarProcedimento{

private Procedimento procedimento;
...

public GerarOracle(Procedimento procedimento, ...){
      this.procedimento   = procedimento;

      ...
}
...
    public String DeclaraCursor(){
        String nomeCursor = "Cursor_"+procedimento.getNome_procedimento();
        return "DECLARE CURSOR "+ nomeCursor+" IS ";
    }
...
}
```

Quadro 2: Classe concreta que contem especificações do Oracle.

```
public class GerarSqlServer extends AbstractGerarProcedimento{

private Procedimento procedimento;
...

public GerarOracle(Procedimento procedimento, ...){
      this.procedimento   = procedimento;

      ...
}
...
    public String DeclaraCursor(){
       String nomeCursor = "Cursor_"+procedimento.getNome_procedimento();
        return "DECLARE "+ nomeCursor+" CURSOR FOR ";
    }
...
}
```

Quadro 3: Classe concreta que contem especificações do SQLServer.

Existe ainda a Classe Concreta *MetodosComplementares*, que serve como auxilio para a classe Abstrata *AbstractGerarProcedimento*, podendo ser chamada pela *GerarOracle* ou *GerarSqlServer*. Ela foi chamada, por exemplo, para devolver uma lista de *Strings* contendo os atributos da tabela auxiliar, já com seus respectivos tipos. Esses dados podem ser necessários também para outra classe que queira estender o *framework*. Por este motivo, foi implementado na Classe dos métodos complementares, a fim de evitar a duplicação de código.

4.3.2 Validação dos Dados

Além da validação inicial, realizada na Seção 3.5 Testes, foi realizada uma validação mais específica durante a execução do estudo de caso. Para esta, foram criados cenários de uso, ou seja, foram criadas tabelas auxiliares e dimensões, em SQL Server e Oracle. Os dados que compõem estas tabelas foram cadastrados em uma ferramenta já existente, *FGCod – SQL* (Santos et al., 2012). Após cadastro destes metadados, o procedimento pode ser gerado através da mesma.

Abaixo, na Figura 18, é ilustrada a tela da ferramenta responsável por informar os dados que farão parte da estrutura do procedimento a ser gerado.

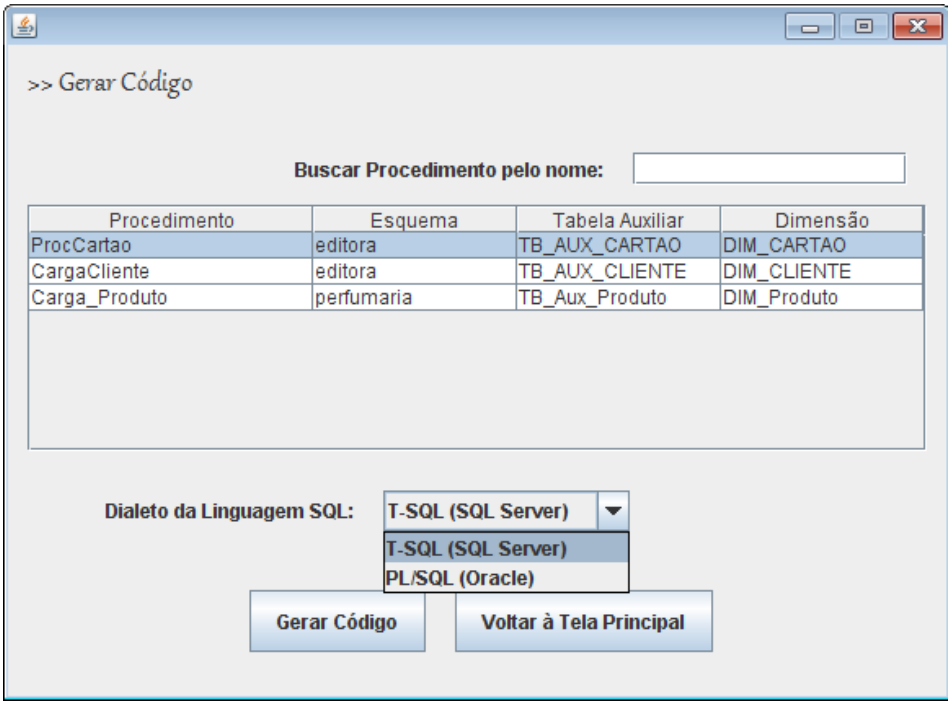

Figura 18: Tela inicial da ferramenta para dar suporte ao *framework.*

Após a geração dos procedimentos, em ambos os dialetos, os mesmos foram executados em seus SGBD's (Sistema Gerenciador de Banco de Dados) a fim de testar se o código gerado atendeu à sua função. Ambos os testes obtiveram êxito, ou seja, a carga dos dados aconteceu de forma correta.

5. CONCLUSÕES

A principal contribuição deste trabalho foi propor um *framework* de suporte a ferramentas de geração de código automática para geração de procedimentos SQL, os quais fazem carga da área de *Staging* para o DW. Para consolidação do trabalho e avaliação do processo de extensão do *framework* desenvolvido, foi realizado um estudo de caso, no qual foram escolhidos dois dialetos da linguagem SQL, o T-SQL e o PL/SQL. O desenvolvimento se deu com base no padrão de projeto *Template Method*. Foi construída uma classe abstrata, na qual se encontrava todos os pontos em comum entre tais dialetos da linguagem SQL, para que mais tarde pudesse ser estendida pelas classes que implementariam as especificidades de cada dialeto. Para geração do procedimento, torna-se necessária a obtenção dos metadados, os quais se encontram num banco de dados, que, neste caso, foi preenchido através de uma ferramenta desenvolvida anteriormente, a FGCod (Santos et al., 2012).

O código da FGCod também foi reaproveitado através da criação de uma classe concreta como auxílio à classe abstrata do *framework*. Esta classe instancia e repassa os dados para as classes estendidas.

De acordo com o proposto inicialmente e ao final do trabalho, pode-se perceber que os objetivos do mesmo encontravam-se atingidos. Foram realizados estudos sobre a arquitetura de um *Data Warehouse* e os principais conceitos referentes ao desenvolvimento de um *framework.* Através dos quais, foram aplicados, de forma prática, os conhecimentos teóricos adquiridos durante toda pesquisa.

Considerando as validações de dados descritas na seção 4.3 Operação, ficou evidente a possibilidade do uso do *framework* para dar suporte à geração de código utilizado na realização de cargas de um DW, em diferentes dialetos da linguagem SQL.

O desenvolvimento deste trabalho proporcionou um enorme aprendizado sobre *frameworks* e outros conceitos relacionados, tais como reutilização de software e padrões de projeto, bem como os conceitos referentes à arquitetura de um DW, processo de carga para criação do mesmo e estrutura de diferentes dialetos da linguagem SQL, em especial o T-SQL e o PL/SQL. O maior desafio encontrado foi em modelar as funcionalidades genéricas, de forma que o *framework* atendesse aos diferentes dialetos em questão.

Este projeto trouxe algumas lições importantes, como o aumento da capacidade de pesquisa de forma interdisciplinar e uma perspectiva diferente sobre a forma de desenvolvimento de um *framework* e a reutilização de código.

5.1 Trabalhos Futuros

Para melhorar sua utilização, os *frameworks* podem se tornar cada vez mais abrangentes no seu domínio. Logo, como consequência deste trabalho, é possível vislumbrar trabalhos futuros, como, por exemplo, desenvolver *frameworks* também para geração de código SQL para povoamento de fatos, agregados e hierarquias.

REFERÊNCIAS

COLAÇO JÚNIOR, M. **Projetando sistemas de apoio à decisão baseados em *data warehouse***. Rio de Janeiro: Axcel Books, 2004.

GAMMA, E.; JOHNSON R.; HELM R.; VLISSIDES J. **Padrões de Projeto**: **Soluções reutilizáveis de software orientado a objeto.** Porto Alegre: Bookman, 2005.

GIMENES, Itana M. De S.; TRAVASSOS, Guilherme H.. **O Enfoque de Linha de Produto para Desenvolvimento de Software**. In: XXII Congresso da Sociedade Brasileira de Computação, 2002, Florianópolis. Anais. Florianópolis: Hotel Castelmar e Parthenon, 2002. p. 1-32.

IMHOFF, C.; GALEMMO, N.; GEIGER. J. G. **Mastering *Data Warehouse* Design: Relational and Dimensional Techniques**. Indianapolis: Wiley Publish, Inc., 2003.

INMON, W. H. **Building the *data warehouse,* Fourth Edition.** 4. ed. Indianapolis, Indiana: Wiley Publishing Inc., 2005.

JOHNSON, R. E. **Documenting *frameworks* using patterns**. SIGPLAN Notices, New York, v. 27, 1992. Trabalho apresentado na OOPSLA, 1992.

KIMBALL, R; ROSS, M. **The *data warehouse* toolkit: The complete Guide to Dimensional Modeling.** 2. ed. John Wiley and Sons, Inc., 2002.

KIMBALL, R. **The *Data Warehouse* ETL Toolkit.** 1 ed. Wiley India (P) Ltd., 2004.

KIMBALL, R; ROSS, M.; THORNTHWAITE, W. **The *data warehouse* lifecycle toolkit**. 2. ed. Indianapolis, Indiana: Wiley Publishing Inc., 2008.

LANDIN, Niklas; NIKLASSON, Axel. Development of Object-Oriented *Frameworks*. Disponível em: <acques.dsc.ufcg.edu.br/cursos/map/recursos/developing-frame.pdf> . Acesso em: outubro de 2012.

MENDONÇA, Alércio B. Dória. ***Framework* Orientado a Objetos: Uma evolução em Engenharia de Software.** 2002.

OMG. **OMG Unified Modeling Language Specification Version 1.5 March 2003**. Disponível em : <www.omg.org>. Acesso em: outubro de 2012.

ROSS, M. **Kimball University: Slowly Changing Dimensions Are Not Always as Easy as 1, 2, 3. 2005.** Disponível em: <http://www.kimballgroup.com/2005/03/10/slowly-changing-dimensions-are-not-always-as-easy-as-1-2-3/> Acesso em Outubro, 2012.

SANTOS, I. P. O.; COSTA, J. K. G.; NASCIMENTO, A. V. R. P.; COLAÇO JÚNIOR, M. **Desenvolvimento e Avaliação de uma Ferramenta de Geração Automática de Código para Ambientes de Apoio à Decisão**. In: XII WTICG, Workshop de Trabalhos de Iniciação Científica e de Graduação Bahia-Alagoas-Sergipe, 2012, Juazeiro. XII Escola Regional de Computação Bahia Alagoas Sergipe - ERBASE 2012.

SANTOS, V; BELO, O. **No Need to Type Slowly Changing Dimensions.** IADIS International Conference Information Systems, 2011a.

SANTOS, V; BELO, O. **Slowly Changing Dimensions Specification a Relational Algebra Approach**. Proc. of Int. Conf. on Advances in Communication and Information Technology, 2011b.

SAUVÉ, Jacques Philippe. **Métodos Avançados de Programação**. Disponível em: <http://www.dsc.ufpb.br/~jacques/cursos/map/html/map2.htm>. Acesso em: setembro de 2012.

SILVA, M.S.; TIMES, V. C.; KWAKYE, M.M.. **A *Framework* for ETL Systems Development.** Journal of Information and Data Management, Vol. 3, No. 3, Outubro, 2012.

SINGH, R.; SINGH, K. **A Descriptive Classification of Causes of Data Quality Problems in *Data Warehouse***. IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 3, No 2, May 2010.

SOMMERVILLE, Ian. **Software Engineering**. 8º Ed. Pearson Addison-Wesley, 2007.

THOMSEN C.; PEDERSEN T. B. **pygrametl: a powerful programming *framework* for extract-transform-load programmers**. Proceedings of the ACM international workshop on Data warehousing and OLAP. Hong Kong, China, 2009.