PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Bacharelado em Ciência da Computação

CONSTRUÇÃO DE UM FRAMEWORK PARA AUXÍLIO NA CRIAÇÃO DE APLICAÇÕES DE EXTRAÇÃO TRANSFORMAÇÃO E CARGA DE DADOS PARA ARMAZÉM DE DADOS

Guilherme Almeida Gonçalves

Poços de Caldas
2011

Guilherme Almeida Gonçalves

CONSTRUÇÃO DE UM FRAMEWORK PARA AUXÍLIO NA CRIAÇÃO DE APLICAÇÕES DE EXTRAÇÃO TRANSFORMAÇÃO E CARGA DE DADOS PARA ARMAZÉM DE DADOS

Monografia apresentada ao Curso de Ciência da Computação da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Luiz Alberto Ferreira Gomes

Poços de Caldas
2011

A minha família e a minha namorada,
pela confiança, apoio, incentivo, compreensão e inspiração.

AGRADECIMENTOS

Agradeço a Deus, pelas portas abertas na minha vida e por ter me guiado em cada momento durante o desenvolvimento desse trabalho.

A meu orientador Luiz Alberto Ferreira Gomes, pela atenção dada sempre que possível e por ter direcionado todo o trabalho, com grande dedicação, competência e paciência.

Aos meus professores, colegas de classe e colegas de trabalho por estarem sempre dispostos a ajudar, incentivar, trocar experiencias e conhecimento.

A minha família pelo inestimável apoio e pela confiança depositado em mim, durante toda a execução do trabalho.

A minha namorada pela compreensão, mas acima de tudo, pela inspiração que sempre foi fundamental durante essa caminhada.

A todos que, de alguma forma contribuíram para realização desse trabalho.

Muito obrigado a todos.

“Tenha coragem de seguir o que seu coração e sua intuição dizem, eles de alguma maneira já sabem o que você realmente quer se tornar. O resto é secundário.”

Steve Jobs

RESUMO

Este projeto, visa a construção de um framework, que auxilie desenvolvedores na construção de aplicações de extração, transformação e carga de dados, para armazéns de dados. O framework da suporte a extração de arquivos texto de tamanho fixo e arquivos texto delimitados, pois eles não possuem as informações sobre os seus dados. E também, da suporte a extração de arquivos XML, pois eles necessitam de um outro arquivo para descrever sua estrutura. Para isso, o framework implementa classes para auxiliar o seu usuário, a fazer o mapeamento desses arquivos, extrair os seus dados e fazer a carga desses dados em uma base de dados relacional.

Palavras chave: Armazem de dados, extração de arquivos texto, gerador de instruções SQL.

ABSTRACT

This project aims to make a framework that helps developers on building extraction, transformation and load application for data warehouses. The framework supports the extraction of fixed length files and delimited files, because they dont have information about their data. And also supports the extraction of XML files, because they need another file to describe their structure. To do that, the framework implements classes to help its user, to map these files, to extract their data and to load them into a relational data base.

Key words: Data Warehouse, file extraction, SQL instruction bulder.

LISTA DE FIGURAS

Figura 1 - Arquitetura básica de um Data Warehouse 17
Figura 2 - Layout exemplo para arquivos de tamanho fixo.....18
Figura 3 - Framework de caixa branca.....22
Figura 4 - Framework de caixa preta.....23
Figura 5 - Ciclo do TDD.....28
Figura 6 - Diagrama de pacotes do framework 30
Figura 7 - Diagrama de domínio das classes de mapeamento de arquivos.....31
Figura 8 - Trecho de código utilizando as classes de mapeamento do framework ...32
Figura 9 - Diagrama de domínio das classes de extração 35
Figura 10 - código utilizando as classes de extração e de carga 36
Figura 11 - Diagrama de domínio das classes de criação de intruções 37
Figura 12 - Exemplo de teste automatizado.....41
Figura 13 - Gráfico de trabalho acumulado 41

LISTA DE QUADROS

Quadro 1 - Tipos de campo definidos pela classe FieldType 35

LISTA DE SIGLAS

DTD – Document Type Definition

ETL – Extraction Transformation and Load

SQL – Structured Query Language

TDD – Test Driven Development

XML – Extensible Markup Language

SUMARIO

1 INTRODUÇÃO 12
1.1 **Justificativa 13**
1.2 **Objetivo 13**

2 CONCEITOS BÁSICOS 15
2.1 **Processo de ETL e conceitos relacionados 15**
2.1.1 ***Considerações iniciais 15***
2.1.2 ***Data Warehouse 15***
2.1.3 ***Processo de extração transformação e carga de dados 16***
2.1.1.1 Extração 17
2.1.3.1.1 Fontes de dados em arquivos texto 18
2.1.3.1.2 Fontes de Dados (Arquivos XML) 19
2.1.1.2 Limpeza 19
2.1.1.3 Transformação 20
2.1.1.4 Carga 20
2.1.4 ***Considerações sobre o processo de ETL 20***
2.2 **Frameworks 21**
2.2.1 ***Características de framework 21***
2.2.2 ***Inversão de controle 23***
2.2.3 ***Considerações sobre frameworks 24***
2.3 **Padrões de projetos 24**
2.3.1 ***Considerações sobre os padrões de projeto 25***
2.3.2 ***Padrões de projeto e frameworks 25***
2.4 **Técnicas de desenvolvimento de software 26**
2.4.1 ***Refatoração 26***
2.4.2 ***Desenvolvimento dirigido por testes 27***
2.4.3 ***Controle de versão 28***
2.4.4 ***Kanban 29***

3 O TRABALHO DESENVOLVIDO 30
3.1 **Estrutura e funcionamento do framework 30**
3.1.1 ***Mapeamento dos arquivos 31***
3.1.1.1 Máscara de arquivos 33
3.1.1.2 Máscara de Registro 33
3.1.1.3 Máscara de Campo 34
3.1.1.4 Tipos de Campo 34
3.1.2 ***Extrator 35***
3.1.3 ***Mecanismo de Persistência 37***
3.1.4 ***Construtores de Instruções 37***
3.1.5 ***Exceções 38***
3.1.6 ***Leitor de XML 38***
3.1.7 ***Relacionamento das classes do framework 38***
3.2 **Desenvolvimento do framework 39**
3.3 **Técnicas utilizadas no desenvolvimento do framework 40**

4 CONCLUSÕES 43

4.1 **Considerações iniciais** **43**
4.2 **Contribuições** **43**
4.3 **Trabalhos Futuros** **43**
4.4 **Considerações finais** **44**

REFERÊNCIAS **45**

ANEXO A – DIAGRAMA DE CLASSES DE PROJETO DO FRAMEWORK **47**

1 INTRODUÇÃO

Na atual economia, as mudanças são continuas e ocorrem de forma muito rápida, assim, as empresa precisam tomar decisões rápidas. A Ciência da Computação, tem apoiado as empresas através de um conjunto de ferramentas e metodologias, conhecidas como *Business Inteligence,* que usa a extração de dados da própria organização, para gerar informações e auxiliar nos processos decisórios da mesma.

Uma das ferramentas mais importantes sob o conceito de *Business Inteligence* é o *Data Warehouse,* que trata do armazenamento, manutenção e recuperação de dados históricos das organizações. Durante o projeto de um *Data Warehouse*, a etapa mais importante e mais critica é o processo que extrai os dados das fontes sem altera sua estrutura, transforma-os de acordo com o modelo lógico do *Data Warehouse* e então os carrega no *Data Warehouse,* deixando-os disponíveis para consulta. Esse processo pode definir o sucesso ou o fracasso do projeto e consome facilmente setenta por cento dos recursos do projeto e manutenção do *Data Warehouse.*

Muitas operações do processo de extração transformação e carga de dados, se repetem uma quantidade considerável de vezes, mostrando, a carência de um conjunto de ferramenta, que contenha essas operações comuns ao desenvolvimento do processo.

Sendo assim, esse projeto visa a construção de um conjunto de classes, denominado *framework,* que implemente essas operações básicas, referente a algumas das necessidades do processo, auxiliando na construção de aplicações que o executam. Como o processo é muito complexo e abrangente o *framework* desenvolvido, é focado apenas nas fontes de dados em formato de texto e XML (*Extensible Markup Language*), e atende as necessidades referentes às etapas de extração e carga de dados, com maior foco na etapa de extração.

Este documento descreve no capitulo dois os aspectos e conceitos relacionados ao processo de extração transformação e carga de dados. Também são abordados, os conceitos sobre *frameworks,* bem como, os padrões que podem ser utilizados no seu desenvolvimento. Além disso são descritos algumas das praticas de desenvolvimento de software, utilizadas para aumentar a qualidade, do

framework construído durante o desenvolvimento do trabalho. No capitulo três é mostrado e explicado sobre o *framework* implementado, característica relacionadas ao seu funcionamento e como foi o seu processo de desenvolvimento. E finalizando o documento o capitulo quatro fala sobre as conclusões obtidas durante o trabalho.

1.1 **Justificativa**

Durante o projeto de *Data Warehouses* o processo de extração transformação e carga de dados é o que consome a maioria dos recursos, tendo isso em vista, existe a necessidade de reduzir esse consumo de recursos, para que as equipes que estão desenvolvendo projetos referente a esse assunto possam aumentar sua eficiência.

Um dos problemas quase sempre enfrentados no projeto, é a necessidade de trabalhar com arquivos texto pois os mesmos não possuem metadados, sendo assim a necessidade de fazer um mapeamento desses arquivos, de forma a encontrar a melhor representação possível dos dados neles contidos.

Já os arquivos XML são estruturados, sendo que essa estrutura consome até noventa por cento de seu tamanho, e além disso, eles necessitam de um outro arquivo para descrevê-los, tornando-se um outro problema para a equipe de desenvolvimento do *Data Warehouse.*

A necessidade de aumentar a eficiência na extração dos arquivos texto e XML se consolida a principal justificativa desse trabalho.

1.2 **Objetivo**

Analisando a necessidades do desenvolvimento de aplicações que executam o processo de extração, transformação e carga de dados,o objetivo desse trabalho é oferecer como produto final um *framework,* capaz de fazer o mapeamento de arquivos texto de tamanho fixo, arquivos texto delimitados e arquivos XML, e auxiliar na extração desses arquivos, implementando operações fundamentais do processo,

além de auxiliar na carga dos dados no *Data Warehouse,* diminuindo a interação do desenvolvedor com a base de dados relacional. Tudo isso de forma a permitir que um desenvolvedor ao implementar uma aplicação com o objetivo de extrair esses tipos de arquivo não necessite de se preocupar com as características fundamentais do domínio de problema mas apenas com as características especificas de sua aplicação.

2 CONCEITOS BÁSICOS

Este capitulo tem o objetivo de fazer uma contextualização sobre o processo de ETL, o desenvolvimento de *frameworks* e também algumas técnicas que podem ser utilizadas durante o desenvolvimento de uma aplicação.

2.1 **Processo de ETL e conceitos relacionados**

2.1.1 ***Considerações iniciais***

Atualmente a economia e caracterizada por continuas e rápidas mudanças e oportunidades, assim para uma empresa ser bem sucedido é necessário tomar decisões rápidas. Para tomar as melhores decisões e necessário analisar as condições passadas e presentes da empresa tão bem quanto as condições da economia atual e previsões para o futuro (WREMBEL ET AL, 2007). Neste cenário a Ciência da Computação tem assumido o papel de apoiar as empresas através do *Business Inteligence*. O *Business Inteligence* é um conjunto de ferramentas e metodologias que usa a extração de dados de uma organização para auxiliar nos processos decisórios gerenciais e da alta administração (ANTONELLI, 2009).

2.1.2 ***Data Warehouse***

Uma das mais importantes tecnologias sobre o conceito de *Business Inteligence* é o *Data Warehouse*. O *Data Warehouse* trata o armazenamento, manutenção e recuperação de dados históricos (TEOREY ET AL, 2007). O *Data Warehouse* é diferente dos sistemas transacionais que dão apoio a negócios em nível operacional, armazenando informações necessárias no dia a dia de uma empresa e eliminando dados antigos, sendo assim os sistemas transacionais

possuem tabelas relativamente pequenas. Já os *Data Warehouses* recebem periodicamente dados históricos em lotes aumentando seu tamanho com o tempo, tamanho esse que pode chegar a Terabytes de Informação. Esse enorme tamanho dos *Data Warehouse* é uma dificuldade no seu projeto e implementação, pois e necessário fazer consultas rápidas em grandes quantidades de dados (TEOREY ET AL, 2007).

O *Data Warehouse* é utilizado por um sistema de apoio a decisão e é alimentado por diferentes fontes de. As fontes de dados são utilizadas por sua própria aplicação, sendo assim autônomo e heterogêneo. Segundo Teorey (2007) essa heterogeneidade significa que os dados são de aplicações possivelmente com implementação em diferentes plataformas, oferecem diferentes funcionalidades, possuem diferentes técnicas de armazenamento, diferentes modelo de dados, dentre outras diferenças sendo necessário então antes de carregar os dados para o *Data Warehouse* reconciliar o modelo de dados através do processo de extração transformação e carga de dados ou comumente conhecido como processo de ETL (*Extraction, Transformation and Load*).

2.1.3 ***Processo de extração transformação e carga de dados***

O processo de ETL é fundamental no projeto de um *Data Warehouse*, pois é durante o processo que o projeto se consolida, mas é também, onde há a maior possibilidade do projeto falhar (KIMBALL ET AL, 2004).

O processo de ETL é responsáveis pela preparação dos dados a serem armazenados em um *Data Warehouse* (ANTONELLI, 2009). Esse processo requer uma analise minuciosa dos aspectos da empresa e essa analise pode ser difícil, demorada e ineficiente, porém necessária (WREMBEL ET AL, 2007). Segundo Kimball (2004) todo esse esforço pode facilmente consumir setenta por cento dos recursos do projeto e manutenção do *Data Warehouse*. Geralmente o processo de ETL é dividido em três etapas: extração dos dados sem alterar sua estrutura, transformação dos seguindo o modelo do *Data Warehouse* e Carga dos dados deixando-os prontos para consulta. A figura 1 ilustra essas três etapas.

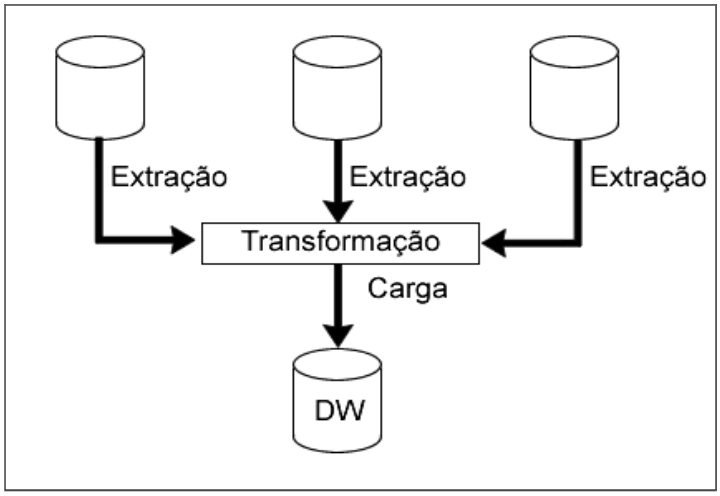

Figura 1 - Arquitetura básica de um Data Warehouse
Fonte: TEOREY, 2007

Mas, segundo Kimball (2004) uma quarta etapa pode ser adicionada antes da etapa de transformação, sendo essa responsável pela limpeza dos dados. Cada uma das etapas é explicada a seguir.

2.1.1.1 Extração

Quando se inicia o projeto de um *Data Warehouse* a primeira etapa é a extração de dados, essa extração é feita em diferentes sistemas computacionais, fazendo com que antes de extrair os dados dessas diferentes fontes, seja necessário definir o relacionamento entre os eles. O Mapa lógico de dados é o artefato utilizado para descrever esse relacionamento, sendo que ele deve detalhar o relacionamento entre as fontes de dados, o modelo de dados pretendido para o *Data Warehouse*, e as manipulações necessárias para transformar os dados originais para o formato de destino. Criar o mapa lógico de dados é importante, pois, a extração física demanda mais esforço e caso não tenha sido bem planejada pode colaborar para o fracasso do projeto (KIMBALL ET AL, 2004).

Depois de ter o mapa lógico de dados a extração deve segundo Antonelli (2009) trazer os dados das diferentes fontes da empresa, sendo que esses dados deve sofrer o minimo possível de alterações de reestruturação.

Durante a extração serão encontrados diferentes tipos de fontes, cada uma em um formato diferente, plataforma diferente e as formas de se conectar aos dados também são diferentes (KIMBALL ET AL, 2004). Durante o desenvolvimento desse trabalho serão abordados apenas a extração das fontes texto e XML.

2.1.3.1.1 Fontes de dados em arquivos texto

Na maioria dos ambientes de *Data Warehouses* não é possível escapar dos arquivos texto O *Data Warehouse* utiliza arquivos texto por três motivos: receber dados entregues pelas fontes de dados, criação de tabelas temporárias, preparação para carregar dados em massa. Um problema acoplado ao uso desses arquivos é o fato que eles não possuem dados sobre a informação contida nele, necessitando de um tratamento especifico (KIMBALL ET AL, 2004). Existem dois tipos de arquivo, os com comprimento fixo,e os delimitado.

Os arquivos de comprimento fixo são criados para representar as fonte de dados quando não é possível acessá-la diretamente. Para processar os arquivos de tamanho limitado é necessário um definir um *layout* de arquivo que ilustre exatamente o arquivo contendo dados como o nome do arquivo, onde os campos começam, o tamanho, o tipo do dado e onde o campo termina (KIMBALL ET AL, 2004). A figura 2 mostra um exemplo de como pode ser esse *layout*.

Nome do Campo	Tamanho	Inicio	Fim	Tipo
Tipo do Registro	2	1	2	Alfa Numérico
Número do Seguro Social	9	3	11	Numérico
Primeiro Nome	20	12	31	Alfa Numérico
Inicial do Nome do Meio	1	32	32	Alfa Numérico
Último Nome	20	53	52	Alfa Numérico
Sufixo do Nome	5	58	57	Alfa Numérico
Aniversário	8	66	65	Numérico
Status	6	72	71	Numérico
Código do Escritório	2	74	73	Numérico
Código do Departamento	2	76	75	Numérico
Código do Cargo	2	78	77	Numérico

Figura 2 - Layout exemplo para arquivos de tamanho fixo
Fonte: KIMBALL, 2004

Na maioria dos casos o *layout* do arquivo de comprimento fixo deve ser definido manualmente na primeira vez e sempre que houver qualquer alteração. Após o *layout* ser definido quando o arquivo for inserido novamente a aplicação deve ser capaz de lembrar do *layout* do arquivo (KIMBALL ET AL, 2004).

Os arquivos delimitados utilizam um caractere especial (pode ser qualquer caractere) para definir onde os campos iniciam e onde eles terminam. O mais utilizado e a virgula (,). A primeira linha dos arquivos delimitados geralmente vem com o nome das colunas e a aplicação deve prever isso (KIMBALL ET AL, 2004).

2.1.3.1.2 Fontes de Dados (Arquivos XML)

Os arquivos XML são basicamente arquivos texto, a diferença está na sua estrutura, e é essa estrutura que torna o XML tão poderoso, mas é essa também sua principal desvantagem pois noventa por cento do seu tamanho é composto por metadados. Para extração de dados em arquivos XML é necessário uma aplicação e um outro arquivo que descreva a estrutura do arquivo XML. Esse documento é chamado de Document Type Definition (DTD) ou atualmente um outro documento vem sendo usado, o XML schema que é capaz de armazenar os mesmos dados, porém, é mais poderoso pois é capaz de armazenar tipos de dados e instruções SQL (*Structured Query Language*) (KIMBALL ET AL, 2004).

2.1.1.2 Limpeza

Esta etapa ocorre juntamente com a de transformação, sendo que são as etapas mais importantes do processo de ETL, agregando o maior valor aos dados. Durante a etapa de limpeza, acontece a verificação de valores válidos, a remoção de dados duplicados, a aplicação das regras de negócio e a geração de metadados capaz de diagnosticar o que há de errado nos dados que vem das fontes. É recomendado também que haja um julgamento humano durante a limpeza de dados (KIMBALL ET AL, 2004).

2.1.1.3 Transformação

Em projetos de *Data Warehouse* sempre é requerido que duas o mais fontes de dados sejam integradas em um único local físico, utilizando suas melhores informações para que seja possível uma melhor visualização dos dados extraídos das fontes. Essa integração acontece durante a etapa de transformação, onde os dados das fontes são transformados para que estejam em um único modelo, o modelo de dados do *Data Warehouse* definido anteriormente no mapa lógico de dados (KIMBALL ET AL, 2004).

2.1.1.4 Carga

A etapa de carga é o momento em que todo o esforço das etapas anteriores é entregue ao usuário final de acordo com o modelo dimensional e de forma que estejam pronto para consultas (KIMBALL ET AL, 2004).

2.1.4 ***Considerações sobre o processo de ETL***

Em resumo o objetivo do processo de ETL é entregar dados mais eficazes adicionando valor aos dados nas etapas de limpeza e transformação dos dados, além de documentar a linha do tempo dos dados (KIMBALL ET AL, 2004).

O processo de ETL merece uma atenção especial pois, embora não tenha muita visibilidade dos usuários finais, é a base do projeto de um *Data Warehouse*, se bem desenvolvido pode determinar o sucesso do projeto e da mesma forma caso seja mal desenvolvido irá determinar o fracasso do projeto (KIMBALL ET AL, 2004).

2.2 **Frameworks**

Quando uma aplicação está sendo desenvolvida e notasse que algumas funcionalidades podem ser compartilhadas em diversos pontos diferentes da aplicação, é o momento em que se percebe a necessidade de criar um conjunto de classes que possam ser reutilizadas, o que é definido como *framework* (GUERRA, 2010).

Um *framework*, é definido por Larman (2004) como um conjunto de componentes extensíveis, que fornecem as funcionalidades básicas e constantes para atender à um domínio de problema, oferecendo um alto grau de reusabilidade de código.

Sendo assim o *framework* pode e geralmente define, a arquitetura e o *design* da aplicação, particionando as classes e os objetos, definindo o relacionamento entre eles e as suas responsabilidades. Através desse trabalho o *framework* permite que o usuário do *framework* possa manter o foco nas tarefas e características especificas de sua aplicação (GAMMA ET AL, 1998).

Segundo Gamma (1998) um *framework* é responsável por definir o *design* que é comum para o domínio de suas aplicações, obrigando ao desenvolvedor que siga um padrão na escrita das operações, e consequentemente diminui as decisões necessárias sobre o *design* da aplicação.

Desenvolver um *framework* é complicado, pois, o projetista precisa garantir que uma arquitetura funcionará para todas as aplicações do domínio de problema, por isso é necessário que haja um baixo acoplamento entre as classes, evitando que qualquer mudança no seu design possa diminuir os seus benefícios (GAMMA ET AL, 1998).

2.2.1 ***Características de framework***

Existem dois tipos de *frameworks*: caixa branca e caixa preta. Nos *framework* de caixa branca o reuso de código acontece através da herança. Dessa forma para que o usuário possa utiliza os serviços do *framework*, ele deve criar subclasses que

estendam as classes abstratas contidas no *framework* para criar aplicações específicas. Para tal, ele deve entender detalhes de como o *framework* funciona para poder usá-lo. Já os *framework* de caixa preta o reuso de código acontece por composição, ou seja, o usuário combina diversas classes concretas existentes no *framework* para obter a aplicação desejada. Sendo assim, ele deve entender apenas a interface do *framework* para poder usá-lo (MALDONADO ET AL, 199-).

Para ser possível exemplificar o comportamento dos dois tipos de *frameworks* é necessário antes, compreender o comportamento de seus componentes.

Os componentes de um *frameworks* podem ser classificados em pontos de especialização e pontos fixos. Os pontos de especialização, geralmente referenciado como *hot spots*, são os aspectos que variam em cada aplicação num mesmo domínio de problema. Os *hot spots* são projetados para serem genéricos de forma que possam se adaptar às necessidades da aplicação.

Os pontos fixos, conhecidos como *frozen spots* definem a arquitetura geral de um sistema de software, seus componentes básicos e os relacionamentos entre eles. Os *frozen spots* permanecem fixos em todas as instanciações do *framework* de aplicação (MALDONADO ET AL, 199-).

A figura 3 mostra um *framework* de caixa branca com um *hot spot*.

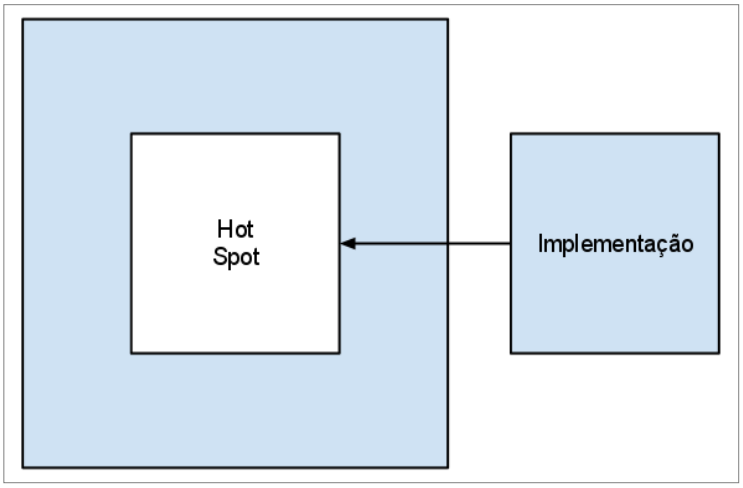

Figura 3 - Framework de caixa branca
Fonte: Maldonado et al, 199-

A figura mostra como o *framework*, exige uma implementação externa para utilização do *hot spot*.

A figura 4 mostra um *framework* de caixa preta também com um *hot spot*.

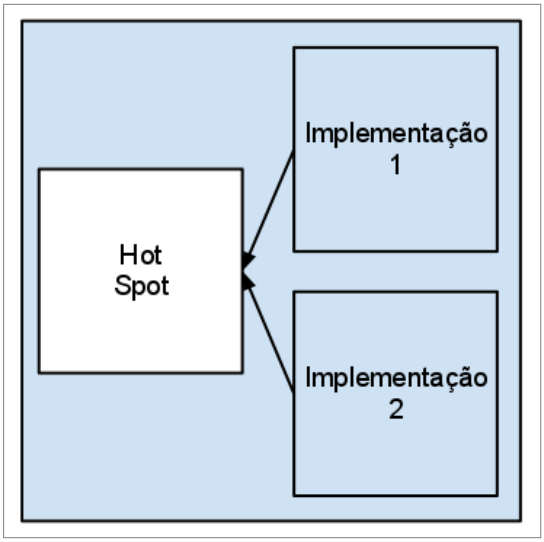

Figura 4 - Framework de caixa preta
Fonte: Maldonado et al, 199-

O ponto interessante no *framework* de caixa preta, e que todas as implementações possíveis para atender o *hot spot* são internas, o que torna mais difícil a sua implementação em relação ao *framework* de caixa branca, pois o projetista precisa prever as possibilidades de implementação que atendam as necessidades do domínio de problema(MALDONADO ET AL, 199-).

2.2.2 ***Inversão de controle***

Os *frameworks*, podem exigir que o desenvolvedor implemente um conjunto de subclasses de suas classes, para personalizar os serviços do *framework*. Neste caso, as classes definidas pelo desenvolvedor receberam mensagens predefinidas pelo *framework*, isso é conhecido como Principio de Hollywood, que diz, “Não ligue para mim, nós ligaremos pra você” (LARMAN, 2004).

2.2.3 ***Considerações sobre frameworks***

Pode-se dizer que a essência de um *framework* não está nas funcionalidades que ele implementa mas sim o quando é possível estender essas funcionalidades de forma que elas se adaptem as necessidades da aplicação (GUERRA, 2010).

Para o sucesso de um *framework* também é necessário que ele funcione de forma simples e rápida sem necessidade de grandes configurações ou de fazer o download de novas bibliotecas, tornando-o atraente aos programadores. Porém *frameworks* que implementam muitas funcionalidades são mais complexos, o ponto importante é que o desenvolvedor não deve ter contato com coisas complexas quando for implementar coisas simples, permitindo que o desenvolvedor tenha contato com funcionalidades simples e com o tempo passe a utilizar funcionalidades mais complexas (GUERRA, 2010).

2.3 **Padrões de projetos**

Alcançar a reusabilidade de código no desenvolvimento de um *framework* ou qualquer aplicação orientada a objeto não é fácil, definir uma herança e uma hierarquia pertinente entre as classes, e evitar a redundância de código, são tarefas tão difíceis que alguns profissionais experientes na modelagem de sistemas dizem ser quase impossível conseguir isso na primeira tentativa. O que esses profissionais fazem é utilizar soluções que já funcionaram pra eles em projetos passados. Isso faz com que seja possível identificar um padrão recorrente nas classes utilizadas, e também na organização entre elas. Alguns desses padrões de projetos se firmaram na área de desenvolvimento de software, tornando desnecessário aos desenvolvedores reinventá-los (GAMMA ET AL, 1998).

Padrões de projetos são soluções utilizadas várias vezes para atender a problemas no mesmo contexto (FOWLER, 2002). Existem várias definições para padrões de projeto, mas, a mais conhecida foi feita por Christopher Alexander, citado por Martin Fowler (2002, p.29) e por Erich Gamma (1998, p.14), onde ele diz que, cada padrão descreve um problema que ocorre várias vezes em um mesmo

ambiente, então esse padrão descreve também, as principais partes de um solução, que pode ser usada milhões de vezes, sem se repetir duas vezes que seja. Os padrões de projetos podem ser classificados em três diferente categorias (GAMMA ET AL, 1998):

a) Padrões de Criação, que tornam a criação, composição e representação do objeto independente;
b) Padrões de Estrutura, que através de herança faz com que classes e objetos componham estruturas maiores;
c) Padrões de Comportamento são focados na definição de responsabilidade entre objetos, descrevendo não só o padrão de classes e objetos, mas também o padrão de comunicação entre eles.

2.3.1 ***Considerações sobre os padrões de projeto***

Os padrões de projeto não são a estrutura final de um código, pois não ha como prever todas as variações de codificação que um sistema pode ter, assim os padrões de projeto devem ser apenas um ponto de partida para uma implementação (FOWLER, 2002).

A maioria dos grandes sistemas utilizam padrões de projeto, isso permite que a compreensão do seu código com maior facilidade (GAMMA ET AL, 1998).

Outro beneficio trazido pelos padrões de projetos, é que as pessoas envolvidas, podem falar sobre a estrutura de classes dos sistemas, em auto nível, pois os padrões criam um vocabulário comum facilitando a comunicação (GAMMA ET AL, 1998).

2.3.2 ***Padrões de projeto e frameworks***

Frequentemente os *frameworks* são confundidos com os padrões de projeto, porém, existem três características que os diferenciam (GAMMA ET AL, 1998):

a) Padrões de projetos são mais abstratos do que *frameworks*, pois eles podem ser implementados, fato que apenas ocorre com exemplos de padrões de projetos;
b) Padrões de projetos são elementos menores que *frameworks*: Um *framework* comum contém diversos padrões de projetos, porém o inverso nunca acontece;
c) Padrões de projetos são menos especializados do que *frameworks*: Cada *framework* possui um domínio de aplicação. Já padrões de projetos podem ser utilizado em quase todo tipo de aplicação, mesmo aqueles que são mais específicos.

2.4 **Técnicas de desenvolvimento de software**

O desenvolvimento software não é uma atividade mecânica ou definida, mas sim, uma atividade complexa. Por possuir essa característica, o mesmo precisa estar apoiado sobre processos disciplinados de desenvolvimento de software, ou falhas e problemas podem acontecer. Quando se diz que falhas podem acontecer, isso não se limita a falhas de funcionamento do software, mas também problemas com, como o software é desenvolvido, que podem aumentar o tempo necessário para sua implementação e diminuindo a qualidade do seu código (PRESSMAN, 2001).

Para o enriquecimento desse trabalho e aumento na qualidade do produto desenvolvido, algumas práticas de desenvolvimento de software foram utilizadas, sendo que essas são descritas nas sessões seguintes.

2.4.1 ***Refatoração***

Refatoração é o processo disciplinado de aperfeiçoamento da estrutura interna de um software sem alterar suas funcionalidades e seu comportamento externo, em outras palavras refatorar é tornar o código do software mais organizado

fazendo com que ele comunique o seu propósito de forma clara e também com que possa receber novas funcionalidades com maior facilidade (FOWLER, 2004).

Em um projeto de software com o passar do tempo é natural que o código fique mais complexo. Essa complexidade diminui a produtividade dos desenvolvedores. Nesse contexto a refatoração é necessária para a reorganização do código, e aumentar novamente a produtividade e descobrir falhas no software (FOWLER, 2004).

É aconselhado que a refatoração seja feita em pequenos passos, assim, caso o desenvolvedor cometa um erro, ele pode ser facilmente identificado (FOWLER, 2004).

Uma boa prática que deve ser seguida paralelamente à de refatoração é a prática de desenvolvimento dirigido a testes que será apresentada na sessão seguinte, pois, tendo bons testes para garantir o comportamento externo do software, a refatoração pode ser executada com maior segurança (FOWLER, 2004).

2.4.2 ***Desenvolvimento dirigido por testes***

Quando um software esta sendo desenvolvido, muitas interferências acontecem, sendo que essas podem nos desviar de uma implementação com um código limpo e que funcione corretamente. A prática de desenvolvimento guiado por testes, ou comumente chamada de TDD (*Test Driven Development*), apoia o desenvolvimento de software nessa questão (BECK, 2010).

O TDD é uma prática que organiza o desenvolvimento, em ciclos de três etapas. A primeira etapa é escrever um teste, onde é definido, quais os elementos envolvidas na operação, e qual deve ser o resultado da operação, dado o comportamento que se espera da mesma. Ao executarmos esse teste, ele deve falhar, pois o código necessário para que ele funcione ainda não foi implementado. Quando a primeira etapa é terminada, o ciclo se encontra na posição chamada "vermelho", para chegarmos a próxima posição do ciclo conhecida como "verde", e preciso implementar a solução mais simples que atenda o teste escrito. Nessa etapa não é necessário se preocupar em implementar uma solução definitiva a solução definitiva deve surgir com alguns ciclos, implementando mais testes e evoluindo a

funcionalidade. A terceira etapa do ciclo e a refatoração do código, de forma a torná-lo mais simples, e fazer com que ele expresse a sua função, conforme foi descrito na sessão anterior. A figura 5 a seguir ilustra o ciclo do TDD (BECK, 2010).

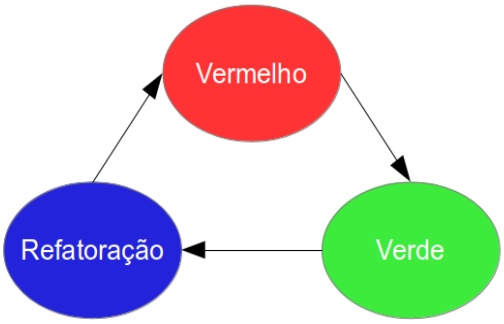

Figura 5 - Ciclo do TDD
Fonte: Elaborado pelo autor.

O TDD traz uma serie de benefícios, como por exemplo, a quantidade de defeitos do código é reduzida e consequentemente há um aumento de qualidade do software entregue ao cliente. Surpresas desagradáveis também podem ser reduzidas, diminuindo o prazo de entrega do software. Além disso, com testes que garantem as funcionalidades do código existente, a segurança para alterar o código, ou mesmo inserir novas funcionalidades aumenta (BECK, 2010).

2.4.3 ***Controle de versão***

Quando um usuário navega por um sistema, uma opção de desfazer suas ultimas ações é sempre de grande ajuda. No contexto do desenvolvimento de software, muitas vezes, uma alteração é feita que acaba comprometendo o seu funcionamento. Então um sistema de controle de versão é para os desenvolvedores como uma tecla de desfazer, permitindo-os voltar a versões anteriores do código sempre que desejado (HUNT ET AL, 2010).

Porém, um bom sistema de controle de versão deve disponibilizar, funções que permitem identificar a diferença entre o código da versão atual em relação a

qualquer outra versão, identificar onde foram feitas as alterações feitas na ultima versão, e quais são elas, dentre outras informações desse tipo, que dão uma noção maior, sobre a linha do tempo do código do software desenvolvido (HUNT ET AL, 2010).

Uma outra funcionalidade encontradas em sistemas de controle de versão e a de permitir que ramificações de um código sejam geradas, permitindo que cada uma delas possa evoluir isoladamente e se juntarem no futuro, sendo que é responsabilidade do sistema de controle de versão a mesclagem dessas ramificações (HUNT ET AL, 2010).

2.4.4 ***Kanban***

Kanban é uma técnica de organização de trabalho, utilizado inicialmente pelo sistema Toyota de produção, onde se divide o trabalho em pequenas tarefas, e associa-os a cartões, assim uma etapa do trabalho só pode ser iniciada quando houver cartões disponíveis (ANDERSON, 2010).

No desenvolvimento de software, o kanban se tornou popular, especialmente nas metodologias de desenvolvimento ágil, como uma ferramenta de controle visual do fluxo e da carga do trabalho, permitindo que uma equipe de desenvolvimento de software possa acompanhar o trabalho em desenvolvimento, e se auto gerenciar (ANDERSON, 2010).

A utilização do kanban traz uma série de benefícios, como por exemplo, auxilia na identificação e no balanceamento da carga de trabalho, permite identificar gargalos no fluxo de trabalho além de auxiliar na extração de algumas métricas como eficiência, throughput, lead time, informações sobre o trabalho acumulado e o tipo de trabalho que está sendo executado (ANDERSON, 2010).

3 O TRABALHO DESENVOLVIDO

Este capítulo tem como objetivo descrever o *framework* desenvolvido durante o trabalho, detalhando seus aspectos conceituais, suas características técnicas bem como o seu funcionamento além de descrever como foram as etapas do seu desenvolvimento.

3.1 **Estrutura e funcionamento do framework**

O *framework* desenvolvido nesse trabalho, visa, auxiliar desenvolvedores de aplicações ETL, a mapear arquivos texto e XML, extrair os seus dados e carrega-los no *Data Warehouse*. Para uma melhor compreensão da estrutura e o funcionamento do *framework*

implementado, antes de descrever cada uma de suas funcionalidades, a figura 6, mostra através de um diagrama de pacotes, uma visão geral sobre sua estrutura.

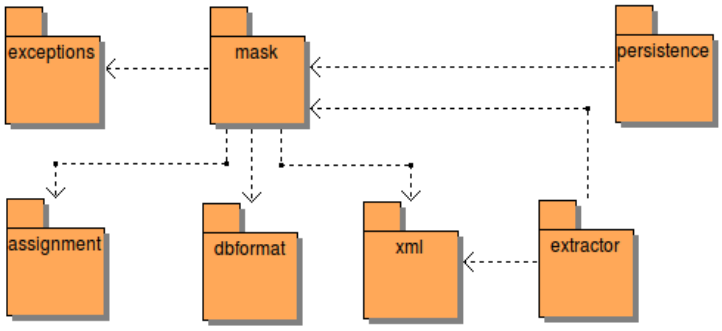

Figura 6 - Diagrama de pacotes do framework
Fonte: Elaborada pelo autor

O pacote *mask* é o que contem as classes mais importantes do *framework*, responsáveis por fazer o mapeamento dos arquivos, dando suporte também as classes contidas no pacote extractor e *persistence*, sendo essas responsáveis respectivamente por extrair os dados dos arquivos e carregar-los no *Data*

Warehouse. As classes no pacote *assignment* são responsáveis por registrar as pendencias de alteração da tabela na base de dados relacional, sempre que houver modificação no arquivo de entrada. O pacote xml, é responsável apenas por auxiliar as classes dos pacotes *mask* e extractor no reconhecimento dos arquivos xml. E por fim o pacote *exceptions* que identifica as exceções na leitura dos arquivos. As sessões seguintes descrevem de forma mais detalhada o funcionamento, a estrutura e as responsabilidades de cada parte do *framework*.

3.1.1 ***Mapeamento dos arquivos***

O framework foi implementado de forma a suportar como entrada, arquivos texto de tamanho fixo, arquivos texto delimitados, e arquivos XML, conforme as necessidades do processo de ETL, levantadas por Kimball e citadas no inicio do capitulo dois.

Os arquivos devem conter, registros referentes a dados extraídos de uma determinada aplicação. Cada um desses registros, é composto por um conjunto de campos. Então, para fazer o mapeamento dos arquivos, são implementadas no pacote *mask*, classes que definem a máscara dos arquivos, dos registros contidos nos arquivos, e dos campos que compõe os registros. Para cada formato de arquivo suportado, o próprio *framework* implementa uma especialização dessas classes, definindo as implementações que serão utilizadas para atender o *hot spot*, seguindo então, o conceito de composição em *frameworks* descrito no capítulo três. Sendo assim, para o desenvolvedor mapear os arquivos em sua aplicação, ele apenas precisa instanciá-las em sua aplicação.

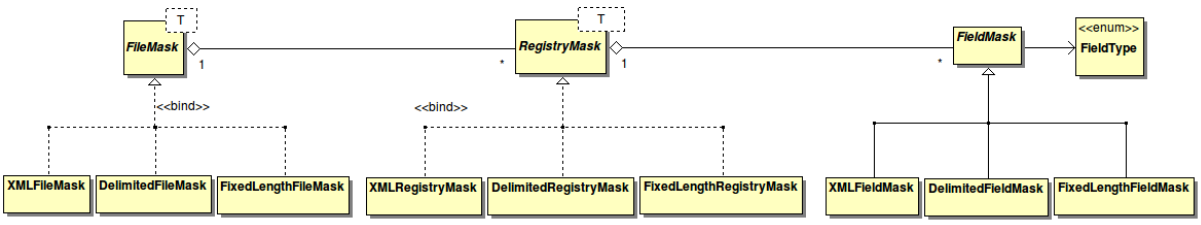

Figura 7 - Diagrama de domínio das classes de mapeamento de arquivos
Fonte: Elaborada pelo autor

Assim como se da a estrutura do arquivo, acontece na utilização das classes, onde, uma instância de uma especialização da classe *FileMask* responsável por definir a máscara de arquivo deve possuir uma ou mais instâncias de uma das especializações da classe *RegistryMask* responsável por definir a máscara de registro, uma, para cada tipo de registro contido no arquivo, isso permite que o framework mapeie arquivos, que contenham diferentes tipos de registro. Seguindo a estrutura, cada instância das classes responsáveis pro definir a máscara de registro, deve possuir, uma ou mais instancias de uma das especializações da classes FieldMask, responsável por definir as máscaras de campo, sendo uma, para cada unidade de informação contida no registro.

A figura 8 exemplifica a utilização dessas classes no mapeamento de um arquivo delimitado.

```
DelimitedFileMask mascaraDeArquivo;
DelimitedRegistryMask mascaraDeAluno;
DelimitedFieldMask mascaraDoTipo;
DelimitedFieldMask mascaraDoNome;
DelimitedFieldMask mascaraDoCPF;

mascaraDeArquivo = new DelimitedFileMask("PROF-ALUNO", ';', "1.0", "Arquivo de Professores e Alunos");

mascaraDeAluno = new DelimitedRegistryMask("Alunos", "1.0", "Aluno");

mascaraDoTipo = new DelimitedFieldMask("Tipo", 0, 9, 0, FieldType.A,true, false);
mascaraDoNome = new DelimitedFieldMask("Nome", 1, 35, 0, FieldType.A,false, false);
mascaraDoCPF = new DelimitedFieldMask("CPF", 2, 14, 0, FieldType.A,false, true);

mascaraDeAluno.addFieldMask(mascaraDoTipo);
mascaraDeAluno.addFieldMask(mascaraDoNome);
mascaraDeAluno.addFieldMask(mascaraDoCPF);

mascaraDeArquivo.addRegistryMask(mascaraDeAluno);
```

Figura 8 - Trecho de código utilizando as classes de mapeamento do framework
Fonte: Elaborado pelo autor.

Note que na figura, os valores dos parâmetros que descrevem cada elemento, estão sendo passados diretamente, mas cabe ao desenvolvedor da aplicação, escolher a forma com que esses dados devem ser encontrados.

Conforme já dito, para cada um dos arquivos suportados, o framework implementa uma especialização das classes de mapeamento de arquivo, registro e campo, pois cada tipo de arquivo, possui características diferentes.

3.1.1.1 Máscara de arquivos

As máscaras de arquivo não possuem muitas diferenças quanto aos dados necessários do arquivo para sua instanciação. Em geral elas precisam receber apenas do código usado para descrever o arquivo, o numero da versão e uma breve descrição. Exceto no caso de arquivos delimitados onde é necessário definir qual é o caractere utilizado para delimitar cada campo.

Quando é requisitado às máscaras que façam o mapeamento de um conjunto de dados, elas também se comportam de maneira semelhante. Na verdade elas apenas identificam qual é o registro referente aos dados recebidos, e então, delegam a máscara de registro, para que a mesma retorne os valores. A única diferença a se notar, é que as máscaras de arquivo texto recebem uma linha do arquivo para ser mapeada, e a máscara de arquivo XML, recebe uma instancia de um elemento XML, que permite trabalhar com um conjunto de tags que representam um registro.

3.1.1.2 Máscara de Registro

As classes de máscara de registro são responsáveis por mapear um tipo de registro a cada instância. Para sua criação é necessário informar o nome da tabela que irá receber os dados, a versão do registro e uma breve descrição. Exceto pela máscara de registros XML, que além desses dados precisa saber qual é o nome da *tag* que representa esse registro.

De forma geral, as máscaras de registro se comportam da mesma forma, exceto por pequenos tratamentos de características particulares. Por exemplo, os arquivos texto que contem mais de um tipo de registro, necessitam de um campo para identificar o registro. Já no caso dos arquivos XML essa identificação pode ser feita através do nome da *tag* que representa o registro, dispensando a necessidade de um campo.

A principal função das máscaras de registro, é fazer a leitura de uma entrada de dados (uma linha no caso dos arquivos texto ou um conjunto de *tags* no caso do

arquivo XML) e retornar um objeto com a informação mapeada. Com esse objeto, em conjunto com as classes de formatação de instruções, é possível gerar as instruções SQL para de criação de tabela, alteração de tabela e inserção de dados, tornando assim, a interação do desenvolvedor com a base de dados relacional mais transparente.

3.1.1.3 Máscara de Campo

A máscaras de campo são as mais simples, das três máscaras utilizadas no *framework*, porém não menos importantes. Elas são responsáveis, por descrever um campo armazenando dados como, tipo, tamanho, o nome respectivo ao campo na base de dados relacional, se o campo é chave primária e se o campo é o campo utilizado para identificar o registro (no caso dos arquivos texto).

A diferença que se encontra em cada uma das especializações, é a informação armazenada para identificar o campo no arquivo. Para os arquivos texto de tamanho fixo, é necessário informar em que posição da linha o campo começa e em que posição ele termina. Já nos arquivos texto delimitados é preciso saber qual o índice do campo (tendo que a linha se comporta como um vetor de dados). E por fim para os arquivos XML é preciso informar qual o nome da tag que representa o campo.

Quando um dado é passado a uma máscara de registro para ser mapeado (lembrando que a máscara de registro possui uma coleção de máscaras de campo), cada unidade de informação que representa um campo é armazenada em uma máscara de campo.

3.1.1.4 Tipos de Campo

Ainda no pacote *mask* existe a classe FieldType responsável por definir os tipos de campo suportados pelo framework, sendo sua principal utilização, na definição das máscaras de campo. Além disso o tipo de campo é utilizado para

auxiliar na criação das instruções SQL. O quadro 1 mostra os tipos de campo definidos pelo framework.

Nome	Descrição
A	Alfanumérico
N	Numérico
MM	Mês
DD	Dia
YYYY	Ano
HH	Hora
MI	Minuto
SS	Segundo
DATE1	Alfanumérico
DATE2	Data
HOUR	Hora
UF	Sigla

Quadro 1 - Tipos de campo definidos pela classe FieldType
Fonte: Elaborado pelo autor.

Os tipos de campos definidos no *framework*, tentam abranger os campos mais utilizados em aplicações mais comuns.

3.1.2 ***Extrator***

As classes de Extração estão contidas no pacote *extractor*, sendo que, da mesma forma que as máscaras, para cada tipo de arquivo suportado, existe uma especialização da classe *FileExtractor* como mostra a figura 9.

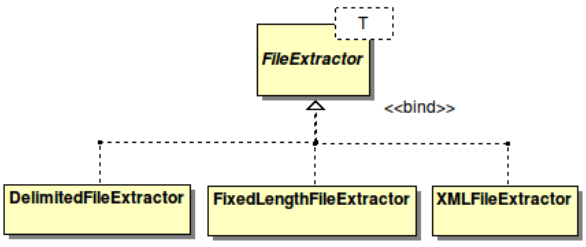

Figura 9 - Diagrama de domínio das classes de extração
Fonte: Elaborada pelo autor

As classes de extração são responsáveis por fazer a leitura de cada elemento do arquivo que represente um registro, no caso dos arquivos texto uma linha, e no caso dos arquivos XML um conjunto de *tags*. Após a leitura, o extrator deve retornar uma instância da respectiva especialização de *RegistryMask* que contenha os dados de elemento lido do arquivo.

Para utilizar as classes extratoras, basta que o usuário do *framework* instancie-as informando a localização do arquivo e qual a máscara desse arquivo. Com isso é possível fazer a navegação entre os registros dos arquivos, e também, extrair cada registro, conforme pode ser visto abaixo na figura 10.

```
XMLRegistryMask mascaraDeRegistro;

MapeadorDeArquivoXML mapeadorXML = new MapeadorDeArquivoXML();
XMLFileExtractor extratorXML = new XMLFileExtractor(
        "/home/guilherme/workspace/ETL-Framework/entrada.xml",
        mapeadorXML.getFileMaskDesign());

try {
    PersistenceMechanism.startConnection("jdbc:mysql://localhost/td", "root", "");
    do {
        mascaraDeRegistro = extratorXML.extractOne();
        PersistenceMechanism.insert(mascaraDeRegistro);
    } while (!extratorXML.next());
    PersistenceMechanism.finishConnection();
}
catch (UnkownRegistryException e) {e.printStackTrace();}
catch (SQLException e1) {e1.printStackTrace();}
```

Figura 10 - código utilizando as classes de extração e de carga
Fonte: Elaborada pelo autor

Note que na figura, foi definida uma classe MapeadorDeArquivoXML, responsável por retornar a máscara do arquivo, a forma com que essa máscara busca esses dados é de responsabilidade da aplicação.

É importante lembrar que segundo Kimball, essa máscara deve ser definida apenas na primeira vez que for feita a importação do arquivo ou caso haja modificações no mesmo.

3.1.3 ***Mecanismo de Persistência***

A classe responsável por fazer a carga dos dados lidos do arquivo no *Data Warehouse*, é a *PersistenceMechanism*, que está no pacote *persistence*. Essa classe é de forma bem simples faz um encapsulamento da comunicação com a base de dados, sendo capaz iniciar e encerrar uma conexão, além de, executar as instruções SQL, que são geradas por uma máscara de registro. Sua utilização deve ser feita em conjunto com as classes de extração, pois o método *extractOne*, retorna uma máscara de registro com os valores extraídos do arquivo, essa máscara de registro, é que será utilizada para geração das instruções, a serem executadas na base de dados relacional. Isso pode ser visto na figura 10.

3.1.4 ***Construtores de Instruções***

O pacote *dbformat* contem as classes responsáveis por gerar as instruções SQL de criação de tabelas, alteração de tabelas e inserção de dados, na base de dados relacional. Foi criada a classe *DataBaseFormat*, que contem a formatação básica e genérica para todas as instruções da linguagem SQL e para cada tipo de instrução foi criada uma especialização, como ilustra a figura 11.

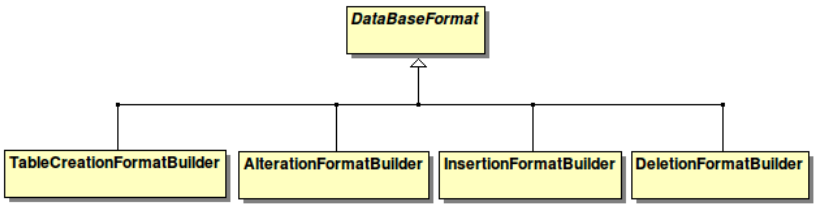

Figura 11 - Diagrama de domínio das classes de criação de intruções
Fonte: elaborada pelo autor

Essas classes são utilizadas pelas classes de máscara de registro, onde elas recebem o dado de cada campo do registro, para gerar a instrução de forma

adequada, respeitando tipagem e tamanho do campo, além de considerar, quais campos fazem chave primarias.

3.1.5 ***Exceções***

Com a criação das classes do pacote de mapeamento dos arquivos, surgiram também, algumas exceções. Essas exceções podem acontecer quando um registro lido de um arquivo não corresponda ao mapeamento feito para ele, ou quando o registro lido do arquivo não tenha sido mapeado.

O tratamento dessas exceções, são de responsabilidade do desenvolvedor que utilizar o *framework* no desenvolvimento de sua aplicação.

3.1.6 ***Leitor de XML***

Como o *framework* suporta arquivos XML, surgiu a necessidade de um leitor de arquivos XML. Algumas ferramentas já implementadas foram estudadas, e todas elas se mostraram muito complexas para a necessidade. Então foi implementado no pacote xml, um leitor de arquivos XML bastante simples que auxilia as operações executadas pelos pacotes *mask* e *extractor*.

3.1.7 ***Relacionamento das classes do framework***

Após a descrição da estrutura, do comportamento e da relação entre as funcionalidades do *framework*, é possível agora, fazer uma visão mais detalhada e com maior compreensão do *framework*. Assim o diagrama de classes de domínio na figura 12, tem como objetivo mostrar a relação entre as classes do *framework*.

Caso seja necessário uma maior compreensão o anexo 1 contem um diagrama de classes de projeto mostrando além da relação entre as classes, a estrutura de cada uma delas.

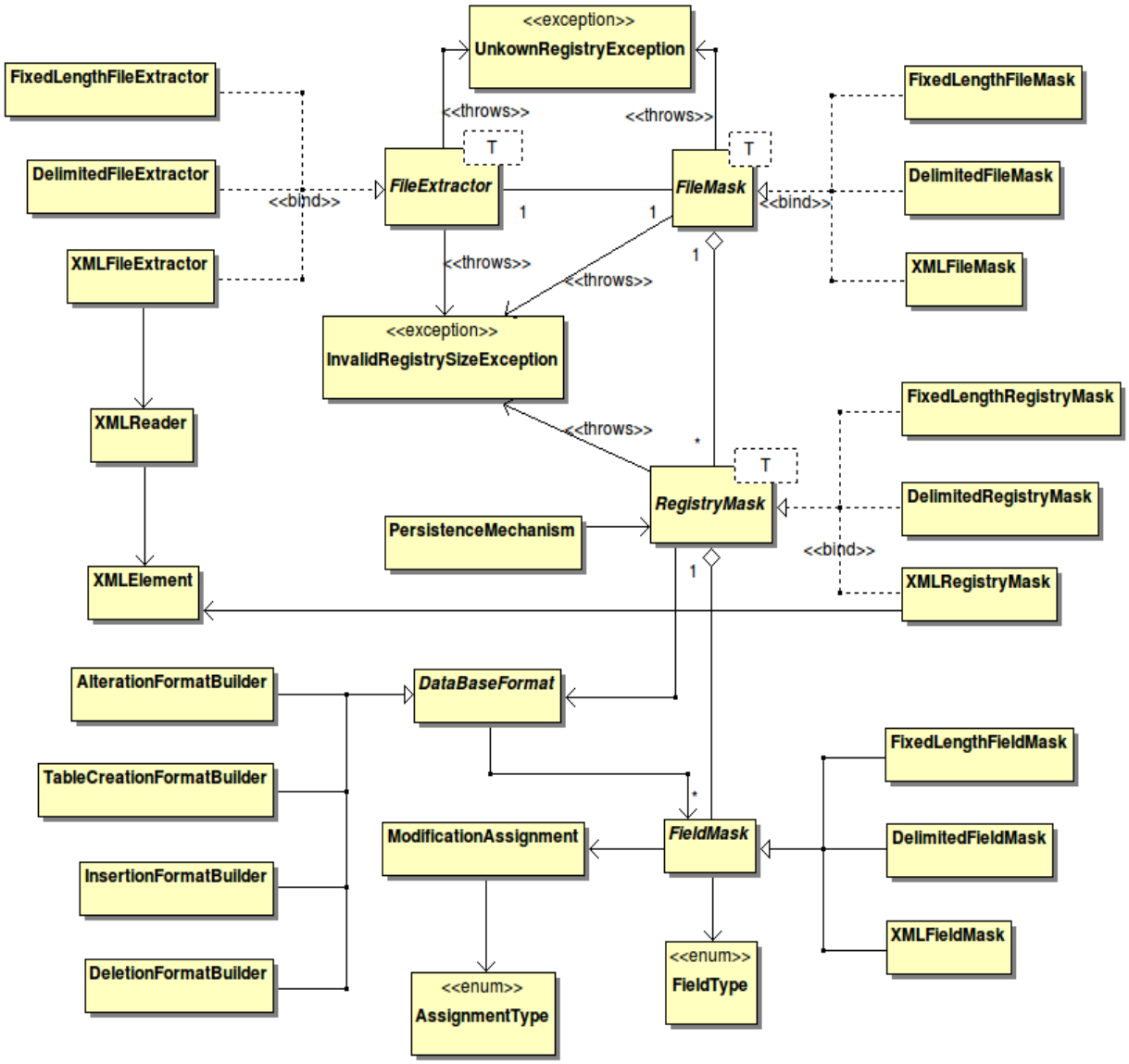

Figura 11 - Diagrama de classes de domínio do framework
Fonte: Elaborado pelo autor.

3.2 **Desenvolvimento do framework**

O desenvolvimento do *framework* se iniciou, com o mapeamento dos arquivos, pois essa foi a primeira evidência identificada. No começo, os arquivos

texto de tamanho fixo eram os único suportados. Nesse momento surgiu a necessidade de identificar as exceções que poderiam ser geradas, caso houvesse problema no arquivo lido ou no seu mapeamento.

Com o mapeamento de arquivos implementados surgiu então a necessidade de passar esses dados para a base de dados relacional, foi implementado então as classes responsáveis por gerar as instruções SQL de criação de tabelas, alteração de tabelas e inserção na base de dados relacional, porém, nessa etapa ainda não havia sido implementado um mecanismo de execução das instruções SQL, tendo essas, que serem passadas para a aplicação executá-las.

Então foram implementados as máscaras para os arquivos texto delimitados e para os arquivos XML, aumentando a abrangência do *framework*. Ao implementar o suporte a arquivos XML, foi necessário criar um interpretador básico de arquivos XML.

Com a parte de mapeamento dos arquivos e geração de instruções SQL implementada surgiu a necessidade de desenvolver uma classe responsável por, fazer a comunicação com a base de dados relacional e executar as instruções SQL geradas.

Por fim, foi implementado as classes que executavam a extração do arquivo concluindo os processos abordados pelo *framework*.

3.3 **Técnicas utilizadas no desenvolvimento do framework**

Durante o desenvolvimento do *framework*, com o intuito de reduzir o tempo gasto para encontrar e resolver falhas, aumentar a qualidade do código produzido e garantir seu funcionamento foi utilizado o TDD. Como todo o código do *framework* foi feito na linguagem Java, foi possível utilizar o *Junit 4*, uma das ferramentas mais consolidadas no assunto. O *Junit* permite a automatização e execução rápida da suíte de testes reduzindo o tempo que seria gasto com *debug* caso não estivesse sendo usado TDD. A figura 12 mostra um exemplo de teste automatizado.

Como prática de organização os casos de teste foram escritos em uma pasta diferente da que contem o código do *framework*, porém, mantendo a mesma estrutura de pacotes.

```
@Test
public void shouldModifyAssignmentState() {
    XMLFieldMask field = new XMLFieldMask("Field", "field", 10, 2, FieldType.A, true);
    ModificationAssignment assignment = new ModificationAssignment(AssignmentType.ALTER, false);

    field.addAssignment(assignment);

    assertFalse(field.getAlterationAssignment().get(0).isSolved());

    field.modifyAssignmentState(true);
    assertTrue(field.getAlterationAssignment().get(0).isSolved());
}
```

Figura 12 - Exemplo de teste automatizado
Fonte: Elaborado pelo autor.

A refatoração do código descrita anteriormente como uma das etapas do TDD, mas também, como uma prática que pode ser usada isoladamente, contribuiu para a qualidade do código do framework, pois, como Fowler disse, as vezes nossa mente se preocupa em fazer com que o software funcione, outras vezes, se preocupa com a qualidade dele.

Com o intuito de aumentar a segurança na alteração do código foi utilizado o git, uma das ferramentas mais completas de controle de versão. O *git* permite que sejam registradas todas as etapas da implementação do código, em um, ou mais repositórios, sendo possível recuperar quando necessário qualquer uma das versões do código. Para facilitar a integração das pessoas envolvidas no projeto, um dos repositórios utilizados para armazenar as versões do projeto, foi o *Github*. O *Github* é um serviço *web*, para compartilhamento de projetos que utilizam *git* para controle de versão.

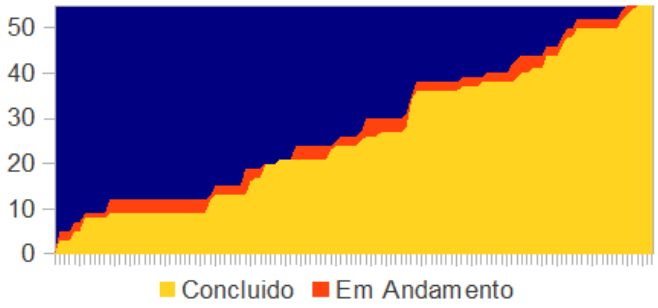

Figura 13 - Gráfico de trabalho acumulado
Fonte: Elaborada pelo autor

Para aumentar a visibilidade do projeto e ajudar no gerenciamento foi utilizado a técnica de kanban. O kanban auxiliou a priorizar as tarefas e também, deu visibilidade com relação ao prazo e a evolução do projeto. O gráfico da figura 13, mostra como foi essa evolução.

Todas essas práticas utilizadas nesse trabalho, dentre outras, são utilizadas para agregar valor, ao processo de desenvolvimento de *software*, é ao código do produto de *software* desenvolvido.

4 CONCLUSÕES

4.1 **Considerações iniciais**

Cada vez mais as empresas estão utilizando as ferramentas de *Business Inteligence*, afim de obter informações mais consistentes sobre o mercado, e sobre sua posição nele, permitindo assim, a criação de uma melhor estratégia de negócio.

O *framework* desenvolvido neste trabalho, tem o objetivo de facilitar a construção de uma das ferramentas sob esse conceito, as aplicações de ETL que fazem parte, do processo de construção de um *Data Warehouse.*

4.2 **Contribuições**

Observando as estrutura das classes implementadas no *framework*, notasse que ele entrega ao seu usuário, uma forma mais simples de fazer o mapeamento dos arquivos por ele suportados, e também, a fazer a extração e a carga dos dados desses arquivos. Outro ponto a se observar, é que o *framework* implementa apenas os procedimentos básicos e comuns para aplicações desse domínio de problema, dando liberdade para que seu usuário se preocupe com questões criticas de sua aplicação.

4.3 **Trabalhos Futuros**

Trabalhos futuros podem vir a incrementar as funcionalidades disponíveis no *framework.* Um bom ponto para iniciar seria implementar classes que auxiliassem na etapa de transformação do processo de ETL, o que acabou sendo uma carência deixada pelo *framework*.

Outras melhorias que podem ser feitas, e aumentar os recursos das classes de geração de instruções SQL, e também, aumentar a quantidade de fontes suportadas, um bom inicio seria fontes de dados relacionais.

4.4 **Considerações finais**

O *framework* então, atende uma necessidade inerente do assunto, mas o processo de ETL é muito complexo e abrangênte, mesmo para uma equipe inteira, com abundancia de recursos, o que tornou inviável para o projeto, tentar construir um *framework* que suprisse todas as necessidades do processo.

Outro ponto importante foi a utilização de práticas de desenvolvimento de software, que foram de grande vália, ao desenvolvimento do projeto. O TDD sem duvida foi a que mais impactou nos resultados do projeto. Ficou claro que o TDD não somente, da segurança na alteração do código, e diminui o tempo gasto para se identificar e resolver seus problemas no código, mas também, acaba interferindo no design da aplicação. Isso acontece pois o TDD propõe, que o desenvolvedor implemente a solução mais simples que atenda um simples teste, e então, evolua a solução, conforme a quantidade de testes aumenta e especifica melhor a necessidade do programa. Essa pratica de pensar na solução de forma incremental, em pequenos passos, é que interferiu no código implementado.

REFERÊNCIAS

ANDERSON, David. **Kanban –** Sucessful Evolutionary Change for Your Technology Business, Washington: Blue Hole, 2010.

ANTONELLI, Ricardo. Conhecendo o Business Inteligence (BI). **Revista TECAP**, Pato Branco, v.3 , n. 3, 2009s.

BECK, Kent. **TDD - Desenvolvimento Guiado por Testes**. Porto Alegre: Bookman, 2010. 240p.

FOWLER, Martin et al. **Patterns of Enterprise Application Architecture.** [s.l]: Addison Wesley, 2002. 560p.

FOWLER, Martin. **Refatoração**: aperfeiçoando o projeto de código existente. Porto Alegre: Bookman, 2004. 365p.

GAMMA, Erich et al. **Design Patterns**: elements of reusable object-oriented software. [s.l.]: Addison Wesley, 1998. 1 CD-ROOM

GUERRA, Eduardo; Herança e Composição os Princípios por Traz dos Padrões. **Revista Mundo J**, Curitiba, n. 039, p. 21-29, 01 de 2010.

HUNT, Andrew; THOMAS, David. **O Programa Pragmático:** de aprendiz a mestre. Porto Alegre: Bookman, 2010. 344p.

LARMAN, Craig. **Utilizando UML e padrões**: uma introdução à análise e ao projeto orientados a objetos e ao processo unificado. 2. ed. Porto Alegre: Bookman, 2004. 607p.

KIMBALL, Ralph; CONSERTA, Joe. **The Data Warehouse ETL Toolkit**: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data. Indianapolis: Wiley, 2004. 491p.

MALDONADO, José et al. **Padrões e Frameworks de Software.** São Carlos 199-. Disponível em: <www.icmc.usp.br/~rtvb/apostila.pdf> Acesso em 10 out. 2011.

PRESSMAN, Róger S. **Software engineering**: a practitioner's approach. 5 ed. New York: McGraw-Hill, 2001. xvii, 860p.

TEOREY, Toby J.; LIGHTSTONE, Sam; NADEAU, Tom. **Projeto e modelagem de bancos de dados**. Rio de Janeiro: Elsevier, 2007. 276p.

WREMBEL, Robert; KONCILIA, Christian (Ed.). **Data warehouses and OLAP**: concepts, architectures and solutions. Hershey: IRM, 2007. 332p.

ANEXO A – DIAGRAMA DE CLASSES DE PROJETO DO FRAMEWORK